Intel® Advanced Vector Extension AVX-AVX512 Tech Discussion

Intel HPCS 2015 Team

http://software.intel.com/sites/default/files/319433-016.pdf

Legal Disclaimers

- Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.
- Relative performance is calculated by assigning a baseline value of 1.0 to one benchmark result, and then dividing the actual benchmark result for the baseline platform into each of the specific benchmark results of each of the other platforms, and assigning them a relative performance number that correlates with the performance improvements reported.
- Intel does not control or audit the design or implementation of third party benchmarks or Web sites referenced in this document. Intel encourages all of its customers to visit the referenced Web sites or others where similar performance benchmarks are reported and confirm whether the referenced benchmarks are accurate and reflect performance of systems available for purchase.
- Intel® Hyper-Threading Technology Available on select Intel® Xeon® processors. Requires an Intel® HT Technology-enabled system. Consult your PC manufacturer. Performance will vary depending on the specific hardware and software used. For more information including details on which processors support HT Technology, visit http://www.intel.com/info/hyperthreading.
- Intel® Turbo Boost Technology requires a Platform with a processor with Intel Turbo Boost Technology capability. Intel Turbo Boost Technology performance varies depending on hardware, software and overall system configuration. Check with your platform manufacturer on whether your system delivers Intel Turbo Boost Technology. For more information, see http://www.intel.com/technology/turboboost
- Intel processor numbers are not a measure of performance. Processor numbers differentiate features within each processor series, not across different processor sequences. See http://www.intel.com/products/processor_number for details. Intel products are not intended for use in medical, life saving, life sustaining, critical control or safety systems, or in nuclear facility applications. All dates and products specified are for planning purposes only and are subject to change without notice
- Intel product plans in this presentation do not constitute Intel plan of record product roadmaps. Please contact your Intel representative to obtain Intel's current plan of record product roadmaps. Product plans, dates, and specifications are preliminary and subject to change without notice
- All dates and products specified are for planning purposes only and are subject to change without notice.
- *Other names and brands may be claimed as the property of others.

Optimization Notice

Optimization Notice

Intel® compilers, associated libraries and associated development tools may include or utilize options that optimize for instruction sets that are available in both Intel® and non-Intel microprocessors (for example SIMD instruction sets), but do not optimize equally for non-Intel microprocessors. In addition, certain compiler options for Intel compilers, including some that are not specific to Intel micro-architecture, are reserved for Intel microprocessors. For a detailed description of Intel compiler options, including the instruction sets and specific microprocessors they implicate, please refer to the "Intel® Compiler User and Reference Guides" under "Compiler Options." Many library routines that are part of Intel® compiler products are more highly optimized for Intel microprocessors than for other microprocessors. While the compilers and libraries in Intel® compiler products offer optimizations for both Intel and Intel-compatible microprocessors, depending on the options you select, your code and other factors, you likely will get extra performance on Intel microprocessors.

Intel® compilers, associated libraries and associated development tools may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include Intel® Streaming SIMD Extensions 2 (Intel® SSE2), Intel® Streaming SIMD Extensions 3 (Intel® SSE3), and Supplemental Streaming SIMD Extensions 3 (Intel® SSSE3) instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors.

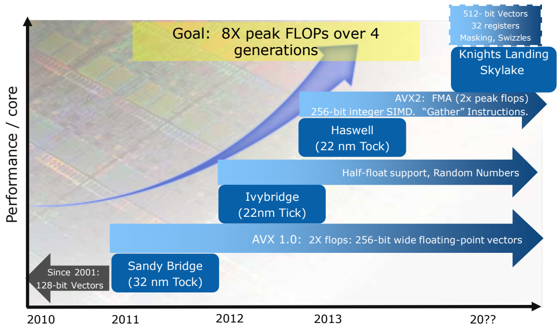
While Intel believes our compilers and libraries are excellent choices to assist in obtaining the best performance on Intel® and non-Intel microprocessors, Intel recommends that you evaluate other compilers and libraries to determine which best meet your requirements. We hope to win your business by striving to offer the best performance of any compiler or library; please let us know if you find we do not.

Notice revision #20101101

Agenda

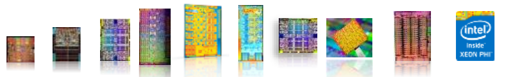
- Quick history!
- AVX feature discussion:
  - AVX overview – vision and blueprint
  - SW: Programming Models & Tools
- Deep dive: AVX1/2/AVX512 ISA
- Getting Started with AVX512
  - Compiler and Tools
  - Methodology and Framework
  - Case Study - American Options using Barone-Adsi Whaley
  - Summary

Intel® Advanced Vector Extensions


Goal: 8X peak FLOPs over 4 generations
512- bit Vectors
32 registers
Masking, Swizzles
Knights Landing
Skylake
AVX2: FMA (2x peak flops)
256-bit integer SIMD. "Gather" Instructions.
Haswell
(22 nm Tock)
Half-float support, Random Numbers
Ivybridge
(22nm Tick)
AVX 1.0: 2X flops: 256-bit wide floating-point vectors
Since 2001:
128-bit Vectors
Sandy Bridge
(32 nm Tock)
Performance / core
2010
2011
2012
2013
20??

Challenges to Application Software - Parallelism

Intel® Xeon® and Intel® Xeon Phi™ Product Families are parallel

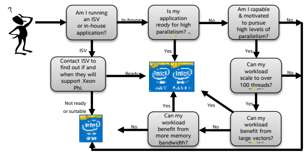


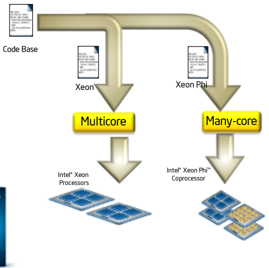
	Intel® Xeon® processor 64-bit	Intel® Xeon® processor 5100 series	Intel® Xeon® processor 5500 series	Intel® Xeon® processor 5600 series	Intel® Xeon® processor code-named Sandy Bridge EP	Intel® Xeon® processor code-named Ivy Bridge EP	Intel® Xeon® processor code-named Haswell EP	Intel® Xeon Processor codenamed Skylake EP	Intel® Xeon Phi™ coprocessor Knights Corner	Intel® Xeon Phi™ processor & coprocessor Knights Landing[1]
Core(s)	1	2	4	6	8	12	18	28	61	70+
Threads	2	2	8	12	16	24	36	56	244	280+
SIMD Width	128	128	128	128	256	256	256	512	512	512

More cores → More Threads → Wider vectors

*Product specification for launched and shipped products available on ark.intel.com. 1. Not launched or in planning.

Consistent Developer Tools and Programming Models


Am I running an ISV or in-house application?
In-house
ISV
Is my application *ready* for high parallelism? ...
No
Am I capable & motivated to pursue high levels of parallelism?
No
Yes
Contact ISV to find out if and when they will support Xeon Phi.
Ready
Can my workload scale to over 100 threads?
Yes
No
Not ready or suitable
Can my workload benefit from more memory bandwidth?
No
Yes
Yes
Can my workload benefit from large vectors?
No
Yes



Standards Parallel Programming Models Vectorize, Parallelize, & Optimize

AVX SW Tools

Servers and clients compute, media and throughput workloads performance is critically dependent on vectorization and parallelization

Intel is leading the deployment of technologies to

- Increase the amount of vectorized code
- Help you identify bottlenecks in your application
- Target next generation CPU's before having silicon

http://whatif.intel.com

Industry-Leading Tools

- Intel® C/C++ Compilers. *Industry leading Vector Programming*
- Intel® Integrated Performance Primitives
- Intel® Math Kernel Library
- Intel® Thread Building Blocks
- Intel® VTune Analyzer and
- Intel® Advisor XE

New Capabilities

Languages Extensions

- *Intel Cilk Plus*

New Languages

- OpenCL:CPU in 2010, CPU+GPU in 2011.

New Libraries

- Intel® Media SDK – HW Acceleration

New Analysis Tools

- Intel® Architecture Code Analyzer
- Intel® Software Tuning Agent
- Intel® Software Development Emulator

OpenCL is a registered trademark of Apple Computer, Inc.

Intel® AVX Technology

Sandy Bridge	**Haswell**	**Future** (in planning, subject to change)
256b AVX1 16 SP / 8 DP Flops/Cycle	256b AVX2 32 SP / 16 DP Flops/Cycle (FMA)	512b AVX512 Server: 64SP / 32 DP Client: 32 SP / 16 DP Flops/Cycle (FMA)

AVX	AVX2
256-bit basic FP	Float16 (IVB 2012)
16 registers	256-bit FP FMA
NDS (and AVX128)	256-bit integer
Improved blend	PERMD
MASKMOV	Gather
Implicit unaligned	

AVX512

512-bit FP/Integer
32 registers
8 mask registers
Embedded rounding
Embedded broadcast
Scalar/SSE/AVX "promotions"
Native media additions
HPC additions
Transcendental support
Gather/Scatter

SNB-2011 **HSW-2013** **Future Processor (Knight Landing & Skylake Xeon)**

AVX512 big picture

- AVX512F
  - 'Foundation' of architecture, required for any AVX512 implementation
    - Many D/Q/SP/DP promotions from AVX2 with AAVX512 features
      - Masking, 32 registers, embedded broadcast or rounding, 512-bit Vector Length
    - New instructions added to accelerate HPC workloads
  - Implementations add features to AVX512F "base"
    - "base" will grow as MIC/Xeon converge on features

	AVX512CD	Conflict Detect : instructions tailored for vectorizing loops with potential address conflicts
	AVX512ER	Exponential and Reciprocal : 'wide' approximateion of Log (22 bits) and RCP/RSQRT (28 bits)
	AVX512PF	Prefetch : Multi-address prefetch instructions using gather/scatter semantics
	AVX512DQ	Additional D/Q/SP/DP instructions (converts, transcendental support, etc)
	AVX512BW	512-bit Byte/Word support (promotions from AVX2, some additions)
	AVX512VL	Vector Length Orthogonality : ability to operate on sub-512 vector sizes

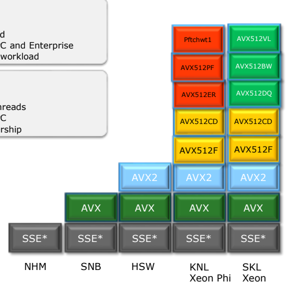
Xeon & Xeon Phi™ New ISA: What Is Where?

- Complex & versatile big cores
- Big focus on latency and single-thread
- State-of-the-art SIMD support for HPC and Enterprise
- Best balance of performance for any workload

- Small & efficient cores
- Big focus on throughput and many-threads
- State-of-the-art SIMD support for HPC
- Industry performance-per-watt leadership

KNL and SKL are the first *"unification"* platforms:

AVX512F is the common SIMD foundation for HPC software development



AVX-512 features (I): More & Bigger Registers

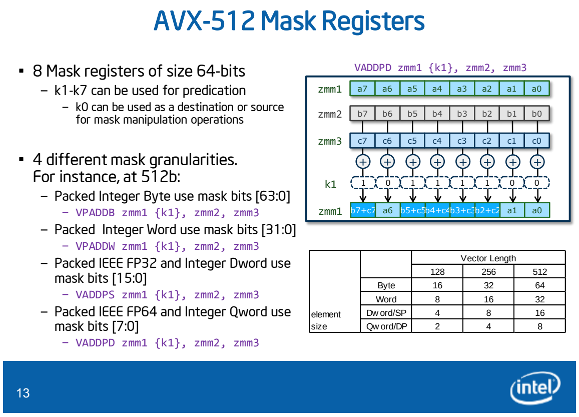
- **AVX**: VADDPS YMM0, YMM3, [mem]
  - Up to 16 AVX registers
    - 8 in 32-bit mode
  - 256-bit width
    - 8 x FP32
    - 4 x FP64

```
float32 A[N], B[N];

for(i=0; i<8; i++)
{
    A[i] = A[i] + B[i];
}
```

- **AVX-512**: VADDPS ZMM0, ZMM24, [mem]
  - Up to 32 AVX registers
    - 8 in 32-bit mode
  - 512-bit width
    - 16 x FP32
    - 8 x FP64

```
float32 A[N], B[N];

for(i=0; i<16; i++)
{
    A[i] = A[i] + B[i];
}
```

- But you need many more features to use all that real estate effectively…

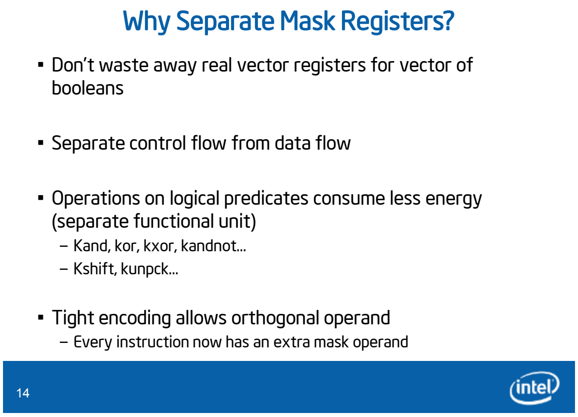


AVX-512 Mask Registers

- 8 Mask registers of size 64-bits
  - k1-k7 can be used for predication
    - k0 can be used as a destination or source for mask manipulation operations
- 4 different mask granularities.
  For instance, at 512b:
  - Packed Integer Byte use mask bits [63:0]
    - `VPADDB zmm1 {k1}, zmm2, zmm3`
  - Packed Integer Word use mask bits [31:0]
    - `VPADDW zmm1 {k1}, zmm2, zmm3`
  - Packed IEEE FP32 and Integer Dword use mask bits [15:0]
    - `VADDPS zmm1 {k1}, zmm2, zmm3`
  - Packed IEEE FP64 and Integer Qword use mask bits [7:0]
    - `VADDPD zmm1 {k1}, zmm2, zmm3`

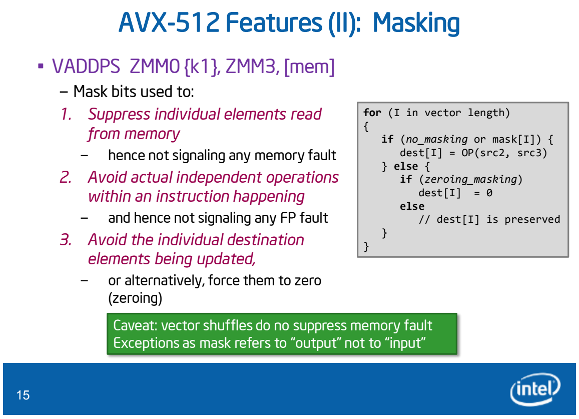
`VADDPD zmm1 {k1}, zmm2, zmm3`

		Vector Length		
		128	256	512
	Byte	16	32	64
	Word	8	16	32
element	Dw ord/SP	4	8	16
size	Qw ord/DP	2	4	8



Why Separate Mask Registers?

- Don't waste away real vector registers for vector of booleans
- Separate control flow from data flow
- Operations on logical predicates consume less energy (separate functional unit)
  - Kand, kor, kxor, kandnot…
  - Kshift, kunpck…
- Tight encoding allows orthogonal operand
  - Every instruction now has an extra mask operand

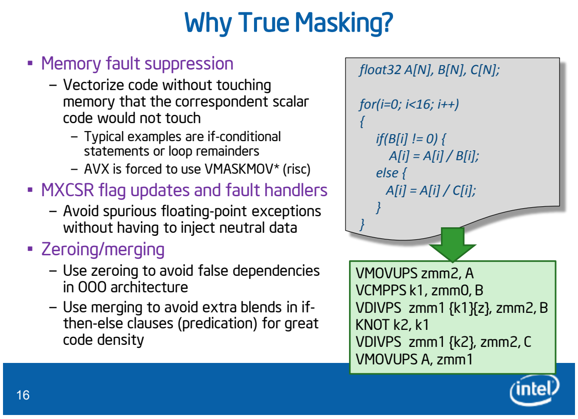


AVX-512 Features (II): Masking

- VADDPS ZMM0 {k1}, ZMM3, [mem]
  - Mask bits used to:
    1. *Suppress individual elements read from memory*
       - hence not signaling any memory fault
    2. *Avoid actual independent operations within an instruction happening*
       - and hence not signaling any FP fault
    3. *Avoid the individual destination elements being updated,*
       - or alternatively, force them to zero (zeroing)

```
for (I in vector length)
{
   if (no_masking or mask[I]) {
      dest[I] = OP(src2, src3)
   } else {
      if (zeroing_masking)
         dest[I]  = 0
      else
         // dest[I] is preserved
   }
}
```

Caveat: vector shuffles do no suppress memory fault
Exceptions as mask refers to "output" not to "input"

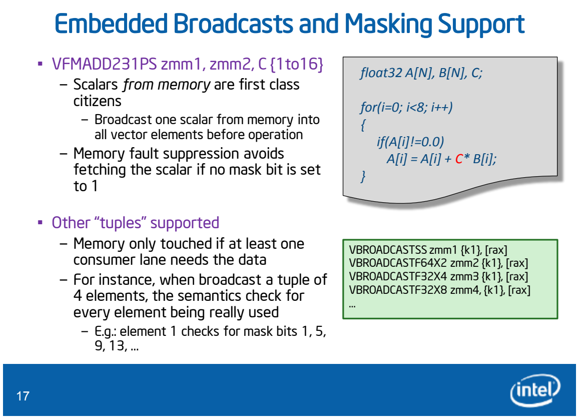


Why True Masking?

- Memory fault suppression
  - Vectorize code without touching memory that the correspondent scalar code would not touch
    - Typical examples are if-conditional statements or loop remainders
    - AVX is forced to use VMASKMOV* (risc)
- MXCSR flag updates and fault handlers
  - Avoid spurious floating-point exceptions without having to inject neutral data
- Zeroing/merging
  - Use zeroing to avoid false dependencies in OOO architecture
  - Use merging to avoid extra blends in if-then-else clauses (predication) for great code density

```
float32 A[N], B[N], C[N];

for(i=0; i<16; i++)
{
   if(B[i] != 0) {
      A[i] = A[i] / B[i];
   else {
      A[i] = A[i] / C[i];
   }
}
```

```
VMOVUPS zmm2, A
VCMPPS k1, zmm0, B
VDIVPS  zmm1 {k1}{z}, zmm2, B
KNOT k2, k1
VDIVPS  zmm1 {k2}, zmm2, C
VMOVUPS A, zmm1
```

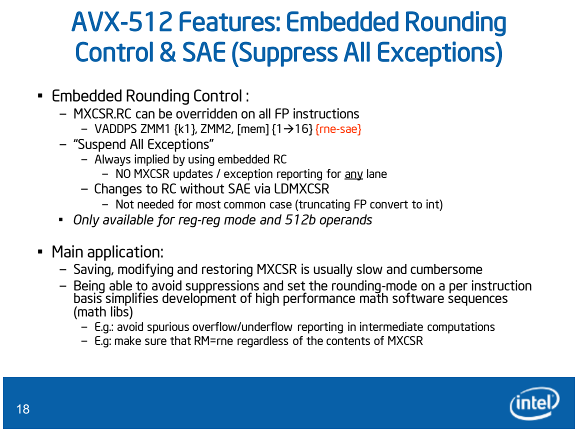


Embedded Broadcasts and Masking Support

- VFMADD231PS zmm1, zmm2, C {1to16}
  - Scalars *from memory* are first class citizens
    - Broadcast one scalar from memory into all vector elements before operation
  - Memory fault suppression avoids fetching the scalar if no mask bit is set to 1
- Other "tuples" supported
  - Memory only touched if at least one consumer lane needs the data
  - For instance, when broadcast a tuple of 4 elements, the semantics check for every element being really used
    - E.g.: element 1 checks for mask bits 1, 5, 9, 13, …

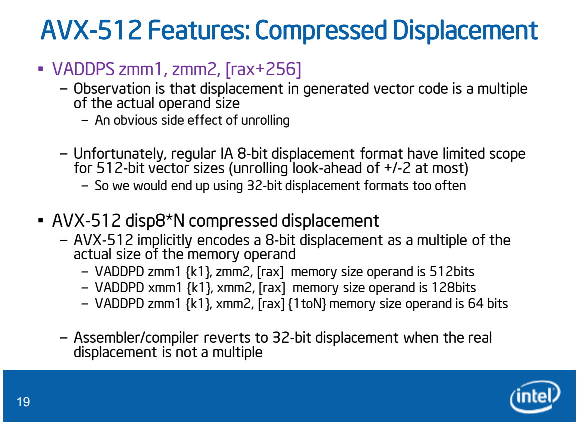
```
float32 A[N], B[N], C;

for(i=0; i<8; i++)
{
   if(A[i]!=0.0)
      A[i] = A[i] + C* B[i];
}
```

```
VBROADCASTSS zmm1 {k1}, [rax]
VBROADCASTF64X2 zmm2 {k1}, [rax]
VBROADCASTF32X4 zmm3 {k1}, [rax]
VBROADCASTF32X8 zmm4, {k1}, [rax]
…
```



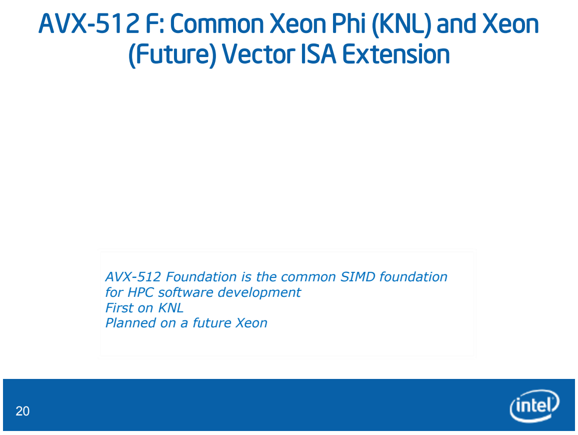
AVX-512 Features: Embedded Rounding Control & SAE (Suppress All Exceptions)

- Embedded Rounding Control :
  - MXCSR.RC can be overridden on all FP instructions
    - VADDPS ZMM1 {k1}, ZMM2, [mem] {1→16} {rne-sae}
  - "Suspend All Exceptions"
    - Always implied by using embedded RC
      - NO MXCSR updates / exception reporting for <u>any</u> lane
    - Changes to RC without SAE via LDMXCSR
      - Not needed for most common case (truncating FP convert to int)
  - *Only available for reg-reg mode and 512b operands*
- Main application:
  - Saving, modifying and restoring MXCSR is usually slow and cumbersome
  - Being able to avoid suppressions and set the rounding-mode on a per instruction basis simplifies development of high performance math software sequences (math libs)
    - E.g.: avoid spurious overflow/underflow reporting in intermediate computations
    - E.g: make sure that RM=rne regardless of the contents of MXCSR



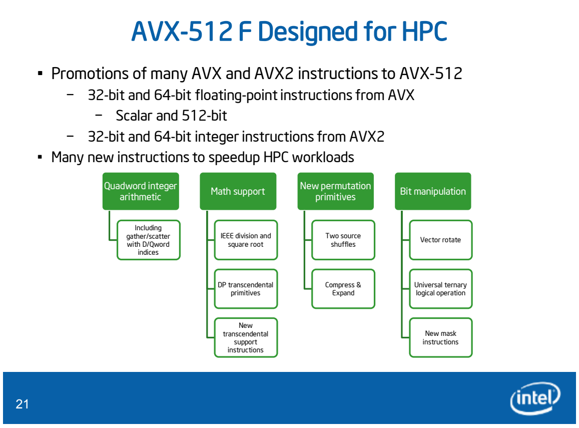
AVX-512 Features: Compressed Displacement

- VADDPS zmm1, zmm2, [rax+256]
  - Observation is that displacement in generated vector code is a multiple of the actual operand size
    - An obvious side effect of unrolling
  - Unfortunately, regular IA 8-bit displacement format have limited scope for 512-bit vector sizes (unrolling look-ahead of +/-2 at most)
    - So we would end up using 32-bit displacement formats too often
- AVX-512 disp8*N compressed displacement
  - AVX-512 implicitly encodes a 8-bit displacement as a multiple of the actual size of the memory operand
    - VADDPD zmm1 {k1}, zmm2, [rax] memory size operand is 512bits
    - VADDPD xmm1 {k1}, xmm2, [rax] memory size operand is 128bits
    - VADDPD zmm1 {k1}, xmm2, [rax] {1toN} memory size operand is 64 bits
  - Assembler/compiler reverts to 32-bit displacement when the real displacement is not a multiple



AVX-512 F: Common Xeon Phi (KNL) and Xeon (Future) Vector ISA Extension

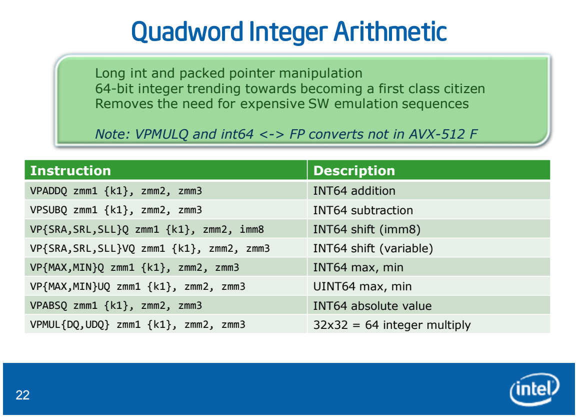
AVX-512 Foundation is the common SIMD foundation for HPC software development
First on KNL
Planned on a future Xeon



AVX-512 F Designed for HPC

- Promotions of many AVX and AVX2 instructions to AVX-512
  - 32-bit and 64-bit floating-point instructions from AVX
    - Scalar and 512-bit
  - 32-bit and 64-bit integer instructions from AVX2
- Many new instructions to speedup HPC workloads

Quadword integer arithmetic
Including gather/scatter with D/Qword indices
Math support
IEEE division and square root
DP transcendental primitives
New transcendental support instructions
New permutation primitives
Two source shuffles
Compress & Expand
Bit manipulation
Vector rotate
Universal ternary logical operation
New mask instructions

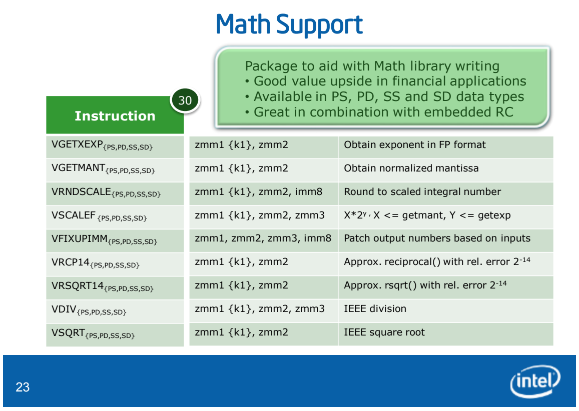


Quadword Integer Arithmetic

Long int and packed pointer manipulation
64-bit integer trending towards becoming a first class citizen
Removes the need for expensive SW emulation sequences

Note: VPMULQ and int64 <-> FP converts not in AVX-512 F

Instruction	Description
VPADDQ zmm1 {k1}, zmm2, zmm3	INT64 addition
VPSUBQ zmm1 {k1}, zmm2, zmm3	INT64 subtraction
VP{SRA,SRL,SLL}Q zmm1 {k1}, zmm2, imm8	INT64 shift (imm8)
VP{SRA,SRL,SLL}VQ zmm1 {k1}, zmm2, zmm3	INT64 shift (variable)
VP{MAX,MIN}Q zmm1 {k1}, zmm2, zmm3	INT64 max, min
VP{MAX,MIN}UQ zmm1 {k1}, zmm2, zmm3	UINT64 max, min
VPABSQ zmm1 {k1}, zmm2, zmm3	INT64 absolute value
VPMUL{DQ,UDQ} zmm1 {k1}, zmm2, zmm3	32x32 = 64 integer multiply

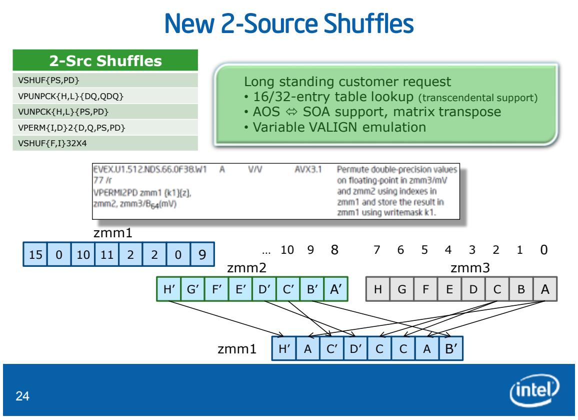


Math Support

Package to aid with Math library writing
- Good value upside in financial applications
- Available in PS, PD, SS and SD data types
- Great in combination with embedded RC

30

Instruction		
VGETXEXP$_{\{PS,PD,SS,SD\}}$	zmm1 {k1}, zmm2	Obtain exponent in FP format
VGETMANT$_{\{PS,PD,SS,SD\}}$	zmm1 {k1}, zmm2	Obtain normalized mantissa
VRNDSCALE$_{\{PS,PD,SS,SD\}}$	zmm1 {k1}, zmm2, imm8	Round to scaled integral number
VSCALEF $_{\{PS,PD,SS,SD\}}$	zmm1 {k1}, zmm2, zmm3	X*2^{y} , X <= getmant, Y <= getexp
VFIXUPIMM$_{\{PS,PD,SS,SD\}}$	zmm1, zmm2, zmm3, imm8	Patch output numbers based on inputs
VRCP14$_{\{PS,PD,SS,SD\}}$	zmm1 {k1}, zmm2	Approx. reciprocal() with rel. error 2^{-14}
VRSQRT14$_{\{PS,PD,SS,SD\}}$	zmm1 {k1}, zmm2	Approx. rsqrt() with rel. error 2^{-14}
VDIV$_{\{PS,PD,SS,SD\}}$	zmm1 {k1}, zmm2, zmm3	IEEE division
VSQRT$_{\{PS,PD,SS,SD\}}$	zmm1 {k1}, zmm2	IEEE square root

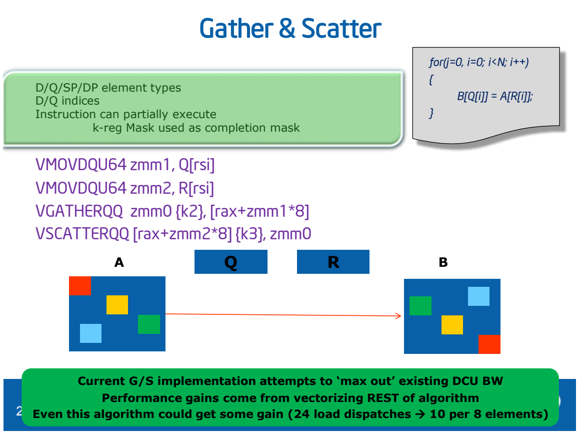


New 2-Source Shuffles

2-Src Shuffles
VSHUF{PS,PD}
VPUNPCK{H,L}{DQ,QDQ}
VUNPCK{H,L}{PS,PD}
VPERM{I,D}2{D,Q,PS,PD}
VSHUF{F,I}32X4

Long standing customer request
- 16/32-entry table lookup (transcendental support)
- AOS ⇔ SOA support, matrix transpose
- Variable VALIGN emulation

EVEX.U1.512.NDS.66.0F38.W1 77 /r VPERMI2PD zmm1 {k1}{z}, zmm2, zmm3/B$_{64}$(mV)	A	V/V	AVX3.1	Permute double-precision values on floating-point in zmm3/mV and zmm2 using indexes in zmm1 and store the result in zmm1 using writemask k1.



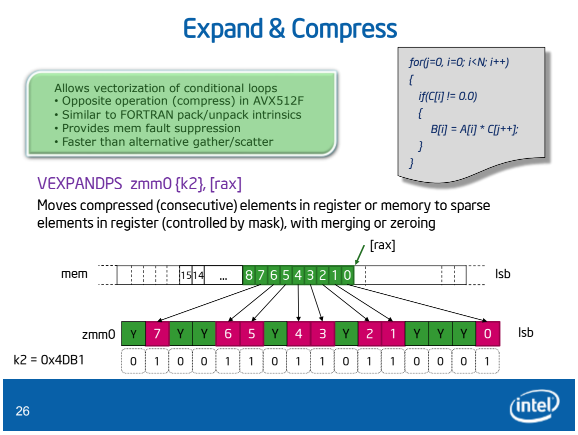
Gather & Scatter

D/Q/SP/DP element types
D/Q indices
Instruction can partially execute
k-reg Mask used as completion mask

```
for(j=0, i=0; i<N; i++)
{
      B[Q[i]] = A[R[i]];
}
```

VMOVDQU64 zmm1, Q[rsi]
VMOVDQU64 zmm2, R[rsi]
VGATHERQQ zmm0 {k2}, [rax+zmm1*8]
VSCATTERQQ [rax+zmm2*8] {k3}, zmm0

Current G/S implementation attempts to 'max out' existing DCU BW
Performance gains come from vectorizing REST of algorithm
Even this algorithm could get some gain (24 load dispatches → 10 per 8 elements)



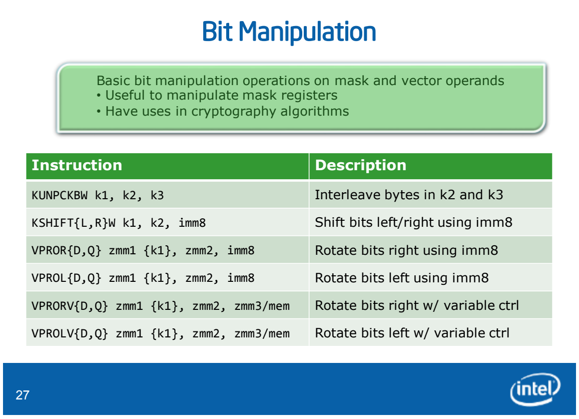
Expand & Compress

Allows vectorization of conditional loops
- Opposite operation (compress) in AVX512F
- Similar to FORTRAN pack/unpack intrinsics
- Provides mem fault suppression
- Faster than alternative gather/scatter

```
for(j=0, i=0; i<N; i++)
{
  if(C[i] != 0.0)
  {
     B[i] = A[i] * C[j++];
  }
}
```

VEXPANDPS zmm0 {k2}, [rax]

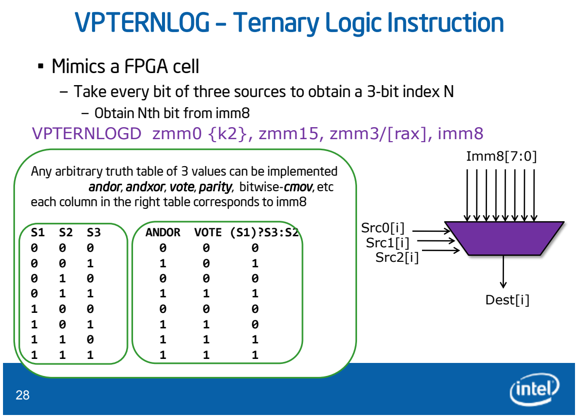
Moves compressed (consecutive) elements in register or memory to sparse elements in register (controlled by mask), with merging or zeroing



Bit Manipulation

Basic bit manipulation operations on mask and vector operands
- Useful to manipulate mask registers
- Have uses in cryptography algorithms

Instruction	Description
`KUNPCKBW k1, k2, k3`	Interleave bytes in k2 and k3
`KSHIFT{L,R}W k1, k2, imm8`	Shift bits left/right using imm8
`VPROR{D,Q} zmm1 {k1}, zmm2, imm8`	Rotate bits right using imm8
`VPROL{D,Q} zmm1 {k1}, zmm2, imm8`	Rotate bits left using imm8
`VPRORV{D,Q} zmm1 {k1}, zmm2, zmm3/mem`	Rotate bits right w/ variable ctrl
`VPROLV{D,Q} zmm1 {k1}, zmm2, zmm3/mem`	Rotate bits left w/ variable ctrl



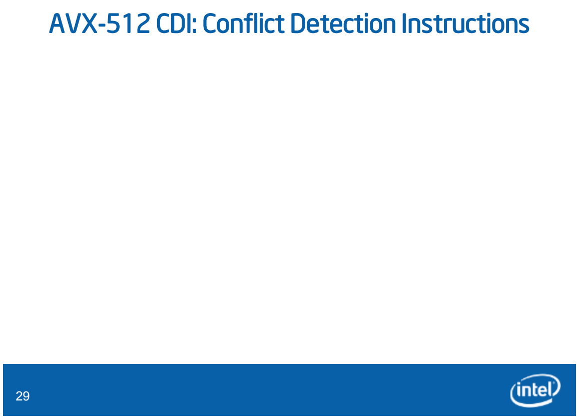
VPTERNLOG – Ternary Logic Instruction

- Mimics a FPGA cell
  - Take every bit of three sources to obtain a 3-bit index N
    - Obtain Nth bit from imm8

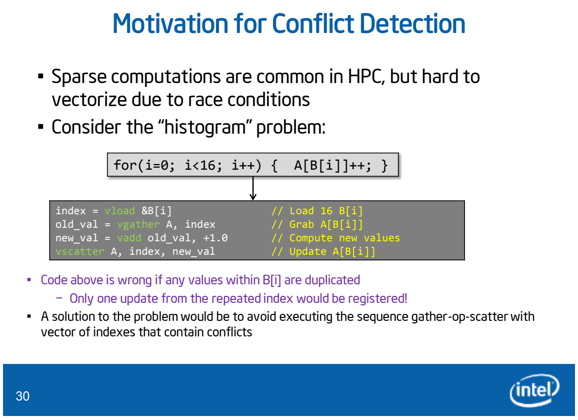
VPTERNLOGD zmm0 {k2}, zmm15, zmm3/[rax], imm8

Any arbitrary truth table of 3 values can be implemented
andor, ***andxor***, ***vote***, ***parity***, bitwise-***cmov***, etc
each column in the right table corresponds to imm8

S1	S2	S3	ANDOR	VOTE	(S1)?S3:S2
0	0	0	0	0	0
0	0	1	1	0	1
0	1	0	0	0	0
0	1	1	1	1	1
1	0	0	0	0	0
1	0	1	1	1	0
1	1	0	1	1	1
1	1	1	1	1	1



AVX-512 CDI: Conflict Detection Instructions



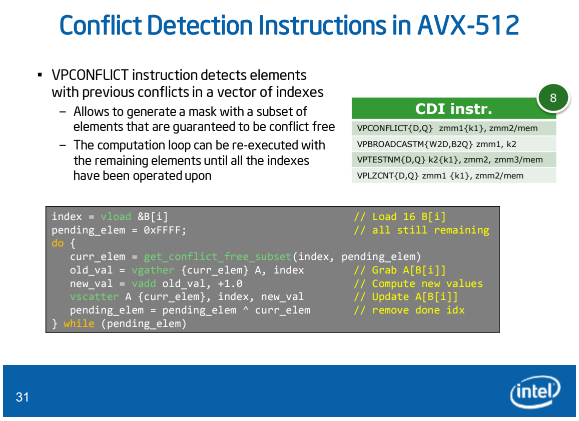
Motivation for Conflict Detection

- Sparse computations are common in HPC, but hard to vectorize due to race conditions
- Consider the "histogram" problem:

```
for(i=0; i<16; i++) {  A[B[i]]++; }
```

```
index = vload &B[i]                     // Load 16 B[i]
old_val = vgather A, index              // Grab A[B[i]]
new_val = vadd old_val, +1.0            // Compute new values
vscatter A, index, new_val              // Update A[B[i]]
```

- Code above is wrong if any values within B[i] are duplicated
  - Only one update from the repeated index would be registered!
- A solution to the problem would be to avoid executing the sequence gather-op-scatter with vector of indexes that contain conflicts

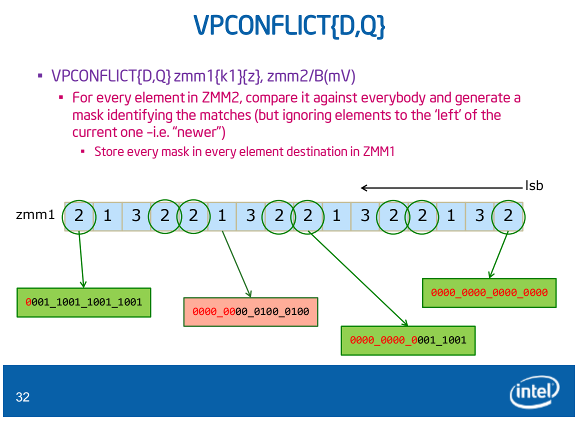


Conflict Detection Instructions in AVX-512

- VPCONFLICT instruction detects elements with previous conflicts in a vector of indexes
  - Allows to generate a mask with a subset of elements that are guaranteed to be conflict free
  - The computation loop can be re-executed with the remaining elements until all the indexes have been operated upon

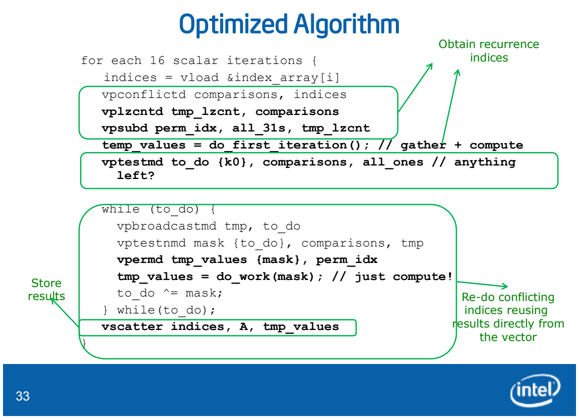
CDI instr.
VPCONFLICT{D,Q} zmm1{k1}, zmm2/mem
VPBROADCASTM{W2D,B2Q} zmm1, k2
VPTESTNM{D,Q} k2{k1}, zmm2, zmm3/mem
VPLZCNT{D,Q} zmm1 {k1}, zmm2/mem

```
index = vload &B[i]                          // Load 16 B[i]
pending_elem = 0xFFFF;                       // all still remaining
do {
   curr_elem = get_conflict_free_subset(index, pending_elem)
   old_val = vgather {curr_elem} A, index    // Grab A[B[i]]
   new_val = vadd old_val, +1.0              // Compute new values
   vscatter A {curr_elem}, index, new_val    // Update A[B[i]]
   pending_elem = pending_elem ^ curr_elem   // remove done idx
} while (pending_elem)
```



VPCONFLICT{D,Q}

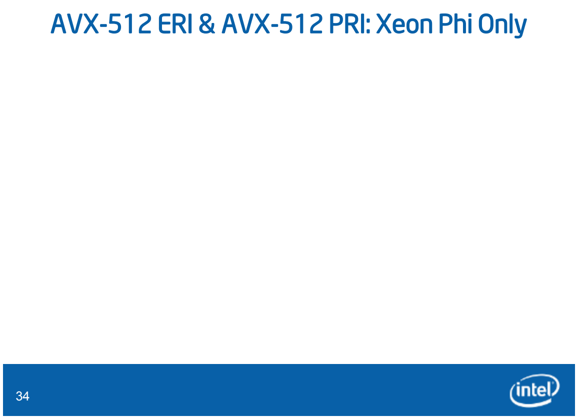
- VPCONFLICT{D,Q} zmm1{k1}{z}, zmm2/B(mV)
  - For every element in ZMM2, compare it against everybody and generate a mask identifying the matches (but ignoring elements to the 'left' of the current one –i.e. "newer")
    - Store every mask in every element destination in ZMM1



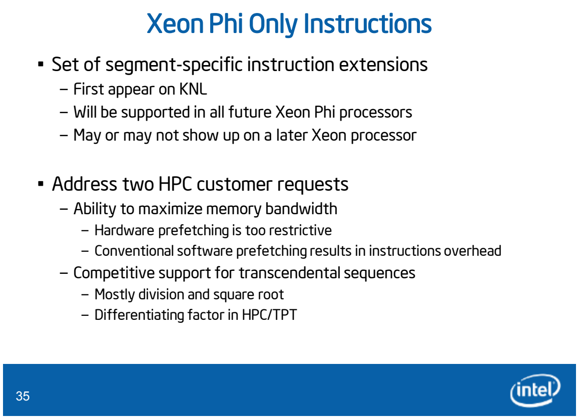
Optimized Algorithm

```
for each 16 scalar iterations {
   indices = vload &index_array[i]
   vpconflictd comparisons, indices
   vplzcntd tmp_lzcnt, comparisons
   vpsubd perm_idx, all_31s, tmp_lzcnt
   temp_values = do_first_iteration(); // gather + compute
   vptestmd to_do {k0}, comparisons, all_ones // anything
     left?

   while (to_do) {
     vpbroadcastmd tmp, to_do
     vptestnmd mask {to_do}, comparisons, tmp
     vpermd tmp_values {mask}, perm_idx
     tmp_values = do_work(mask); // just compute!
     to_do ^= mask;
   } while(to_do);
   vscatter indices, A, tmp_values
}
```

Obtain recurrence indices
Store results
Re-do conflicting indices reusing results directly from the vector

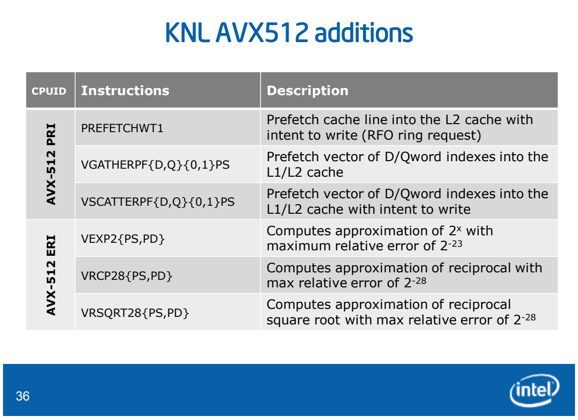


AVX-512 ERI & AVX-512 PRI: Xeon Phi Only



Xeon Phi Only Instructions

- Set of segment-specific instruction extensions
  - First appear on KNL
  - Will be supported in all future Xeon Phi processors
  - May or may not show up on a later Xeon processor
- Address two HPC customer requests
  - Ability to maximize memory bandwidth
    - Hardware prefetching is too restrictive
    - Conventional software prefetching results in instructions overhead
  - Competitive support for transcendental sequences
    - Mostly division and square root
    - Differentiating factor in HPC/TPT



KNL AVX512 additions

CPUID	Instructions	Description
AVX-512 PRI	PREFETCHWT1	Prefetch cache line into the L2 cache with intent to write (RFO ring request)
	VGATHERPF{D,Q}{0,1}PS	Prefetch vector of D/Qword indexes into the L1/L2 cache
	VSCATTERPF{D,Q}{0,1}PS	Prefetch vector of D/Qword indexes into the L1/L2 cache with intent to write
AVX-512 ERI	VEXP2{PS,PD}	Computes approximation of 2^x with maximum relative error of 2^{-23}
	VRCP28{PS,PD}	Computes approximation of reciprocal with max relative error of 2^{-28}
	VRSQRT28{PS,PD}	Computes approximation of reciprocal square root with max relative error of 2^{-28}

KNL AVX512 additions

CPUID	Instructions	Motivation
AVX-512 PRI	PREFETCHWT1	Reduce ring traffic in core-to-core data communication
	VGATHERPF{D,Q}{0,1}PS	Reduce overhead of software prefetching: *dedicate side engine to prefetch sparse structures while devoting the main CPU to pure raw flops*
	VSCATTERPF{D,Q}{0,1}PS	
AVX-512 ERI	VEXP2{PS,PD}	Speed-up key FSI workloads: Black-Scholes, Montecarlo
	VRCP28{PS,PD}	Key building block to speed up most transcendental sequences (in particular, division and square root): Increasing precision from 14=>28 allows to reduce one complete Newton-Raphson iteration
	VRSQRT28{PS,PD}	

Summary of AVX512 on KNL

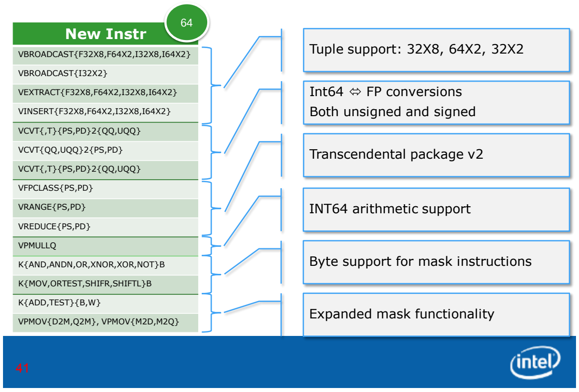
- AVX-512 F: new 512-bit vector ISA extension
  - Common between Xeon (SKL) and Xeon Phi (KNL)
- AVX-512 CDI Conflict detection instructions
  - Improves autovectorization
  - On Xeon Phi first
- AVX-512 ERI & PRI
  - 28-bit transcendentals and new prefetch instructions
  - On Xeon Phi only

Xeon (SKX) additions to AVX512F

AVX512DQ

- Complete Qword support
  - VPMULLQ packed 64x64 ➔ 64
  - Packed/Scalar converts of signed/unsigned to SP/DP
  - Arithmatic shift right
  - Etc
- Extend mask architecture to word and byte
  - Byte masks are natural for packed Qword operands
- Minor additions to transcendental support
- Convert AVX512 mask ←→ 'SSE/AVX' mask
- 'aggregate datatype' support
  - Broadcast/insert/extract complex singles etc

AVX512DQ : additional HPC focus



New Instr 64

New Instr
VBROADCAST{F32X8,F64X2,I32X8,I64X2}
VBROADCAST{I32X2}
VEXTRACT{F32X8,F64X2,I32X8,I64X2}
VINSERT{F32X8,F64X2,I32X8,I64X2}
VCVT{,T}{PS,PD}2{QQ,UQQ}
VCVT{QQ,UQQ}2{PS,PD}
VCVT{,T}{PS,PD}2{QQ,UQQ}
VFPCLASS{PS,PD}
VRANGE{PS,PD}
VREDUCE{PS,PD}
VPMULLQ
K{AND,ANDN,OR,XNOR,XOR,NOT}B
K{MOV,ORTEST,SHIFR,SHIFTL}B
K{ADD,TEST}{B,W}
VPMOV{D2M,Q2M}, VPMOV{M2D,M2Q}

Tuple support: 32X8, 64X2, 32X2

Int64 ⇔ FP conversions
Both unsigned and signed

Transcendental package v2

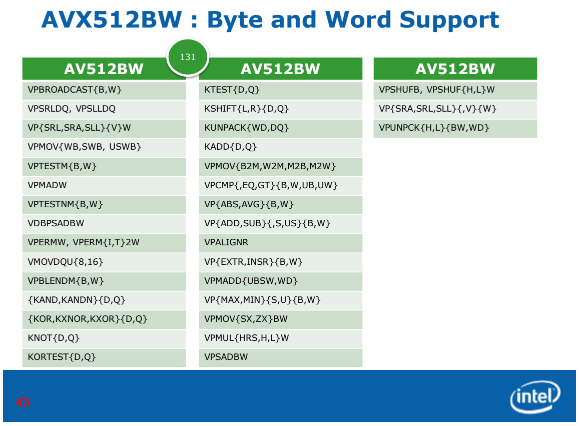
INT64 arithmetic support

Byte support for mask instructions

Expanded mask functionality

AVX512BW

- Full support for Byte/Word operations
  - MMX/SSE2/AVX2 re-promoted to AVX512 semantics
- Mask operations extended to 32/64 bits
  - 32-bit mask refers to AVX512 'short' operands
  - 64-bit mask refers to AVX512 byte operands
- Loads/Stores/Broadcastsfor AVX512 semantics
- Permute architecture extended to words
  - Vpermw, vpermi2w, vpermt2w
- New PSAD instruction,etc

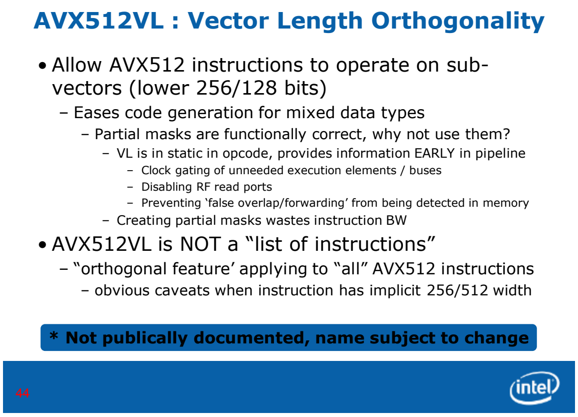


AVX512BW : Byte and Word Support

131

AV512BW	AV512BW	AV512BW
VPBROADCAST{B,W}	KTEST{D,Q}	VPSHUFB, VPSHUF{H,L}W
VPSRLDQ, VPSLLDQ	KSHIFT{L,R}{D,Q}	VP{SRA,SRL,SLL}{,V}{W}
VP{SRL,SRA,SLL}{V}W	KUNPACK{WD,DQ}	VPUNPCK{H,L}{BW,WD}
VPMOV{WB,SWB, USWB}	KADD{D,Q}	
VPTESTM{B,W}	VPMOV{B2M,W2M,M2B,M2W}	
VPMADW	VPCMP{,EQ,GT}{B,W,UB,UW}	
VPTESTNM{B,W}	VP{ABS,AVG}{B,W}	
VDBPSADBW	VP{ADD,SUB}{,S,US}{B,W}	
VPERMW, VPERM{I,T}2W	VPALIGNR	
VMOVDQU{8,16}	VP{EXTR,INSR}{B,W}	
VPBLENDM{B,W}	VPMADD{UBSW,WD}	
{KAND,KANDN}{D,Q}	VP{MAX,MIN}{S,U}{B,W}	
{KOR,KXNOR,KXOR}{D,Q}	VPMOV{SX,ZX}BW	
KNOT{D,Q}	VPMUL{HRS,H,L}W	
KORTEST{D,Q}	VPSADBW	

43

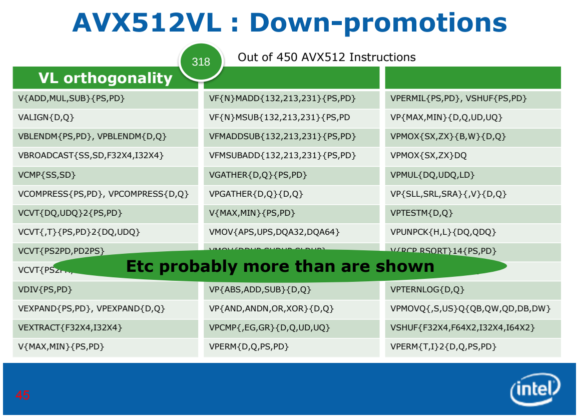


AVX512VL : Vector Length Orthogonality

- Allow AVX512 instructions to operate on sub-vectors (lower 256/128 bits)
  - Eases code generation for mixed data types
    - Partial masks are functionally correct, why not use them?
      - VL is in static in opcode, provides information EARLY in pipeline
        - Clock gating of unneeded execution elements / buses
        - Disabling RF read ports
        - Preventing 'false overlap/forwarding' from being detected in memory
      - Creating partial masks wastes instruction BW
- AVX512VL is NOT a "list of instructions"
  - "orthogonal feature' applying to "all" AVX512 instructions
    - obvious caveats when instruction has implicit 256/512 width

*** Not publically documented, name subject to change**

44



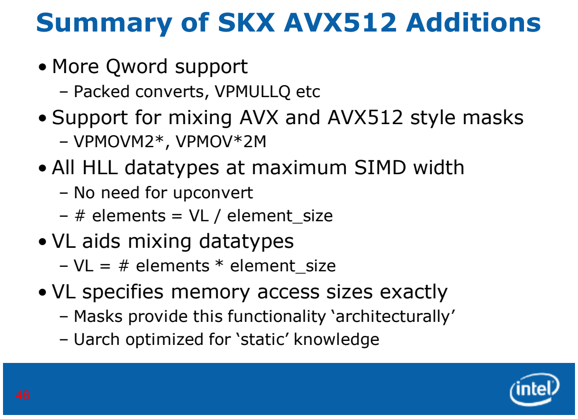
AVX512VL : Down-promotions

318 Out of 450 AVX512 Instructions

VL orthogonality		
V{ADD,MUL,SUB}{PS,PD}	VF{N}MADD{132,213,231}{PS,PD}	VPERMIL{PS,PD}, VSHUF{PS,PD}
VALIGN{D,Q}	VF{N}MSUB{132,213,231}{PS,PD	VP{MAX,MIN}{D,Q,UD,UQ}
VBLENDM{PS,PD}, VPBLENDM{D,Q}	VFMADDSUB{132,213,231}{PS,PD}	VPMOX{SX,ZX}{B,W}{D,Q}
VBROADCAST{SS,SD,F32X4,I32X4}	VFMSUBADD{132,213,231}{PS,PD}	VPMOX{SX,ZX}DQ
VCMP{SS,SD}	VGATHER{D,Q}{PS,PD}	VPMUL{DQ,UDQ,LD}
VCOMPRESS{PS,PD}, VPCOMPRESS{D,Q}	VPGATHER{D,Q}{D,Q}	VP{SLL,SRL,SRA}{,V}{D,Q}
VCVT{DQ,UDQ}2{PS,PD}	V{MAX,MIN}{PS,PD}	VPTESTM{D,Q}
VCVT{,T}{PS,PD}2{DQ,UDQ}	VMOV{APS,UPS,DQA32,DQA64}	VPUNPCK{H,L}{DQ,QDQ}
VCVT{PS2PD,PD2PS}	VMOV{DDUP,SHDUP,SLDUP}	V{RCP,RSQRT}14{PS,PD}
VCVT{PS2[illegible]	[illegible]	[illegible]
VDIV{PS,PD}	VP{ABS,ADD,SUB}{D,Q}	VPTERNLOG{D,Q}
VEXPAND{PS,PD}, VPEXPAND{D,Q}	VP{AND,ANDN,OR,XOR}{D,Q}	VPMOVQ{,S,US}Q{QB,QW,QD,DB,DW}
VEXTRACT{F32X4,I32X4}	VPCMP{,EG,GR}{D,Q,UD,UQ}	VSHUF{F32X4,F64X2,I32X4,I64X2}
V{MAX,MIN}{PS,PD}	VPERM{D,Q,PS,PD}	VPERM{T,I}2{D,Q,PS,PD}

Etc probably more than are shown

45

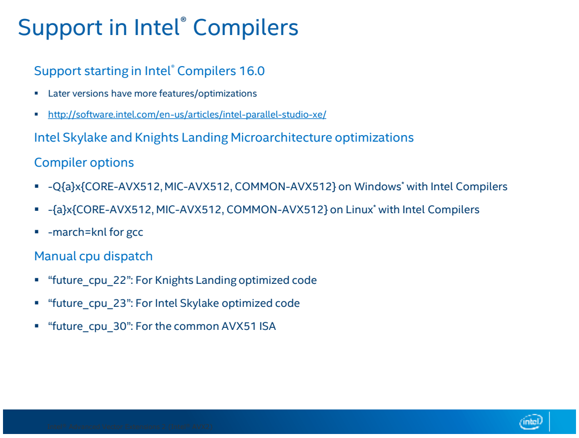


Summary of SKX AVX512 Additions

- More Qword support
  - Packed converts, VPMULLQ etc
- Support for mixing AVX and AVX512 style masks
  - VPMOVM2*, VPMOV*2M
- All HLL datatypes at maximum SIMD width
  - No need for upconvert
  - # elements = VL / element_size
- VL aids mixing datatypes
  - VL = # elements * element_size
- VL specifies memory access sizes exactly
  - Masks provide this functionality 'architecturally'
  - Uarch optimized for 'static' knowledge

46

Getting Started with AVX512 – Tools and Optimization Methodology



Support in Intel® Compilers

Support starting in Intel® Compilers 16.0

- Later versions have more features/optimizations
- http://software.intel.com/en-us/articles/intel-parallel-studio-xe/

Intel Skylake and Knights Landing Microarchitecture optimizations

Compiler options

- -Q{a}x{CORE-AVX512, MIC-AVX512, COMMON-AVX512} on Windows* with Intel Compilers
- -{a}x{CORE-AVX512, MIC-AVX512, COMMON-AVX512} on Linux* with Intel Compilers
- -march=knl for gcc

Manual cpu dispatch

- "future_cpu_22": For Knights Landing optimized code
- "future_cpu_23": For Intel Skylake optimized code
- "future_cpu_30": For the common AVX51 ISA

Intel® Advanced Vector Extensions 2 (Intel® AVX2)

Support in GCC and YASM Compilers

Support starting in NASM 2.11.08, binutils 2.25

Support starting in GCC 5.0:

- Compiler options: -mavx512f, -mavx512cd, -march=knl
- To switch on all KNL NI: -march=knl
- All AVX-512 NI are supported through intrinsics and inline assembly
- Autovectorization/autogeneration utilize some of AVX-512 NI

Other Intel Tools

Intel® VTune™ Amplifier 2016 Update 1

- http://software.intel.com/en-us/intel-vtune-amplifier-xe/

Intel® Math Kernel Library (Intel® MKL) 11.2.1

- http://software.intel.com/en-us/articles/intel-mkl/

Intel® Integrated Performance Primitives (Intel® IPP) 9.0

- http://software.intel.com/en-us/articles/intel-ipp/

Intel® Software Development Emulator 7.21

- http://www.intel.com/software/sde

Comprehensive support for Intel Skylake and Knights Landing Microarchitecture and Intel® AVX-512 across a broad set of software development tools

Optimization Tip: Compiler Optimization Report

Control the level of detail in the report generated:

- /Qopt-report[0|1|2|3] (Windows*)
- -opt-report[0|1|2|3] (Linux*, MacOS* X)

Select the places of interests:

- /Qopt-report-phase[:phase] (Windows*)
- -opt-report-phase[=phase] (Linux*, Mac OS* X)
  - `ipo_inl` - Interprocedural Optimization Inlining Report
  - `ilo` – Intermediate Language Scalar Optimization
  - `hpo` – High Performance Optimization
  - `hlo` – High-level Optimization
  - `all` – All optimizations (not recommended, output too verbose).

Save report output to file:

- /Qopt-report-file:[file] (Windows*)
- -opt-report-file=[file] (Linux*, MacOS* X)

```
% icc -O3 -opt-report-phase=hlo -opt-report-phase=hpo
```

```
…
LOOP INTERCHANGE in loops at line: 7 8 9
Loopnest permutation ( 1 2 3 ) --> ( 2 3 1 )
…
Loop at line 8 blocked by 128
Loop at line 9 blocked by 128
Loop at line 10 blocked by 128
…
Loop at line 10 unrolled and jammed by 4
Loop at line 8 unrolled and jammed by 4
…
…(10)…   loop was not vectorized: not inner loop.
…(8)…    loop was not vectorized: not inner loop.
…(9)…    PERMUTED LOOP WAS VECTORIZED
…
```

Optimization Tip: Floating Point Operations

The Floating Point (FP) Model: -fp-model (/fp:)

- Choose the floating point semantics at coarse level
- Specify the compiler rules for value safety, expression valuation
- -fp-model
  - `fast [=1]` Allows value unsafe optimization
  - `fast =2` Allows additional approximation
  - `precise` Value safe optimization only
  - `source|double|extended` imply "precise" unless overridden Intermediate result in expression evaluation
  - `strict` precise + except + disable fma + don't assume default floating-point environment, output too verbose).

Denormalized Number and Flush-to-Zero

- Extends the lower range of IEEE Floating Point numbers
- Skylake: moderate performance penalty; KNL: higher cost
- Use FTZ, DAZ if you create but don't handle denormals
  - Works for SSE/AVX/AVX2/AVX512; not available on X87

Optimization Tip: Floating Point Approximations

Reciprocal and Reciprocal of square root

- On KNL, RCP28PS, RSQRT28PS, RCP28PD, RSQRTPD give 24-bit SP, 28-bit DP
- On Skylake and KNL, vrcp14ps, vrsqrt14ps, vrcp14pd. Vrsqrt14pd, Gives 14-bit for SP **and** DP
  - Haswell has only 11-bit SP, vrcpps, vrsqrtps. It's a big improvement
  - Compile generates 14-bit versions, but 11-bit version in single precision still exist via intrinsics
- -no-prec-div and –no-prec-sqrt override the –fp-model settings
  - Full divide is expensive and it is not pipelined x/y ⬌ x*(1.0/y)

Base 2 Exponential Function

- KNL implements vexp2ps vexp2pd. Both give 23-bit accuracy
- On Skylake, call Intel vectorized libm, svml_exp2ps_ep()

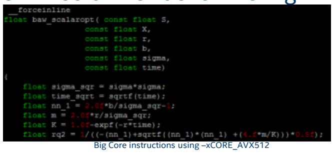
For Exponential Function of other bases, convert to Base 2 (not *e* not 10)

- Base 2 always has performance advantage, because of table lookup implementation.
- For other bases, change the base to 2, using change of base formula.

Optimization Tip: AVX512 vector Instruction Listing

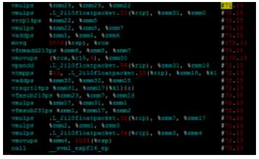
Curious about AVX512 Instructions

- Use –S assembly listing options
- Name a listing file using -o *filename*



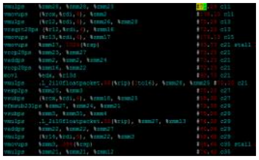
```
 __forceinline
float baw_scalaropt( const float S,
                     const float X,
                     const float r,
                     const float b,
                     const float sigma,
                     const float time)
{
    float sigma_sqr = sigma*sigma;
    float time_sqrt = sqrtf(time);
    float nn_1 = 2.0f*b/sigma_sqr-1;
    float m = 2.0f*r/sigma_sqr;
    float K = 1.0f-expf(-r*time);
    float rq2 = 1/((-(nn_1)+sqrtf((nn_1)*(nn_1) +(4.f*m/K)))*0.5f);
```

Big Core instructions using –xCORE_AVX512



```
vmulps      %zmm29, %zmm29, %zmm22                              #71.[illegible]
vmulps      .L_2il0floatpacket.[illegible](%rip), %zmm31, %zmm0 #[illegible]
vrcp14ps    %zmm22, %zmm3                                       #[illegible]
vmulps      %zmm22, %zmm5, %zmm7                                #[illegible]
vaddps      %zmm5, %zmm4, %zmm6                                 #[illegible]
movq        1024(%rsp), %rcx                                    #[illegible]
vfnmadd213ps %zmm6, %zmm5, %zmm7                                #[illegible]
vmovups     (%rcx,%r15,4), %zmm30                               #[illegible]
vpandd      .L_2il0floatpacket.[illegible](%rip), %zmm31, %zmm19 #[illegible]
vcmpps      $[illegible], .L_2il0floatpacket.[illegible](%rip), %zmm18, %k1 #[illegible]
vaddps      %zmm30, %zmm30, %zmm18                              #[illegible]
vrsqrt14ps  %zmm31, %zmm17{%k1}{z}                              #[illegible]
vfmsub213ps %zmm23, %zmm7, %zmm18                               #[illegible]
vmulps      %zmm17, %zmm31, %zmm1                               #[illegible]
vfmsub231ps %zmm1, %zmm17, %zmm2                                #[illegible]
vmulps      .L_2il0floatpacket.[illegible](%rip), %zmm7, %zmm17 #[illegible]
vmulps      %zmm2, %zmm1, %zmm5                                 #[illegible]
vmulps      .L_2il0floatpacket.[illegible](%rip), %zmm3, %zmm4  #[illegible]
vmovups     %zmm4, 1024(%rsp)                                   #[illegible]
call        __svml_expf16_ep                                    #[illegible]
```

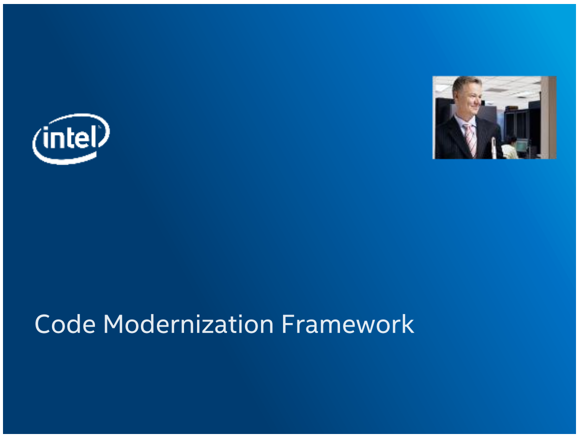
Big Core instructions using –xCORE_AVX512



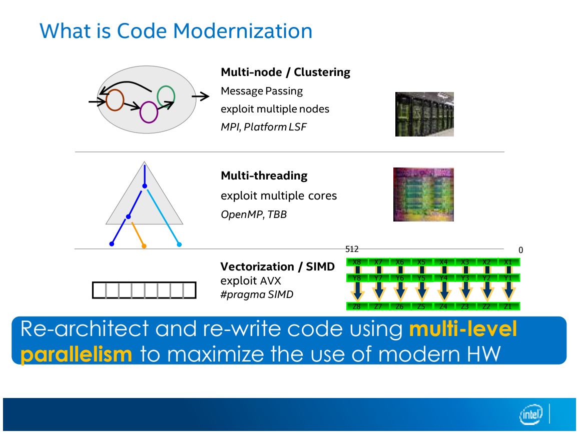
```
vmulps      %zmm28, %zmm28, %zmm23                       #72.[illegible] c11
vmovups     (%rcx,%rdi,4), %zmm2                         #[illegible] c11
vmulps      (%r12,%rdi,4), %zmm26, %zmm29                #[illegible] c13
vrsqrt28ps  (%r12,%rdi,4), %zmm16                        #[illegible] c13
vmovups     (%r13,%rdi,4), %zmm17                        #[illegible] c15
vmovups     %zmm17, 1024(%rsp)                           #[illegible] c21 stall
vrcp28ps    %zmm23, %zmm27                               #[illegible] c21
vaddps      %zmm2, %zmm2, %zmm24                         #[illegible] c21
vrcp28ps    %zmm16, %zmm22                               #[illegible] c21
movl        %edx, %r13d                                  #[illegible] c21
vmulps      .L_2il0floatpacket.[illegible](%rip){1to16}, %zmm28, %zmm25 #[illegible] c21
vexp2ps     %zmm25, %zmm3                                #[illegible] c27
vsubps      (%rcx,%rdi,4), %zmm18, %zmm23                #[illegible] c27
vfmsub231ps %zmm27, %zmm24, %zmm21                       #[illegible] c28
vsubps      %zmm3, %zmm21, %zmm4                         #[illegible] c29
vmulps      .L_2il0floatpacket.[illegible](%rip), %zmm27, %zmm13 #[illegible] c29
vaddps      %zmm22, %zmm22, %zmm27                       #[illegible] c29
vmulps      (%r14,%rdi,4), %zmm22, %zmm3                 #[illegible] c29
vmovups     %zmm3, [illegible](%rsp)                     #[illegible] c35 stall
vmulps      %zmm21, %zmm21, %zmm12                       #[illegible] c35
```

Curious about the AVX512 Instruction Compiler Generated?

- Compiler knows sqrt() is used in denominator and then use rsqrt(x) instead of sqrt(x)
- Big core calls runtime library __svml_expf16_ep, KNL use vexp2ps instruction



Code Modernization Framework



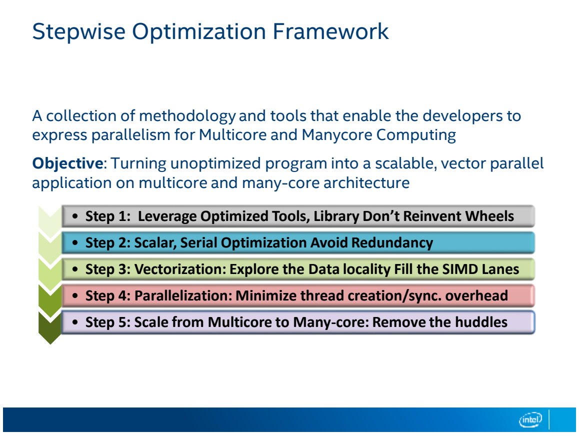
What is Code Modernization

Multi-node / Clustering
Message Passing
exploit multiple nodes
MPI, Platform LSF

Multi-threading
exploit multiple cores
OpenMP, TBB

Vectorization / SIMD
exploit AVX
#pragma SIMD

Re-architect and re-write code using **multi-level parallelism** to maximize the use of modern HW

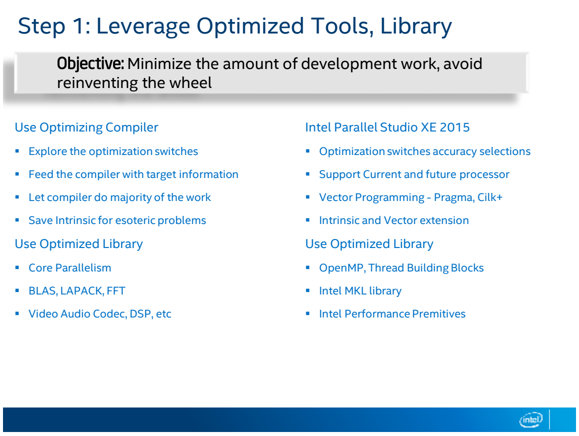


Stepwise Optimization Framework

A collection of methodology and tools that enable the developers to express parallelism for Multicore and Manycore Computing

Objective: Turning unoptimized program into a scalable, vector parallel application on multicore and many-core architecture

- **Step 1: Leverage Optimized Tools, Library Don't Reinvent Wheels**
- **Step 2: Scalar, Serial Optimization Avoid Redundancy**
- **Step 3: Vectorization: Explore the Data locality Fill the SIMD Lanes**
- **Step 4: Parallelization: Minimize thread creation/sync. overhead**
- **Step 5: Scale from Multicore to Many-core: Remove the huddles**



Step 1: Leverage Optimized Tools, Library

Objective: Minimize the amount of development work, avoid reinventing the wheel

Use Optimizing Compiler

- Explore the optimization switches
- Feed the compiler with target information
- Let compiler do majority of the work
- Save Intrinsic for esoteric problems

Use Optimized Library

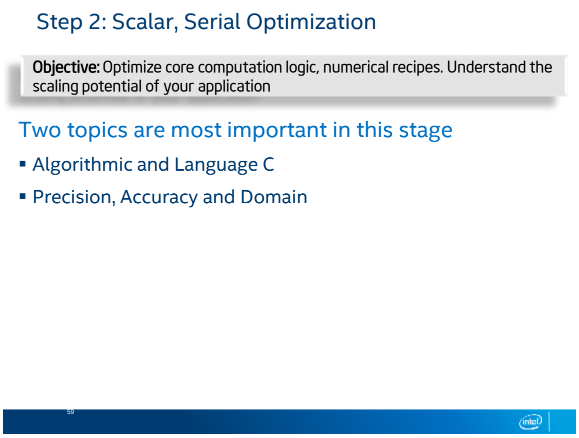
- Core Parallelism
- BLAS, LAPACK, FFT
- Video Audio Codec, DSP, etc

Intel Parallel Studio XE 2015

- Optimization switches accuracy selections
- Support Current and future processor
- Vector Programming - Pragma, Cilk+
- Intrinsic and Vector extension

Use Optimized Library

- OpenMP, Thread Building Blocks
- Intel MKL library
- Intel Performance Premitives

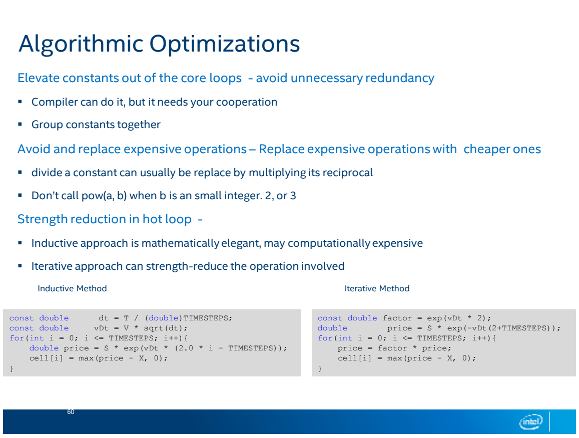


Step 2: Scalar, Serial Optimization

Objective: Optimize core computation logic, numerical recipes. Understand the scaling potential of your application

Two topics are most important in this stage

- Algorithmic and Language C
- Precision, Accuracy and Domain



Algorithmic Optimizations

Elevate constants out of the core loops - avoid unnecessary redundancy

- Compiler can do it, but it needs your cooperation
- Group constants together

Avoid and replace expensive operations – Replace expensive operations with cheaper ones

- divide a constant can usually be replace by multiplying its reciprocal
- Don't call pow(a, b) when b is an small integer. 2, or 3

Strength reduction in hot loop -

- Inductive approach is mathematically elegant, may computationally expensive
- Iterative approach can strength-reduce the operation involved

Inductive Method

```
const double      dt = T / (double)TIMESTEPS;
const double     vDt = V * sqrt(dt);
for(int i = 0; i <= TIMESTEPS; i++){
    double price = S * exp(vDt * (2.0 * i - TIMESTEPS));
    cell[i] = max(price - X, 0);
}
```

Iterative Method

```
const double factor = exp(vDt * 2);
double        price = S * exp(-vDt(2+TIMESTEPS));
for(int i = 0; i <= TIMESTEPS; i++){
    price = factor * price;
    cell[i] = max(price - X, 0);
}
```

Understand C/C++ Type Conversion Rule

C/C++ implicit type conversion rule

- `double` is higher in the type hierarchy than `float` in C/C++
- A variables promotes to double if it operates with another double.
- `0.5*V*V` will trigger a implicit conversion if V is a float
- double is at least 2X slower than float
- Type convert is very expensive. It is 6 cycles inside VPU engine

Avoid using floating point literals, Always type your constants

- Use `const float HALF = 0.5f;`

Choose the right runtime functions API calls

- `sqrt(), exp(), log()` requires double parameter
- `sqrtf(), expf(), logf()` takes float parameter

Floating Point Hardware Resources

Use Vector Processing Unit for floating point arithmetic operations

- X87 is a legacy unit that also executes FP Instructions
- Compiler –fp-model strict select x87 for FP operations
- VPU is preferred place because of SIMD parallelism and performance

Use X87 on restricted cases

- Reproduce the same results 15 years ago, right or wrong
- Generate FP exceptions for debugging purpose

FP Accuracy Mode and Domain Exclusion

Accuracy affects the performance of your program

- Choose the accuracy your problem requires
- Higher accuracy has higher cost

Set accuracy for libraries

- Intel MKL Accuracy Mode HA, LA, EP: API calls
  `vmlSetMode(VML_EP);`

Set accuracy for compiler generated expressions

- Intel® Compiler: Compiler switches
  `-fimf_precision=low,high,medium`
  `-fimf_accuracy_bits=11`

Understand the Domain of Your Problem

Not all application operate differently for certain class of FP inputs, yet still pay the price of detecting those value classes.

Exclude those FP value class can speed up calculations

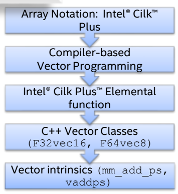
- Use `-fimf-domain-exclusion=<n1>`
- <n1> exclusive or of bit masks
- 15: common exclusions
- 31: avoid all corner case
- Exclude zero, infinities, nan and Extreme value for log, logf, /, sin
  `-fimf-domain-exclusion=23:log,logf,/,sin`

Value to exclude	Zero	Denormals	infinities	Nans	Extreme value	none
mask	16	8	4	2	1	0

Step 3 Vector Programming

Objective: Explore the Data locality, Fill up the Vector Lanes

- SIMD Parallelism Require data alignment
  - Convert the input from AOS to SOA
  - Memory declaration
    `__attribute__((aligned(64)) float a;`
  - Memory allocation `_mm_malloc(size, align)`
  - TBB: `scalable_aligned_malloc(size, align)`
- Branch Breaks SIMD Execution
  - Conditional logic has to be masked at a cost
  - Functional calls can be hazardous
- Start Vector Programming with Compiler directives
  - Provide hints on Alignment, Aliases, Data Dependency
- Use Intel® Advisor XE 2016



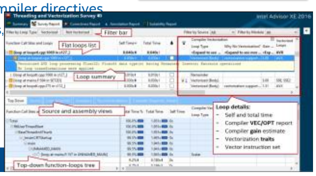
Array Notation: Intel® Cilk™ Plus

Compiler-based Vector Programming

Intel® Cilk Plus™ Elemental function

C++ Vector Classes `(F32vec16, F64vec8)`

Vector intrinsics `(mm_add_ps, vaddps)`



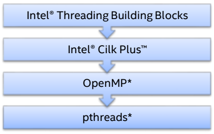
Threading and Vectorization Survey
Intel Advisor XE 2016
Summary
Survey Report
Filter bar
Flat loops list
Loop summary
Source and assembly views
Top-down function-loops tree

Loop details:
- Self and total time
- Compiler VEC/OPT report
- Compiler gain estimate
- Vectorization traits
- Vector instruction set

Step 4: Parallelization

Objective: Keep all the cores and threads busy, asynchronously

- Partition the work at high level
- Target Coarse granularity
- Manage thread creation overhead
- Minimize thread Synchronization
- Affinitize worker thread to processor threads
- Use Intel® Advisor XE 2016

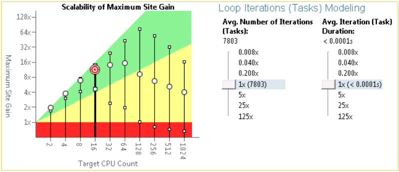


Intel® Threading Building Blocks

Intel® Cilk Plus™

OpenMP*

pthreads*



Avg. Number of Iterations (Tasks): 7803
0.008x
0.040x
0.200x
1x (7803)
5x
25x
125x

Avg. Iteration (Task) Duration: < 0.0001s
0.008x
0.040x
0.200x
1x (< 0.0001s)
5x
25x
125x

Step 5: Scale from Multicore to Manycore

Objective: Scale the program to hundreds of threads, explore heterogeneity

Reduce the memory footprint to bare minimum

- Use registers and Caches wisely
- Reduce function call overhead - Inlining
- Recalculate vs Going to memory gain

Improve Data Affinity

- Memory allocation from the worker threads

Block the data for the size of cache/thread

- Improve Memory access efficiency
- Avoid cache thrashing

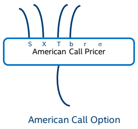
Case Study American Call Option Approximation

Algorithm Description

- Common analytical approach to option pricing
- Pricing American Call Option Using Approximation
- *Efficient Analytic Approximation of American Option Values*
  Journal of Finance June 1989 by Giovanni Barone-Adesi, Robert E. Whaley

Description:

- Start with Black-Scholes PDE, Decomposes the American call into the European call + the early exercise premium *C(S,T) = c(S,T) +* $\varepsilon_c(S, T)$
- Fine the solution to the Non-linear equation using Newton-Raphson iteration $S_{i+1} = S_i + \frac{g()}{g'()}$



Original C/C++ Implementation:

- http://finance.bi.no/~bernt/gcc_prog/algoritms_v1/algoritms/node24.html

Data generation

- S, X,T, b, are stream variables, σ, r are scalar constant
- Use C/C++ runtime random generator rand_r
- Measure the Performance measurement of the pricer only.

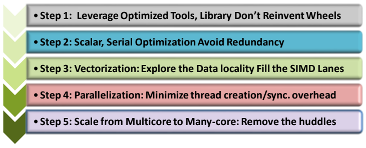
Code Modernization – Methodology & Process

Objectives

- Tracking various optimization methodology
- Identify the best known methods of achieving various step's objectives

Optimization Methodology

√ Scalar, Serial optimization
√ Vectorization
√ Parallelization


- Step 1: Leverage Optimized Tools, Library Don't Reinvent Wheels
- Step 2: Scalar, Serial Optimization Avoid Redundancy
- Step 3: Vectorization: Explore the Data locality Fill the SIMD Lanes
- Step 4: Parallelization: Minimize thread creation/sync. overhead
- Step 5: Scale from Multicore to Many-core: Remove the huddles

Code Modernization:
Step 2: Scalar, Serial Optimization

Data type and algorithm Changes

- From double/float to float

Changes to avoid C/C++ auto conversion

- Constants are explicitly typed.
- Function call are also typed `expf()` instead of `exp()`

Minimize function calls & function call overhead

- pow((nn-1), 2.0)
- cnd()-> erf(),
- All function are inlined (`__forcedinline`)

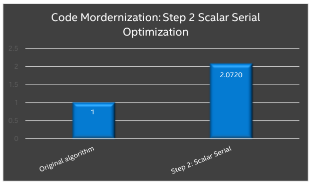
Calculation of sub-expressions

- Calculate 1/q2, 1/q2_inf

Compiler Invocation line

- `-xMIC_AVX512 -O3 -ipo -qopt-report=5 -fimf-precision=low -fimf-domain-exclusion=31 -no-prec-div -no-prec-sqrt`

Performance Data was generated on Intel Haswell EP running at 14-core, 28-thread in dual socket



Code Modernization
Step 3: Vector Programming

Starting point

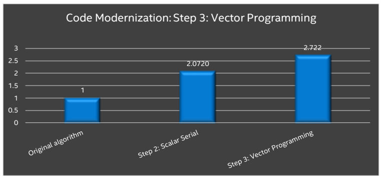
- From Step 2: Scalar, Serial optimized program

Vector Programming using Pragma SIMD

- Add annotation to the scalar optimized routine
- Give scalar syntax a vector semantic
- Compiler generate the vector code

Performance Comparison

- #pragma SIMD leverage Scalar C/C++ syntax with higher productivity
- 2.72X better than Scalar Serial Optimized version



Code Modernization
Step 4: Parallelization

Objective

- Create a user-level thread for each processor thread

Strategy

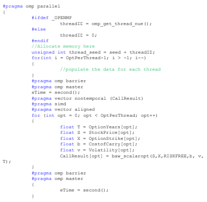
- Create threads early, Use OpenMP 4.0
- Allocate memory populate data from the worker thread
- Divide the workload into equal amount for each thread
- Each thread works on its own previously vectorized loop

Methodology

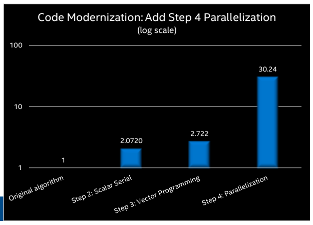
- Use all available processor threads.
- Set affinity mode scattered
- Each thread generate its own data
- Use `parallel`-section and `for`-section wisely

Performance Result

- 28 core, 56 threads
- 30X over single thread vectorized version
- 170X cumulative performance improvement



```
#pragma omp parallel
{
        #ifdef _OPENMP
                threadID = omp_get_thread_num();
        #else
                threadID = 0;
        #endif
        //Allocate memory here
        unsigned int thread_seed = seed + threadID;
        for(int i = OptPerThread-1; i > -1; i--)
        {
                //populate the data for each thread
        }
        #pragma omp barrier
        #pragma omp master
        sTime = second();
        #pragma vector nontemporal (CallResult)
        #pragma simd
        #pragma vector aligned
        for (int opt = 0; opt < OptPerThread; opt++)
        {
                float T = OptionYears[opt];
                float S = StockPrice[opt];
                float X = OptionStrike[opt];
                float b = CostofCarry[opt];
                float v = Volatility[opt];
                CallResult[opt] = baw_scalaropt(S,X,RISKFREE,b, v,
T);
        }
        #pragma omp barrier
        #pragma omp master
        {
                eTime = second();
        }
```



Summary

Converged vector ISA between Xeon and Xeon Phi

- AVX512F is an interesting common subset that covers most of HPC needs

Targeted additions that deliver high value on Xeon Phi

- High precision transcendental instructions and advanced prefetch support

More Complete AVX512 ISA on Xeon

- Support for all data types and vector lengths
- Additional HPC instructions

Significant tools support for AVX512 targeting both Xeon and Xeon Phi

- Compilers and library supports
- Code modernization can lead to significant performance gains from AVX512 and other modern features