AVX512
Dr. Christopher Dahnken
Software and Services Group
Vector ISA Overview
Intel Instruction Set Extension from 1998-2010

<table>
<thead>
<tr>
<th>Year</th>
<th>Extension</th>
<th>New Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1998</td>
<td>Intel® SSE</td>
<td>70 new instructions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4 single-precision vector FP</td>
</tr>
<tr>
<td></td>
<td></td>
<td>scalar FP instructions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>cacheability instructions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>control &amp; conversion instructions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>media extensions</td>
</tr>
<tr>
<td>1999</td>
<td>Intel® SSE2</td>
<td>144 new instructions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 double-precision vector FP</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8/16/32/64 vector integer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>128-bit integer memory &amp; power management</td>
</tr>
<tr>
<td>2004</td>
<td>Intel® SSE3</td>
<td>13 new instructions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FP vector calculation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>x87 integer conversion</td>
</tr>
<tr>
<td></td>
<td></td>
<td>128-bit integer unaligned load thread sync.</td>
</tr>
<tr>
<td>2006</td>
<td>Intel SSSE3</td>
<td>32 new instructions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>enhanced packed integer calculation</td>
</tr>
<tr>
<td>2007</td>
<td>Intel® SSE4.1</td>
<td>47 new instructions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>packed integer calculation &amp; conversion</td>
</tr>
<tr>
<td></td>
<td></td>
<td>better vectorization by compiler</td>
</tr>
<tr>
<td></td>
<td></td>
<td>load with streaming hint</td>
</tr>
<tr>
<td>2008</td>
<td>Intel® SSE4.2</td>
<td>7 new instructions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>string (XML) processing</td>
</tr>
<tr>
<td></td>
<td></td>
<td>POP-Count</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CRC32</td>
</tr>
<tr>
<td>2009</td>
<td>Intel® AES</td>
<td>7 new instructions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6 for acceleration of AES algorithm</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Plus carry-less 64-bit multiply</td>
</tr>
</tbody>
</table>
# Intel Instruction Set Extension after 2010

<table>
<thead>
<tr>
<th>Year</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2011</td>
<td>Intel® AVX</td>
</tr>
<tr>
<td></td>
<td>Promotion of 128 bit FP vector instructions to 256 bit</td>
</tr>
<tr>
<td>2011</td>
<td>Intel® Xeon Phi™ Vector Instruction Set</td>
</tr>
<tr>
<td>2012</td>
<td>“AVX-1.5”</td>
</tr>
<tr>
<td></td>
<td>7 new instructions</td>
</tr>
<tr>
<td></td>
<td>16 bit FP support</td>
</tr>
<tr>
<td></td>
<td>RDRAND</td>
</tr>
<tr>
<td></td>
<td>…</td>
</tr>
<tr>
<td>2013</td>
<td>Intel® AVX-2</td>
</tr>
<tr>
<td></td>
<td>Promotion of integer instruction to 256 bit</td>
</tr>
<tr>
<td></td>
<td>- FMA</td>
</tr>
<tr>
<td></td>
<td>- Gather</td>
</tr>
<tr>
<td></td>
<td>- TSX/RTM</td>
</tr>
<tr>
<td>2014</td>
<td>“Broadwell New Instructions”</td>
</tr>
<tr>
<td></td>
<td>Some 5 new instruction to enhance support for random number generation, prefetching and multi-precision arithmetic</td>
</tr>
<tr>
<td>TBD</td>
<td>Intel® AVX-512</td>
</tr>
<tr>
<td></td>
<td>Promotion of vector instructions to 512 bits and a lot more</td>
</tr>
</tbody>
</table>
Intel® Advanced Vector Extensions

Goal: 8X peak FLOPs over 4 generations

Roadmap illustration - subject to change
Broadwell New Instructions
Codename for 14nm Successor of 4th Generation Intel Core Architecture (code name Haswell)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADCX/ADOX</td>
<td>New instructions accelerating e.g. multi-precision arithmetic and public-key cryptography</td>
</tr>
<tr>
<td>RDSEED</td>
<td>Nondeterministic random number generator (ANSI X9.82, parts 2 &amp; 4)</td>
</tr>
<tr>
<td>PREFETCHW</td>
<td>Prefetch with a hint, that cache line will be modified soon; has been a NOP on Intel processors in the past</td>
</tr>
</tbody>
</table>

- Instructions can only be used by intrinsics thus no need for a new compiler switch
- Intrinsic support already in 14.0 compiler (as well in GCC 4.8)
Introducing AVX-512
Intel® AVX Technology

### AVX
- 256-bit basic FP
- 16 registers
- NDS (and AVX128)
- Improved blend
- MASKMOV
- Implicit unaligned

### AVX2
- Float16 (IVB 2012)
- 256-bit FP FMA
- 256-bit integer
- PERMD
- Gather

### AVX-512
- 512-bit FP/Integer
- 32 registers
- 8 mask registers
- Embedded rounding
- Embedded broadcast
- Scalar/SSE/AVX “promotions”
- HPC additions
- Transcendental support
- Gather/Scatter

### Future Processors (KNL & Xeon)
in planning, subject to change

#### SNB
- 2011

#### HSW
- 2013
AVX-512 features (I): More & Bigger Registers

**AVX / AVX-2:** VADDPS YMM0, YMM3, [mem]
- Up to 16 AVX registers
  - 8 in 32-bit mode
- 256-bit width
  - 8 x FP32
  - 4 x FP64

**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

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

```c
float32 A[N], B[N];
for(i=0; i<8; i++)
{
}
```

```c
float32 A[N], B[N];
for(i=0; i<16; i++)
{
}
```
AVX-512 Mask Registers

8 Mask registers of size 64-bits

- k1-k7 can be used for predication
  - k0 always has value 0xFFFFFFFFFFFFFFFF
  - 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]
  - VADDP S zmm1 {k1}, zmm2, zmm3
- Packed IEEE FP64 and Integer Qword use mask bits [7:0]
  - VADD PD zmm1 {k1}, zmm2, zmm3
Why Separate Mask Registers?

- Don’t waste away real vector registers for vector of bools
- Separate control flow from data flow
- Boolean operations on logical predicates consume less energy (separate functional unit)
- 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)

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

- {z} bit syntax to EVEX z-bit - implies ‘zeroing’ – default is ‘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

```c
float32 A[N], B[N], C[N];
for(i=0; i<16; i++)
{
    if(B[i] != 0) {
    } 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, …

```plaintext
float32 A[N], B[N], C;
for(i=0; i<8; i++)
{
  if ( A[i]!=0.0 )
}
```

```
VBBROADCASTSS zmm1 {k1}, [rax]
VBBROADCASTF64X2 zmm2 {k1}, [rax]
VBBROADCASTF32X4 zmm3 {k1}, [rax]
VBBROADCASTF32X8 zmm4, {k1}, [rax]
...```

Copyright© 2016, Intel Corporation. All rights reserved. *Other brands and names are the property of their respective owners.
AVX-512 Feature: Embedded Rounding Control & SAE
(Suppress All Exceptions)

- 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 any 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 Instruction Subsets and Availability
Intel® AVX-512 for future Intel® XEON and Intel® MIC Architecture

Targeting common ISA across Xeon and MIC

- Significantly converged compared to past
- Some deltas persist

Intel® AVX-512 is first, major step to achieve a unified instruction set; it is split into a family of instruction set extensions (sub sets)

- Some of these sub-sets will be available on next generations of both, MIC and Xeon processors
- Some will be available on either only the next Intel® MIC processor (KNL) or Intel® XEON (TBD) processor
- Many – but not all – of these subsets have been published in detail in summer 2013 – see software.intel.com

Beside new instructions from AVX-512, the future MIC architecture might miss some special purpose instructions available already today on Xeon/x86 like Intel® TSX
The Intel® AVX-512 Subsets

**AVX-512 F**
- 512-bit Foundation instructions common between MIC and Xeon
  - Comprehensive vector extension for HPC and enterprise
  - All the key AVX-512 features: masking, broadcast...
  - 32-bit and 64-bit integer and floating-point instructions
  - Promotion of many AVX and AVX2 instructions to AVX-512
  - Many new instructions added to accelerate HPC workloads

**AVX-512 CD**
- Conflict Detection instructions
  - Allow vectorization of loops with possible address conflict
  - Will show up on Xeon

**AVX-512ER**
- AVX-512 extensions for exponential and prefetch operations
  - Fast (28 bit) instructions for exponential and reciprocal and transcendental functions (as well as RSQRT)
  - New prefetch instructions: gather/scatter prefetches and PREFETCHWT1

**AVX-512PR**
-
# Other New Instructions

<table>
<thead>
<tr>
<th>MPX</th>
<th>Intel® MPX – Intel Memory Protection Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>- Set of instructions to implement checking a pointer against its bounds</td>
</tr>
<tr>
<td></td>
<td>- Pointer Checker support in HW (today a SW only solution of e.g. Intel Compilers)</td>
</tr>
<tr>
<td></td>
<td>- Debug and security features</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SHA</th>
<th>Intel® SHA – Intel Secure Hash Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>- Fast implementation of cryptographic hashing algorithm as defined by NIST FIPS PUB 180</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CLFLUSHOP</th>
<th>Single Instruction – Flush a cache line</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>- needed for future memory technologies</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>XSAVE{S,C}</th>
<th>Save and restore extended processor state</th>
</tr>
</thead>
</table>
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

Useful for pointer manipulation
64-bit becomes a first class citizen
Removes the need for expensive SW emulation sequences

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

*Note: VPMULQ and int64 <-> FP converts not in AVX-512 F*
## Math Support

### Instruction

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VGETXEXP&lt;sub&gt;(PS,PD,SS,SD)&lt;/sub&gt;</td>
<td>Obtain exponent in FP format</td>
</tr>
<tr>
<td>VGETMANT&lt;sub&gt;(PS,PD,SS,SD)&lt;/sub&gt;</td>
<td>Obtain normalized mantissa</td>
</tr>
<tr>
<td>VRNDSCALE&lt;sub&gt;(PS,PD,SS,SD)&lt;/sub&gt;</td>
<td>Round to scaled integral number</td>
</tr>
<tr>
<td>VSCALEF&lt;sub&gt;(PS,PD,SS,SD)&lt;/sub&gt;</td>
<td>Approx. reciprocal() with rel. error 2&lt;sup&gt;-14&lt;/sup&gt;</td>
</tr>
<tr>
<td>VFIXUPIMM&lt;sub&gt;(PS,PD,SS,SD)&lt;/sub&gt;</td>
<td>Patch output numbers based on inputs</td>
</tr>
<tr>
<td>VRCP14&lt;sub&gt;(PS,PD,SS,SD)&lt;/sub&gt;</td>
<td>Approx. rsqrt() with rel. error 2&lt;sup&gt;-14&lt;/sup&gt;</td>
</tr>
<tr>
<td>VRSQRT14&lt;sub&gt;(PS,PD,SS,SD)&lt;/sub&gt;</td>
<td>IEEE division</td>
</tr>
<tr>
<td>VDIV&lt;sub&gt;(PS,PD,SS,SD)&lt;/sub&gt;</td>
<td>IEEE square root</td>
</tr>
<tr>
<td>VSQRT&lt;sub&gt;(PS,PD,SS,SD)&lt;/sub&gt;</td>
<td></td>
</tr>
</tbody>
</table>

### 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

---

Copyright© 2016, Intel Corporation. All rights reserved. *Other brands and names are the property of their respective owners.*
# New 2-Source Shuffles

## 2-Src Shuffles

<table>
<thead>
<tr>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>VSHUF(PS,PD)</td>
</tr>
<tr>
<td>VPUNPCK(H,L)(DQ,QDQ)</td>
</tr>
<tr>
<td>VUNPCK(H,L)(PS,PD)</td>
</tr>
<tr>
<td>VPERM(I,D)<a href="D,Q,PS,PD">2</a></td>
</tr>
<tr>
<td>VSHUF(F,I)32X4</td>
</tr>
</tbody>
</table>

## Long standing customer request

- 16/32-entry table lookup (transcendental support)
- AOS ↔ SOA support, matrix transpose
- Variable VALIGN emulation

### Example

Input: ` EVEX.U1.512.NDS.66.0F38.W1 A V/V AVX3.1 `  

```
VPERM2PD zmm1 [k1][z], zmm2, zmm3/B64(mV)
```

Output:

```
zmm1  \[15\ 0\ 10\ 11\ 2\ 2\ 0\ 9\]
zmm2  \[10\ 9\ 8\ 7\ 6\ 5\ 4\ 3\ 2\ 1\ 0\]
zmm3  \[H\ G\ F\ E\ D\ C\ B\ A\]
```

---

*Other brands and names are the property of their respective owners.*
Expand & Compress

VEXPANDPS  zmm0 {k2}, [rax]

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

\[ \text{for}(j=0, i=0; i<N; i++) \]
\[ \{ \]
\[ \quad \text{if}(C[i] \neq 0.0) \]
\[ \quad \quad B[i] = A[i] \times C[j++] ; \]
\[ \} \]

Allows vectorization of conditional loops
- Opposite operation (compress) not in AVX-512F
- Similar to FORTRAN pack/unpack intrinsics
- Provides mem fault suppression
- Faster than alternative gather/scatter
## Bit Manipulation

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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>KUNPCKBW k1, k2, k3</td>
<td>Interleave bytes in k2 and k3</td>
</tr>
<tr>
<td>KSHIFT{L,R}W k1, k2, imm8</td>
<td>Shift bits left/right using imm8</td>
</tr>
<tr>
<td>VPROR{D,Q} zmm1 {k1}, zmm2, imm8</td>
<td>Rotate bits right using imm8</td>
</tr>
<tr>
<td>VPROL{D,Q} zmm1 {k1}, zmm2, imm8</td>
<td>Rotate bits left using imm8</td>
</tr>
<tr>
<td>VPRORV{D,Q} zmm1 {k1}, zmm2, zmm3/mem</td>
<td>Rotate bits right w/ variable ctrl</td>
</tr>
<tr>
<td>VPROLV{D,Q} zmm1 {k1}, zmm2, zmm3/mem</td>
<td>Rotate bits left w/ variable ctrl</td>
</tr>
</tbody>
</table>
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 with andor, andxor, vote, parity, bitwise-cmov, etc. Each column in the right table corresponds to imm8.
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:

```c
for(i=0; i<16; i++) { 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

```plaintext
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)
```

This not even is the fastest version
AVX-512 ER & AVX-512 PR
AVX-512 ER & AVX-512 PR

Set of segment-specific instruction extensions
- First appear on KNL
- Will be supported in all future Intel® MIC processors

Address two HPC customer requests
- Ability to maximize memory bandwidth
  - Hardware prefetching is too restrictive
  - Conventional software prefetching results in instructions overhead
- Flexible support for transcendental operations - accuracy versus speed
  - Mostly division and square root
  - Differentiating factor in HPC/TPT
# Instruction Set Description

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

<table>
<thead>
<tr>
<th>CPUID</th>
<th>Instructions</th>
<th>Motivation</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVX-512 PRI</td>
<td>PREFETCHWT1</td>
<td>Reduce coherent traffic in core-to-core data communication</td>
</tr>
<tr>
<td>AVX-512 PRI</td>
<td>VGATHERPF{D,Q}{0,1}PS</td>
<td>Reduce overhead of software prefetching: <em>dedicate side engine to prefetch sparse structures while devoting the main CPU to pure raw flops</em></td>
</tr>
<tr>
<td>AVX-512 PRI</td>
<td>VSCATTERPF{D,Q}{0,1}PS</td>
<td></td>
</tr>
<tr>
<td>AVX-512ERI</td>
<td>VEXP2{PS,PD}</td>
<td>Speed-up key FSI workloads: Black-Scholes, Montecarlo</td>
</tr>
<tr>
<td>AVX-512ERI</td>
<td>VRCP28{PS,PD}</td>
<td>Key building block to speed up most transcendental sequences (in particular, division and square root): <em>Increasing precision from 14=&gt;28; removes one complete Newton-Raphson iteration</em></td>
</tr>
<tr>
<td>AVX-512ERI</td>
<td>VRSQRT28{PS,PD}</td>
<td></td>
</tr>
</tbody>
</table>
Summary AVX-512F, CD, ER, PR

AVX-512 F: new 512-bit vector ISA extension
- Common between Xeon and Xeon Phi (KNL)

AVX-512 CD Conflict detection instructions
- Improves autovectorization

AVX-512 ER & PR
- 28-bit transcendentalals and new prefetch instructions
- On Intel® MIC only

Developer tools support
- Intel® Compilers 14.x/15.x via switch –xmic-512 etc
- Intel® Software Development Emulator (Intel® SDE)
References / Summary

Reference manual for public Intel® AVX-512 instructions and a lot more


Great discussion on Intel forum / Agner’s critical view on AVX-512 -

▪ see too www.agner.org
References / Summary

Reference manual for public Intel® AVX-512 instructions and a lot more


Great discussion on Intel forum / Agner’s critical view on AVX-512 -

- see too www.agner.org
Legal Disclaimer & Optimization Notice

INFORMATION IN THIS DOCUMENT IS PROVIDED “AS IS”. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

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.

Copyright © , Intel Corporation. All rights reserved. Intel, the Intel logo, Xeon, Xeon Phi, Core, VTune, and Cilk are trademarks of Intel Corporation in the U.S. and other countries.

Optimization Notice

Intel’s compilers 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 SSE2, SSE3, and 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. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804
VPCONFLICT\{D,Q\}

- VPCONFLICT\{D,Q\} \(zmm1\{k1\}\{z\}, \text{zmm2/B(mV)}\)
  - 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
}