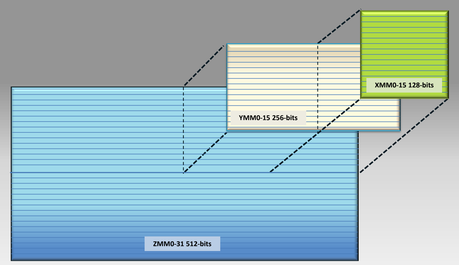
Intel® AVX-512 Architecture

Comprehensive vector extension for HPC and enterprise

Elena Demikhovsky
Intel® Software and Services Group
Israel

More And Bigger Registers



32 SIMD registers 512 bit wide

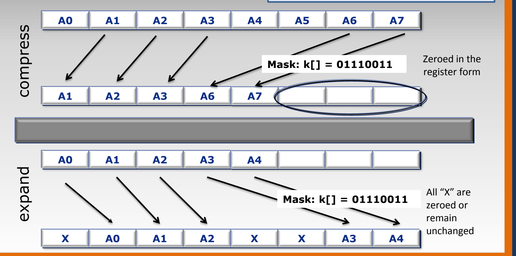
Compress And Expand

Compress values from 512-bit vectors compound of f64, f32, i64, i32 elements using mask and store in register or memory

VCOMPRESSPD zmm1/mV {k1}, zmm2

VEXPANDPS zmm1 {k1}{z}, zmm2/mV

```
for (i =0; i < N; i++) {
  if (topVal > b[i])   {
     *dst = a[i];
      dst++;
   }
}
```



Conflict Detection

Sparse computations are hard for vectorization

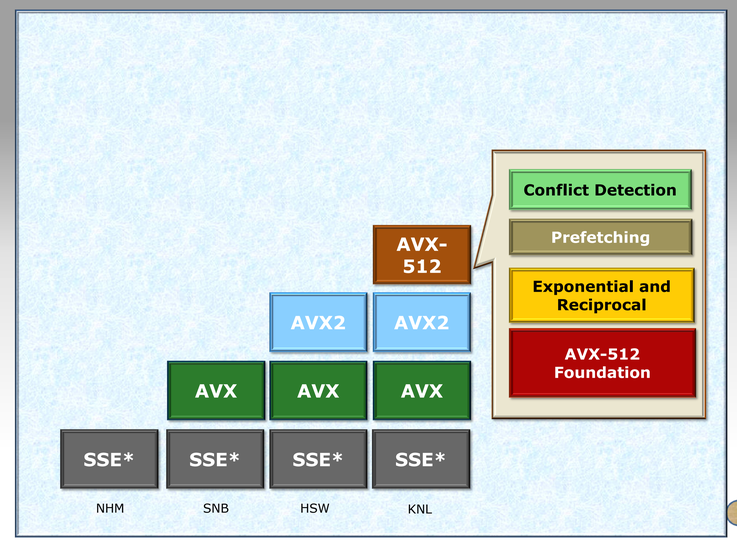
```
for(i=0; i<16; i++) {  A[B[i]]++; }

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

VPCONFLICT instruction detects elements with conflicts

```
index = vload &B[i]                                  // Load 16 B[i]
pending_elem = 0xFFFF;
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)
```

AVX-512 – What's new?



- *512-bit wide vectors, 32 SIMD registers*
- *8 new mask registers*
- *Embedded Rounding Control*
- *Embedded Broadcast*
- *New Math instructions*
- *2-source shuffles*
- *Gather and Scatter*
- *Compress and Expand*
- *Conflict Detection*

Embedded Broadcast

A source from memory is repeated across all the elements.

```
vbroadcastss zmm3, [rax]
vaddps zmm1, zmm2, zmm3

vaddps zmm1, zmm2, [rax] {1to16}
```

Embedded Rounding Control

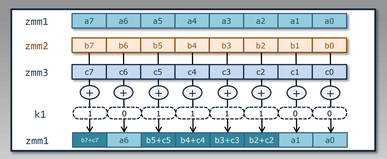
- Static (per instruction) rounding rode
- No need to access MXCSR any more!

```
vaddps zmm7 {k6}, zmm2, zmm4 {rd}
vcvtdq2ps zmm1, zmm2, {ru}
```

All exceptions are always suspended by using embedded RC

Masking

Unmasked elements remain unchanged:
VADDPD zmm1 {k1}, zmm2, zmm3
Or zeroed:
VADDPD zmm1 {k1} {z}, zmm2, zmm3



- **Memory fault suppression**
- **Avoid FP exceptions**
- **Avoid extra blends**

```
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
```

Masking in LLVM

Predication Scheme

Source code

```
If (condition) {

  A = ..
} else {

  A = ..
}
```

LLVM IR

```
%Mask = cmp (condition)
 …
 %A1 =
 …
 %A2 =
 %A = SELECT %Mask, %A1, %A2
```

Machine code

```
CMP %regMask, ops
 …
ADD %regA1, x1, y1
 …
ADD %regA2 = x2, y2
BLEND %regA, %regMask, %regA1, %regA2
```

Goal:
To set predicates for instructions that calculate A1 and A2 (Mask and Not-Mask)

Result:
If the mask is all-zero, instruction will not be executed.

```
Mask = cmp (condition)
Not-Mask = not(Mask)
 {Mask} C1 =
 {Mask} B1 =
 {Mask} A = B1+C1

 {Not-Mask} C2 =
 {Not-Mask} B2 =
 {Not-Mask} A = B2+C2
```

Mask Propagation Pass – design ideas

```
topVal =
Mask = cmp()

C1 = topVal*2
B1 =
A1 = B1 + C1
C2 =
B2 =
A2 = B2 + C2
A = BLEND(Mask, A1, A2)
```

```
topVal =
Mask = cmp()

Mask C1 = topVal*2
Mask B1 =
Mask A1 = B1 + C1
     C2 =
     B2 =
     A2 = B2 + C2
A = BLEND(Mask, A1, A2)
```

```
topVal =
Mask = cmp()
!Mask = Not(Mask)
Mask C1 = topVal*2
Mask B1 =
Mask A1 = B1 + C1
!Mask C2 =
!Mask B2 =
!Mask A2 = B2 + C2
  A = BLEND(Mask, A1, A2)
```

```
topVal =
Mask = cmp()
!Mask = Not(Mask)
Mask C1 = topVal*2
Mask B1 =
Mask A = B1 + C1
!Mask C2 =
!Mask B2 =
!Mask A = B2 + C2
```

A new Machine Pass:
- Before register allocation
- Start from the "blend" operands and go up recursively till mask definition
- Check all users of the destination operand before applying the mask

❖ Mask Propagation Pass does not guarantee full mask propagation over the whole path from blend to compare

❖ Load/Store operations require exact masking

❖ FP operations require masking if exceptions are not suppressed

– IR generators should use compiler intrinsics

Predicated memory accesses in LLVM IR would be helpful !