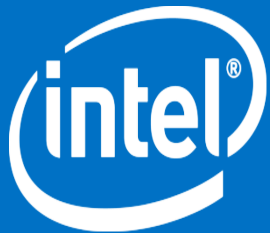


Vectorization of Control Flow with New Masked Vector Intrinsics

Elena Demikhovsky

Intel® Software and Services Group – Israel

April, 2015

2015 European LLVM Conference
London, England

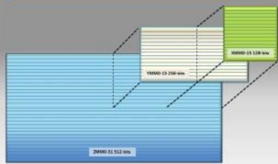
Poster at 2013 US LLVM Developers' Meeting

Intel® AVX-512 Architecture

Comprehensive vector extension for HPC and enterprise

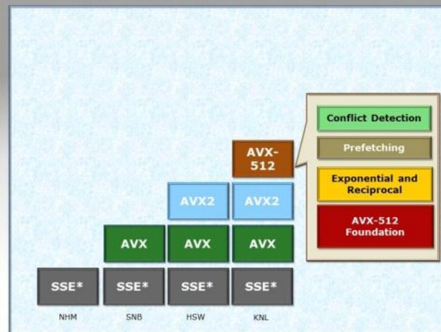
Elena Demikhovsky
Intel® Software and Services Group
Israel

More And Bigger Registers



32 SIMD registers 512 bit wide

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}
```

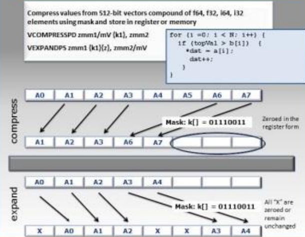
Embedded Rounding Control

- 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

Compress And Expand



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)
```

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

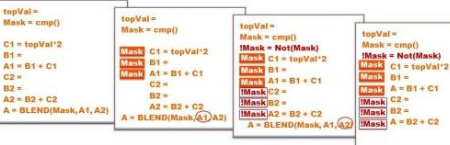
```
CMP %regMask, ops
...
ADD %regA1, x1, y1
...
ADD %regA2 = x2, y2
BLEND %regA, %regMask, %regA1, %regA2
```

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



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 !

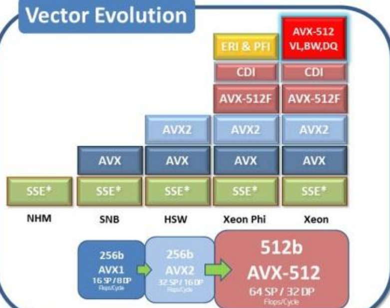
Poster at 2014 US LLVM Developers' Meeting

Intel® AVX-512 architecture evolution and support in Clang/LLVM

2014 LLVM Developers' Meeting, Oct 28-29

Robert.Khasanov
Zinovy.Y.Nis
@intel.com

Vector Evolution



AVX-512 Features

- 512b-wide vectors(%ZMM0-31)
- Masked instructions, 64b-mask registers (%K0-7)
- Gathers/Scatters
- Permutations
- Embedded broadcast
- Embedded rounding control
- Compressed displacement
- Embedded suppression of all exceptions

AVX-512 in Clang/LLVM

- **Total: 651 instructions, 4000+ intrinsics**
- 30% of these instructions implemented
- Encodings, lowering and intrinsics covered with tests
- 100+ patches, 9000+ LOCs
- Work in progress!

Available in trunk since July 2014 !

clang -march=knl... | clang -march=skx ...

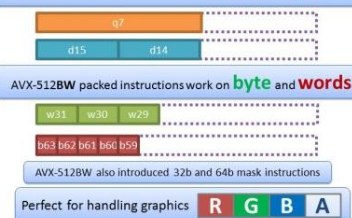
New features

AVX-512VL: Vector Length Orthogonality

Apply **AVX-512F** instructions to **128b (%XMM)** and **256b (%YMM)** registers

- AVX-512**F** (starting with Xeon Phi)
  - VADDPD (%rcx), **%zmm2, %zmm3**
- AVX-512**{F,VL}** (starting with Skylake Xeon)
  - VADDPD (%rcx), **%ymm2, %ymm3**
  - VADDPD (%rcx), **%xmm2, %xmm3**

AVX-512BW: Byte & word support (131)



Perfect for handling graphics **R G B A**

AVX-512DQ: New HPC instructions (64)

Extended Tuple support: 32x8, 64x2, 32x2	INT64 arithmetic support
Int64 ⇔ FP conversions	Byte support for mask instructions
Transcendental package enhancements	Expanded mask functionality

Enabling compiler optimizations

If-Conversion with memory accesses

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

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

Currently, this loop can't be vectorized in LLVM IR:

```
LV: Found a loop: for.body
LV: Can't if-convert the loop.
LV: Not vectorizing: Cannot prove legality.
```

Merge-masking:
VADDPD zmm1 {k1}, zmm2, zmm3
$dest_j \leftarrow mask_j{==}1\ ?\ src1_j + src2_j : dest_j$

Zero-masking:
VADDPD zmm1 {k1} {z}, zmm2, zmm3
$dest_j \leftarrow mask_j{==}1\ ?\ src1_j + src2_j : 0$

With AVX-512 we can generate this smart code!

```
VCMPNEQPS k1, zmm0, B
VMOVUPS zmm2 { k1}, A
VMULPS  zmm1 {k1}, zmm2, B
VMOVUPS A{k1}, zmm1
```

Potential LLVM IR extended with special intrinsics for masking

```
%a= call <16 x float> @llvm.masked.load (<16 x float>*  %a.ptr, <16 x i1> %mask)
%mul= call <16 x float> @llvm.masked.fmul (<16 x float> %a, %b, <16 x i1> %mask)
call void @llvm.masked.store(<16 x float> %mul,  <16 x float>* %a.ptr, <16 x i1> %mask)
```

Vectorization of Peeled Loops

```
float A[N], B[N], C[N];
...
for (i=0; i < N; ++i)  {
   C[i] = A[i] + B[i];
}
```

```
i = 0;
V = N – (N % VF); // VF is # of elements in vector
// Vector part
for ( ; i < V; i += VF) {
   // vectorized!
   C[i:i+VF-1:1] =  A[i:i+VF-1:1] + B[i:i+VF-1:1];
}
// Peeled part
//  Not vectorized!
for ( ; i < N; ++i) C[i] = A[i] + B[i];
```

```
// Vectorized
VMOVUPS ZMM1, ZERO_VEC
VADDPD ZMM1, ZMM1, A[0]  // 0..7
VADDPD ZMM1, ZMM1, B[0]
VMOVUPS C[0], ZMM1
...
// Peeled part
// Clone of loop body but with masks
KMOVW K1,  MASK // Here, MASK is N % VF
VMOVUPS ZMM1, ZERO_VEC
VADDPD ZMM1 {k1},  ZMM1, A[32] // 32..36
VADDPD ZMM1 {k1},  ZMM1, B[32]
VMOVUPS C[32] {k1}, ZMM1
```

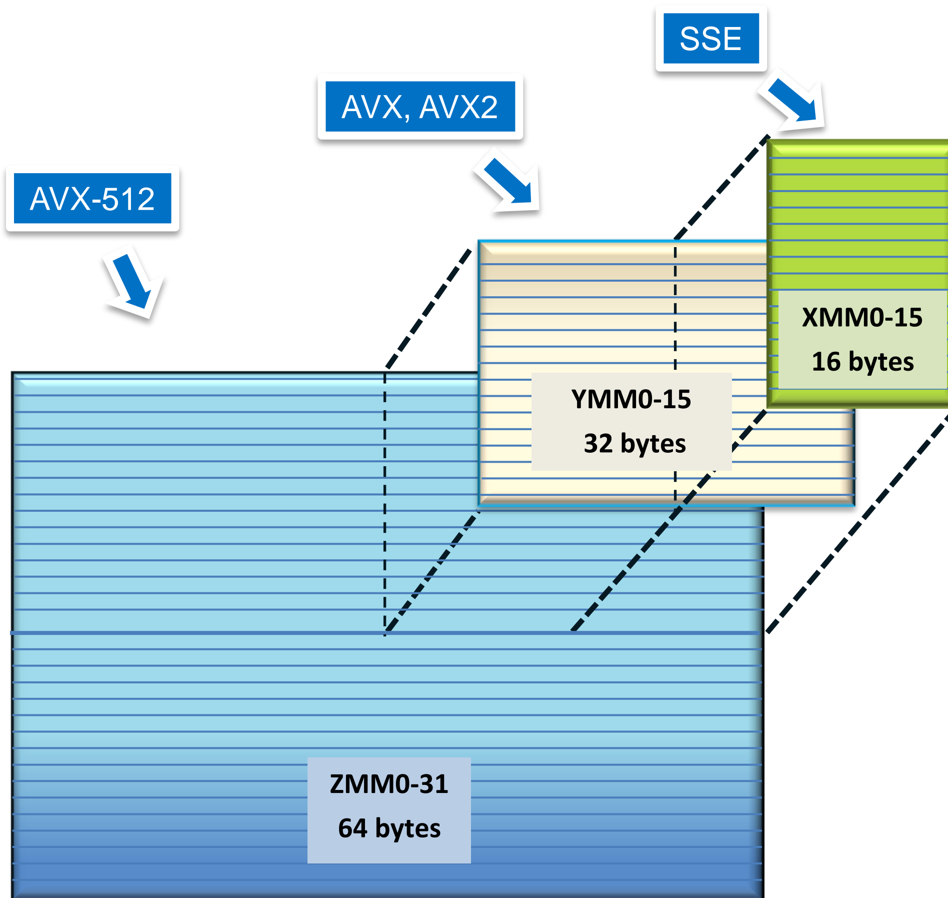
More samples

*Other names and brands may be claimed as the property of others.

Legal Disclaimer

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 © 2014, Intel Corporation. All rights reserved. Intel, Pentium, Xeon, Xeon Phi, Core, VTune, Cilk, and the Intel logo are trademarks of Intel Corporation in the U.S. and other countries. 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.

AVX-512 - Greatly increased register file



32 x 512 bit registers

- Higher throughput
- Greatly improved unrolling and inlining opportunities

SIMD instructions

- arithmetic operations, integer and FP
- logical operations
- memory, including gather and scatter
- vector shuffles
- But! The branch remains scalar
  - no multiway branches in SIMD

Masking in AVX-512

- New feature of AVX-512?
  - We have the “maskmov” instruction in AVX for masking load and store

- What’s new?
  - Special mask registers
    - 8 new 64-bit registers
    - 1 bit per vector lane, up to 64 lanes
  - Result of comparison is written to the mask register
  - Used in instructions to select vector lanes
  - Masked-off elements remain unchanged or zeroed

```
VCMPPS k3, zmm26, zmm30              // k3 <- comparison result
VADDPS zmm1 {k3},    zmm2, zmm3      // Masked-off elements remain unchanged
VADDPS zmm1 {k3}{z}, zmm2, zmm3      // Masked-off elements are zeroed
```

Why Masking?

- Masking operations is the next most significant step in vectorization
  - Mask Load and Store to avoid memory access violations
  - Mask FP operation to avoid FP exceptions
- Masked vector instructions enable direct vectorization of code regions with control flow divergence

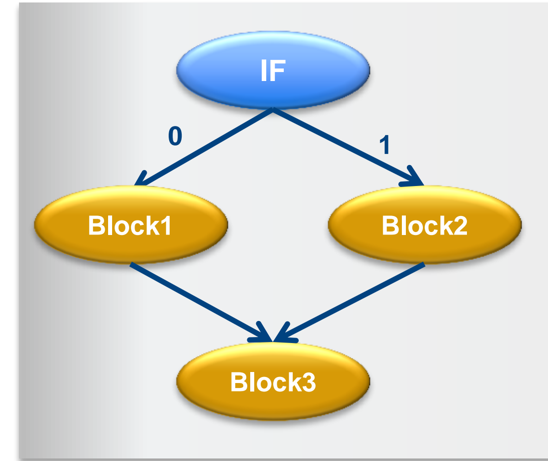
LLVM IR - no masking support

- Why LLVM is not interested in masked instructions?
  - Most of targets do not support masked instructions
  - Optimization of instructions with masks is problematic
- What happens when we try to vectorize a loop with control flow divergence?
  - Avoid masks if you can

But we know we can't always avoid masking.
How do we move forward?

Avoid masks if you can

```
for (int i = 0; i < N; i++) {
  if (Trigger[i] < Val)
    A[i] = B[i] + 0.5;
  else
    A[i] = B[i] - 1.5;
}
```



There are many existing techniques that allow loop vectorization without masks

- Static divergence analysis to identify uniform branches
- Hoisting and sinking of equivalent operations
- If-conversion with blend

The focus here is on transformation capabilities

Assume dependence analysis is done and transformation is legal

Avoid masks if you can

```
for (int i = 0; i < N; i++) {
  if (Trigger[i] < Val)
    A[i] = B[i] + 0.5;
  else
    A[i] = B[i] - 1.5;
}
```

Hoist Loads

```
for (int i = 0; i < N; i++) {
  TmpB = B[i];
  if (Trigger[i] < Val)
    A[i] = TmpB + 0.5;
  else
    A[i] = TmpB - 1.5;
}
```

Avoid masks if you can

```
for (int i = 0; i < N; i++) {
  if (Trigger[i] < Val)
    A[i] = B[i] + 0.5;
  else
    A[i] = B[i] - 1.5;
}
```

Hoist Loads

```
for (int i = 0; i < N; i++) {
  TmpB = B[i];
  if (Trigger[i] < Val)
    A[i] = TmpB + 0.5;
  else
    A[i] = TmpB - 1.5;
}
```

Sink Stores

```
for (int i = 0; i < N; i++) {
  TmpB = B[i];
  if (Trigger[i] < Val)
    TmpA = TmpB + 0.5;
  else
    TmpA = TmpB - 1.5;
  A[i] = TmpA
}
```

Avoid masks if you can

```
for (int i = 0; i < N; i++) {
  if (Trigger[i] < Val)
    A[i] = B[i] + 0.5;
  else
    A[i] = B[i] - 1.5;
}
```

Hoist Loads

```
for (int i = 0; i < N; i++) {
  TmpB = B[i];
  if (Trigger[i] < Val)
    A[i] = TmpB + 0.5;
  else
    A[i] = TmpB - 1.5;
}
```

Sink Stores

```
for (int i = 0; i < N; i++) {
  TmpB = B[i];
  if (Trigger[i] < Val)
    TmpA = TmpB + 0.5;
  else
    TmpA = TmpB - 1.5;
  A[i] = TmpA
}
```

Blend

```
for (int i = 0; i < N; i+=16) {
  TmpB = B[i:i+15];
  Mask = Trigger[i:i+15] < Val
  TmpA1 = TmpB + 0.5;
  TmpA2 = TmpB - 1.5;
  TmpA = BLEND Mask, TmpA1, TmpA2
  A[i:i+15] = TmpA;
}
```

Memory access under divergent control

```
for (int i = 0; i < N; i++) {
  if (Trigger[i] < Val) {
    A[i] = B[i] + 0.5;
  }
}
```

Here we cannot avoid masking!

- Load hoisting and Store sinking is not allowed in this case
- Masking of memory operation is required in order to vectorize this loop

Our goal

- Allow LLVM compiler to make best use of Advanced SIMD architectures
  - Including Intel AVX and AVX-512
- Do not complicate code for other targets
  - Avoid introducing new masked instructions into LLVM IR

Not Instructions? Let's go for Intrinsics.

- Consecutive memory access – Masked Vector Load and Store
- The syntax is coherent with instruction

Load

```
%Val = load <4 x double>, <4 x double>* %Ptr, align 8
```

```
%Val = call <4 x double> @masked.load.v4f64 (<4 x double>* %Ptr, i32 8,
                                 <4 x i1> %Mask, <4 x double> %PassThru)
```

Store

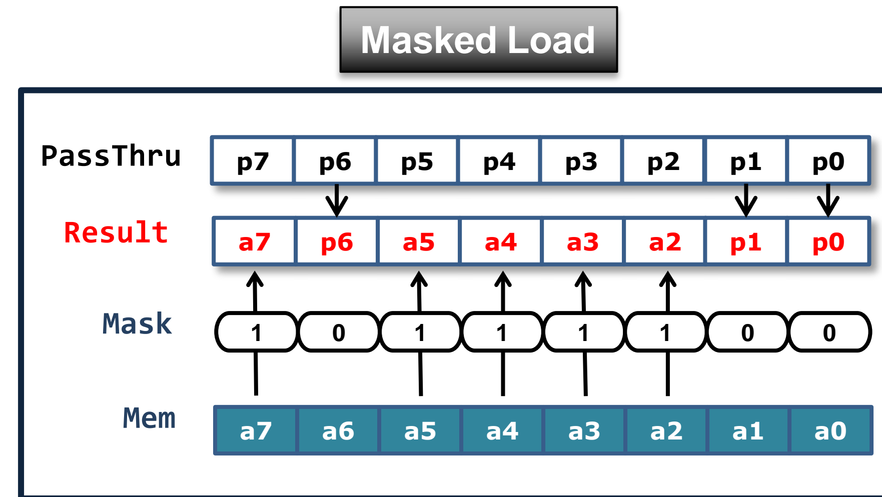
```
store  <4 x double> %Val, <4 x double>* %Ptr, align 8
```

```
call void @masked.store.v4f64 (<4 x double> %Val, <4 x double>* %Ptr, i32 8,
                                 <4 x i1> %Mask)
```

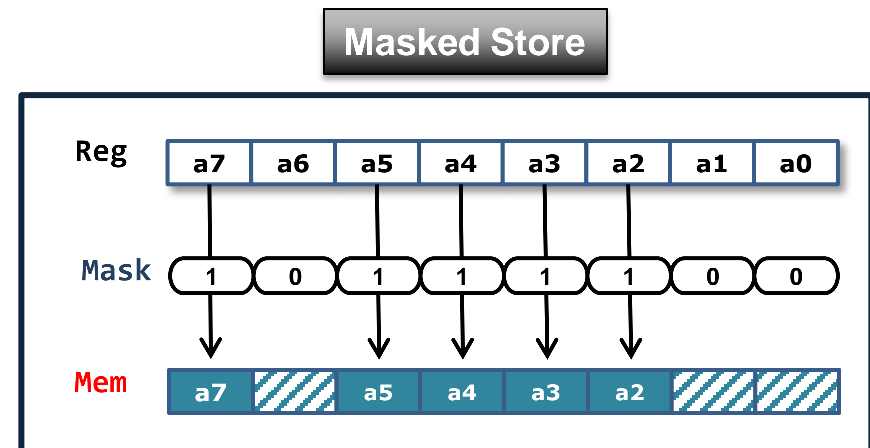
Masked Vector Load and Store

- Access memory according to the provided mask
- The mask holds a bit for each vector lane
- While loading, the masked-off lanes are taken from the PassThru operand.
- No memory access for all-zero mask
- All ones mask is equal to the regular vector Load / Store

Masked Load



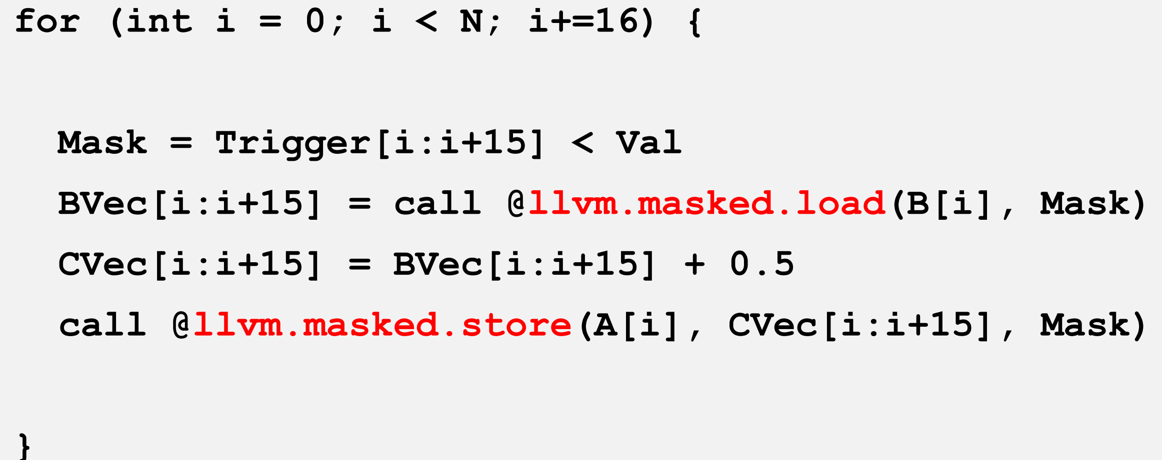
Masked Store



Legal Disclaimer & Optimization Notice

Vectorizing the loop with Masked Load and Store

```
for (int i = 0; i < N; i++) {
  if (Trigger[i] < Val) {
    A[i] = B[i] + 0.5;
  }
}
```



Vectorizing the loop with Masked Load and Store

```
for (int i = 0; i < N; i++) {
  if (Trigger[i] < Val) {
    A[i] = B[i] + 0.5;
  }
}
```

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

  Mask = Trigger[i:i+15] < Val
  BVec[i:i+15] = call @llvm.masked.load(B[i], Mask)
  CVec[i:i+15] = BVec[i:i+15] + 0.5
  call @llvm.masked.store(A[i], CVec[i:i+15], Mask)

}
```

Note, this instruction is not masked, although could be!

Legal Disclaimer & Optimization Notice

Who generates masked intrinsics?

Vectorizer generates masked loads and stores when:

- Load / Store instruction inside predicated basic block
- Memory access is consecutive for induction variable, regardless of the mask
  - A[i] is consecutive, A[i*2] is not
- Target supports masked operation
- Cost model shows potential performance gain
  - AVX and AVX2 have the “maskmov” instructions, designed to avoid executing a chain of conditional scalar operations
  - AVX-512 has more efficient support for masked operations than AVX

SLP Vectorizer can also benefit from this feature

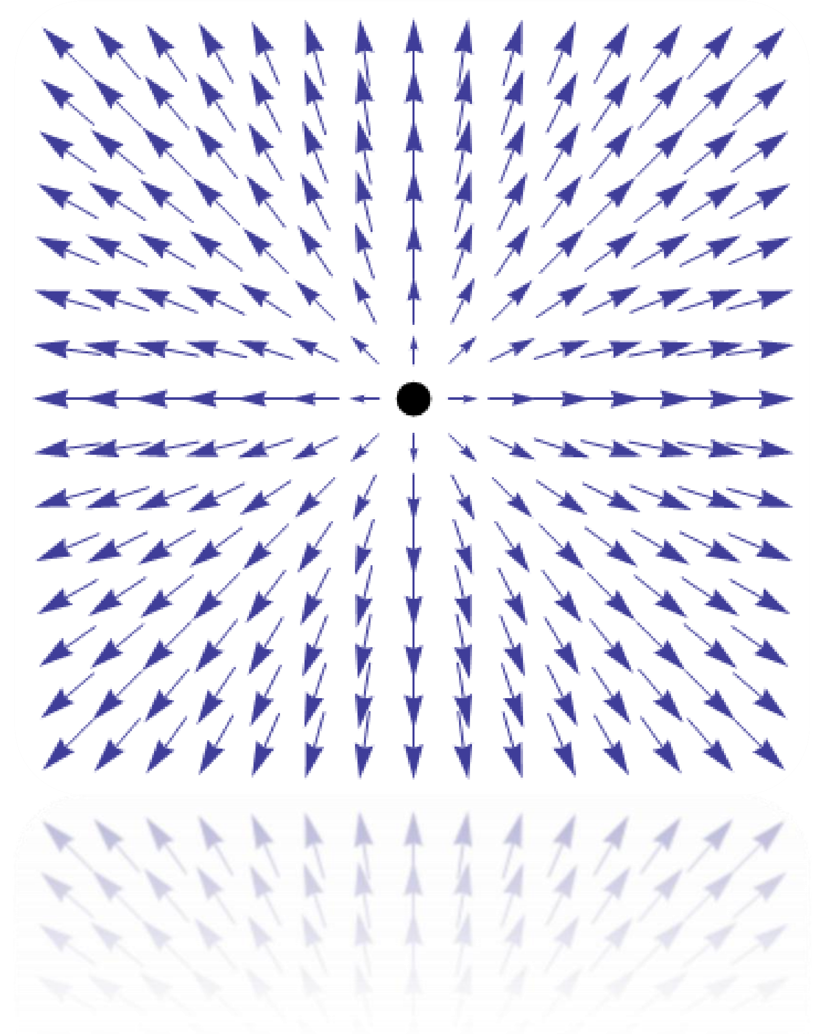
Targets and Data Types

- Target independent syntax
  - CodeGenPrepare Pass scalarizes the masked intrinsic if target does not support it
- Overloaded vector types

```
<4 x double> @masked.load.v4f64 (<4 x double>* %Ptr, i32 8, <4 x i1> %Mask,
                                  <4 x double> %PassThru)
void @masked.store.v16i32 (<16 x i32> %Val, <16 x i32>* %Ptr, i32 4,
                           <16 x i1> %Mask)
```

We talked about Control Flow Divergence.

And what happens with Data Divergence?



Data Divergence

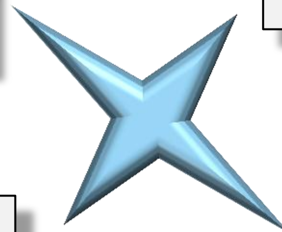
Non-consecutive memory access?

Strided Read

```
for (i=0; i< size; i++)
  Sum += B[i*2]
```

Random Read

```
for (i=0; i< size; i++)
   Sum += B[C[i]]
```



Consecutive Reads and Random Write

```
for (i=0; i<size; i++)
  out[index[i]] = in[i] + 0.5;
```

Predicated non-consecutive Read

```
for (i=0; i< size; i++)
  if (trigger[i])
    A[i] += B[i*i]
```

Intel AVX-512 architecture has masked gather and scatter instructions – all these loops may be vectorized

Solution for Random Memory Access

- Vector Gather and Scatter Intrinsics
- With mask, (the mask may be all-ones)

Load / Gather

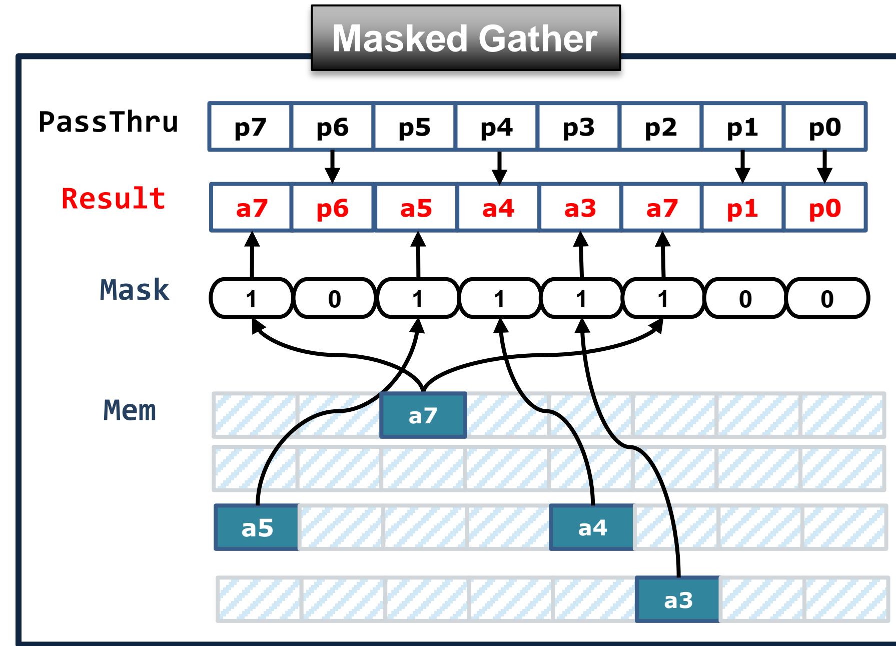
```
%Val = call <4 x double> @masked.load.v4f64   (<4 x double>* %Ptr, i32 8,
                                               <4 x double> %PassTru)
                                               <4 x i1> %Mask,
%Val = call <4 x double> @masked.gather.v4f64 (<4 x double*> %Ptrs, i32 8,
                                               <4 x i1> %Mask,
                                               <4 x double> %PassTru)
```

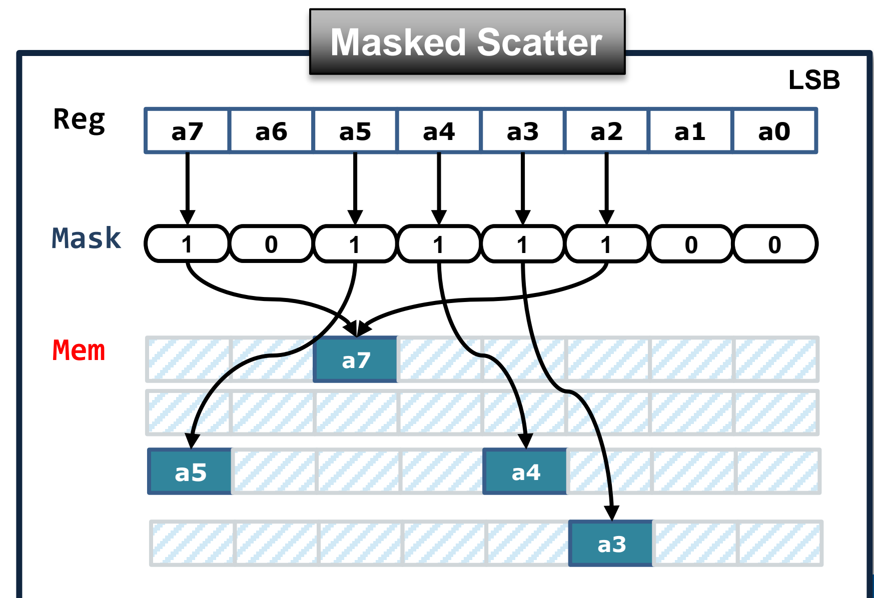
Store / Scatter

```
@masked.store.v4f64   (<4 x double> %Val, <4 x double>* %Ptr, i32 8,
                       <4 x i1> %Mask)
@masked.scatter.v4f64 (<4 x double> %Val, <4 x double*> %Ptrs, i32 8,
                       <4 x i1> %Mask)
```

Gather And Scatter
How does it work?

- Works with vector of pointers
- Access memory according to the provided mask
- The mask holds a bit per lane
- The masked-off lanes are taken from the PassThru operand.
- No memory access for all-zero mask
- Scatter with overlapping vector indices are guaranteed to be ordered from LSB to MSB





Gather and Scatter Intrinsics

When do we use them?

- Memory access is random with respect to induction variable
  - Strides A[i*2],
  - Multi-dimensional arrays A[i][j],
  - Variable indices A[B[i]]
  - Structures A[i].b
- Target should support gather and scatter
- Cost model shows potential performance gain

Masked Gather - Example

```
for (unsigned i=0; i<size; i++) {
    if (trigger[i] > 0)
      out[i] = in[index[i]] + (double) 0.5;
}
```

Masked Load +
Masked Gather +
Masked Store

```
%mask = icmp sgt <8 x i32> %trigger, zeroinitializer

// load "index" array
%index = call <8 x i32> @llvm.masked.load.v8i32(<8 x i32>* %index_ptr, i32 4,
                                            <8 x i1> %mask, <8 x i32> undef)
%se_index = sext <8 x i32> %index to <8 x i64>

// Prepare vector GEP - broadcast base + vector index

%ptrs = getelementptr <8 x double*> %brcst_in, <8 x i64> %se_index

%vin = call <8 x double> @llvm.masked.gather.v8f64(<8 x double*> % ptrs,
                                                   i32 8, <8 x i1> %mask..)

%res = fadd <8 x double> %vin, <double 5.000000e-01, double 5.000000e-01, ..>

call void @llvm.masked.store.v8f64(<8 x double> %out, <8 x double>* %res,
                                   i32 8, <8 x i1> %mask)
```

Gather

Strided memory access

Strided access is a specific case of gather / scatter

---- Stride is a compile time constant ----

```
for (unsigned i=0; i<size; i++) {
  out[i] = in[i*2] + (double) 0.5;
}
```

Why we are talking about strides?

- Gather is faster than scalar loads but still expensive
- Vector Load + Shuffle is more optimal in many cases
- Not all targets support “gathers”

Strided memory access – what can be done?

```
for (unsigned i=0; i<size; i++) {
  out[i] = in[i*2] + (double) 0.5;
}
```

A. Create gather intrinsic and optimize it later
B. Create loads + shuffles
C. Introduce another intrinsic, for example

```
<8 x double>
@llvm.strided.load.v8f64(double *%ptr, i32 2 /*stride*/ ..)
```

Or with indices

```
<8 x double>
@llvm.indexed.load.v8f64(double *%ptr, <i32 0, i32 2, i32 4 .. >)
```

Legal Disclaimer & Optimization Notice

Strided memory access with mask

```
for (unsigned i=0; i<size; i++) {
  if (trigger[i]) {
    out[i] = in[i*2] + (double) 0.5;
  }
}
```

- Masked load with stride

```
<8 x double>
@llvm.masked.strided.load.v8f64(double *%ptr, i32 2 /*stride*/,
                                <8 x i1> %mask,
                                <8 x double> %PassThru)
```

- Masked load with indices

```
<8 x double>
@llvm.masked.indexed.load.v8f64(double *%ptr,
                                <i32 0, i32 2, i32 4 .. >,
                                <8 x i1> %mask,
                                <8 x double> %PassThru)
```

Legal Disclaimer & Optimization Notice

Gather for a strided access - Example

```
for (unsigned i=0; i<size; i++) {
  out[i] = in[i*2] + (double) 0.5;
}
```

A. Create a “gather”

```
// get sequential indices
%splat_i = insertelement <8 x i64> undef, i64 %i, i32 0
%brcst.i = shufflevector <8 x i64> % splat_i, <8 x i64> undef, <8 x i32> zeroinitializer
%induction = add <8 x i64> % brcst.i, <i64 0, i64 1, i64 2, i64 3, i64 4, i64 5, i64 6, i64 7>

// set the stride
%strided_index = shl <8 x i64> %induction, <i64 1, i64 1, i64 1, i64 1, i64 1, i64 1, i64 1,
i64 1>

// get the vector of pointers
%splat_in = insertelement <8 x double*> undef, double* %in, i32 0
%brcst.in = shufflevector <8 x double*> % splat_in, <8 x double*> undef, <8 x i32>
zeroinitializer
%gep.random_access = getelementptr <8 x double*> % brcst.in, <8 x i64> %strided_index

// gather (load) all values
%even = call <8 x double> @llvm.masked.gather.v8f64(<8 x double*> %gep.random_access, i32 8,
                                                    <8 x i1> <1,1,1..>, <8 x double> undef)
```

Gather for a strided access - Example

```
for (unsigned i=0; i<size; i++) {
  out[i] = in[i*2] + (double) 0.5;
}
```

B. Create loads + shuffles

```
// Stride is 2 - you need 2 loads and 1 shuffle
// Stride is 4 - you need 4 loads and 3 shuffles

// Load 1
%lo = load <8 x double>* %in
%in2 = add %in, 64

// Load 2 - the last load is masked!
%hi = call @llvm.masked.load (<8 x double>* %in2, <8 x i1> < 1,1,1,1,1,1,1,0 >,..)
%even = shufflevector %lo, %hi, <0, 2, 4, 6, 8, 10, 12, 14>
```

C. Create an “indexed load”

```
%1 = call <8 x double> @llvm.indexed.load.v8f64(double* %in, <8 x i32> <0, 2, 4, ..)
```

Vectorizing FP operations

```
float  *A;
for (unsigned i = 0; i < N; i++) {

  if (A[i] != 0)
    C = B / A[i];
   …
}
```

- FP exceptions mode is not supported by LLVM
- The loop is vectorized by LLVM in spite of potential fp-divide-by-zero exception

Correct FP behavior

What should we do in order to be correct?

Use safe values

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

  Mask = (A[i:i+15] != 0)
  SafeDivider = BLEND Mask, A[i:i+15], AllOnes
  C_safe = B / SafeDivider
  C_new = BLEND Mask, C_safe, C
}
```

Masking is designed to solve this problem

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

  Mask = (A[i:i+15] != 0)
  C_new = call @llvm.masked.fdiv(B, A[i:i+15], Mask, C)
}
```

Pros

The FP behavior is correct

Never be broken during optimization

Cons

Intrinsics are hard to optimize

All FP operations may throw exceptions – more than 20 operations should be covered

Status

- Masked Load and Store intrinsics are supported in 3.6
- Gather and Scatter intrinsics are in progress
- Strided Load and Store – the discussion was opened to target ARM interleaved loads and stores
- FP operations are next in line

Summary

Masking is an essential feature of advanced vector architectures including the new AVX-512 Intel® Architecture

Intrinsics with masks allow to vectorize many loops that remain scalar today

We appreciate the support of LLVM community. We want to thank the people who help us define the form of the intrinsics and review the code.

Legal Disclaimer & Optimization Notice

References & Related work

1. S. Timnat, O. Shaham, A. Zaks "Predicate Vectors If You Must".
2. D. Nuzman, I. Rosen, A. Zaks "Auto-vectorization of interleaved data for SIMD", PLDI, 2006
3. "Automatic SIMD Vectorization of SSA-based Control Flow Graphs", PhD Thesis, July 2014, Ralf Karrenberg

Legal Disclaimer & Optimization Notice

- No license to any intellectual property rights is granted by this document.
- Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.
- This document contains information on products, services and/or processes in development. All information provided here is subject to change without notice. Contact your Intel representative to obtain the latest forecast, schedule, specifications and roadmaps.
- 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.
- The products and services described may contain defects or errors known as errata which may cause deviations from published specifications. Current characterized errata are available on request.
- Copies of documents which have an order number and are referenced in this document may be obtained by calling 1-800-548-4725 or by visiting www.intel.com/design/literature.htm.
- Intel, the Intel logo are trademarks of Intel Corporation in the U.S. and/or other countries.
- *Other names and brands may be claimed as the property of others
- © 2015 Intel Corporation.

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.