# Scalable Vectorization in LLVM

ARM

Graham Hunter, Amara Emerson

ARM HPC Compiler Team
3rd November 2016

# Who we are

- ARM HPC compiler and tools team, based in Manchester
- Not your usual ARM suspects, most of us are new the community
- Formed to support research & begin development of compiler extensions to support SVE
- Our mission is now to provide tools and support the software ecosystem for HPC & server on ARM

# Why we're here

- Introduce the SVE architecture and how it affects optimizing compilers
- Present our extensions to LLVM that enable auto-vectorization and codegen support for SVE
- Give you a rationale for the design decisions we made during development
- Get feedback on our approach, and work with the community to integrate our work back into upstream LLVM & Clang

# ARMv8-A
# Scalable Vector Extension

# Introducing the Scalable Vector Extension (SVE)

A vector extension to the ARMv8-A architecture, it's major new features:

- Gather-load & scatter-store
- Per-lane predication
- Predicate-driven loop control and management
- Vector partitioning and SW managed speculation
- Extended integer and floating-point horizontal reductions

SVE is **not** an extension of Advanced SIMD

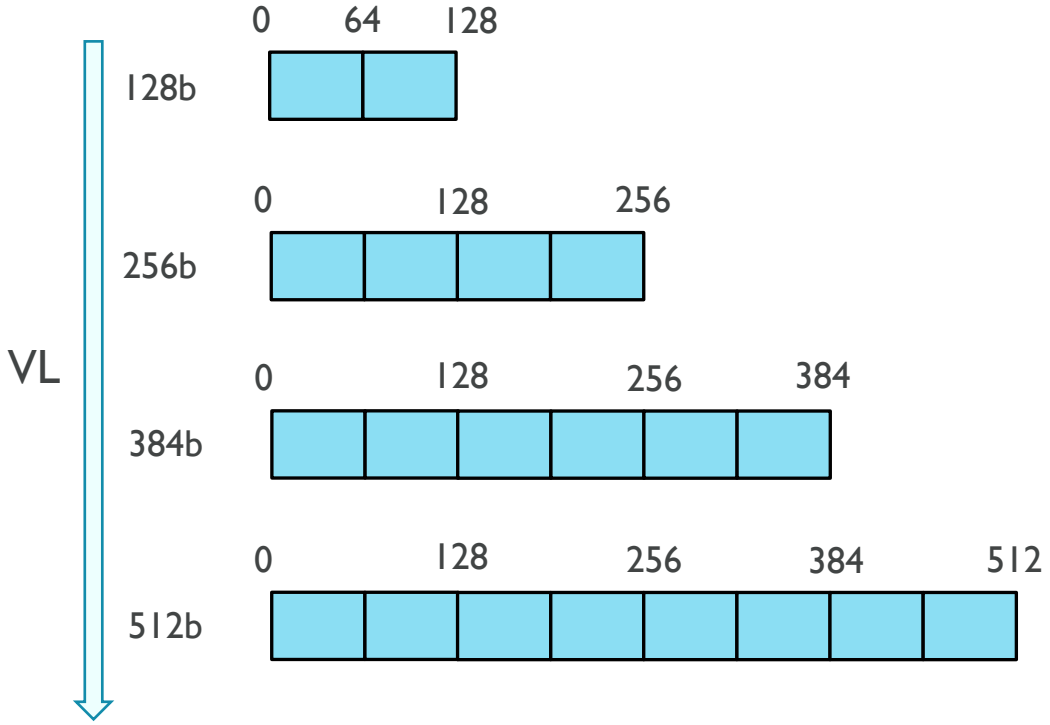
- A separate architectural extension with a new set of A64 instruction encodings
- Focus is HPC scientific workloads, not media/image processing

# What's the vector length?

- There is **no** preferred vector length
  - Vector Length (VL) is hardware choice, from 128 to 2048 bits, in increments of 128
  - Does not need to be a power-of-2
  - *Vector Length Agnostic* programming adjusts dynamically to the available VL
  - No need to recompile, or to rewrite hand-coded SVE assembler or C intrinsics
  - Has extensive implications for loop optimizations

```
add z0.d, p0/m, z0.d, z1.d
```

Merge predicate






ARM

# Vectorization styles

There are two styles of vectorization possible using the SVE architecture.

- Scalar induction variable controlled loops
  - Works in a similar way to the upstream LoopVectorize code, but with some changes in order to work around the unknown vector length
- Predicate controlled loops
  - A new way of controlling loop execution by utilizing the SVE predication features
  - Allows elimination of scalar loop tails

# A simple SVE loop

Let's see how this loop is vectorized using the predicated loop control:

```
void simple(int *restrict a,
            int *restrict b) {
  for (unsigned i=0; i < 22; ++i)
    a[i] = b[i] + i;
}
```

- Loop adds the induction variable i, which can be a vector, to b[i].

# A simple SVE loop

```
simple:
    mov     x8, xzr
    index   z1.s, #0, #1
    cntw    x10
    mov     z0.s, w10
    movz    w9, #22
    whilelo p0.s, xzr, x9
.Lvecbody:
    add     z3.s, z1.s, z0.s
    ld1w    z2.s, p0/z, [x1, x8, lsl #2]
    add     z1.s, p0/m, z1.s, z2.s
    st1w    z1.s, p0, [x0, x8, lsl #2]
    incw    x8
    whilelo p0.s, x8, x9
    mov     z1.d, z3.d
    b.first .Lvecbody
```

A scalar induction variable as normal

# A simple SVE loop

```
simple:
        mov     x8, xzr
        index   z1.s, #0, #1
        cntw    x10
        mov     z0.s, w10
        movz    w9, #22
        whilelo p0.s, xzr, x9
.Lvecbody:
        add     z3.s, z1.s, z0.s
        ld1w    z2.s, p0/z, [x1, x8, lsl #2]
        add     z1.s, p0/m, z1.s, z2.s
        st1w    z1.s, p0, [x0, x8, lsl #2]
        incw    x8
        whilelo p0.s, x8, x9
        mov     z1.d, z3.d
        b.first .Lvecbody
```

Create a counting vector, containing [0, 1, 2, 3, … ]

Create vector induction variable increment

# A simple SVE loop

```
simple:
        mov     x8, xzr
        index   z1.s, #0, #1
        cntw    x10
        mov     z0.s, w10
        movz    w9, #22
        whilelo p0.s, xzr, x9
.Lvecbody:
        add     z3.s, z1.s, z0.s
        ld1w    z2.s, p0/z, [x1, x8, lsl #2]
        add     z1.s, p0/m, z1.s, z2.s
        st1w    z1.s, p0, [x0, x8, lsl #2]
        incw    x8
        whilelo p0.s, x8, x9
        mov     z1.d, z3.d
        b.first .Lvecbody
```

Create entry predicate, up to a maximum of 22 iterations

# A simple SVE loop

```
simple:
        mov     x8, xzr
        index   z1.s, #0, #1
        cntw    x10
        mov     z0.s, w10
        movz    w9, #22
        whilelo p0.s, xzr, x9
.Lvecbody:
        add     z3.s, z1.s, z0.s
        ld1w    z2.s, p0/z, [x1, x8, lsl #2]
        add     z1.s, p0/m, z1.s, z2.s
        st1w    z1.s, p0, [x0, x8, lsl #2]
        incw    x8
        whilelo p0.s, x8, x9
        mov     z1.d, z3.d
        b.first .Lvecbody
```

All load/stores are done using the main loop predicate or a subset if conditional

# A simple SVE loop

```
simple:
    mov       x8, xzr
    index     z1.s, #0, #1
    cntw      x10
    mov       z0.s, w10
    movz      w9, #22
    whilelo   p0.s, xzr, x9
.Lvecbody:
    add       z3.s, z1.s, z0.s
    ld1w      z2.s, p0/z, [x1, x8, lsl #2]
    add       z1.s, p0/m, z1.s, z2.s
    st1w      z1.s, p0, [x0, x8, lsl #2]
    incw      x8
    whilelo   p0.s, x8, x9
    mov       z1.d, z3.d
    b.first   .Lvecbody
```

Increment the scalar induction variable by VL

# A simple SVE loop

```
simple:
        mov     x8, xzr
        index   z1.s, #0, #1
        cntw    x10
        mov     z0.s, w10
        movz    w9, #22
        whilelo p0.s, xzr, x9
.Lvecbody:
        add     z3.s, z1.s, z0.s
        ld1w    z2.s, p0/z, [x1, x8, lsl #2]
        add     z1.s, p0/m, z1.s, z2.s
        st1w    z1.s, p0, [x0, x8, lsl #2]
        incw    x8
        whilelo p0.s, x8, x9
        mov     z1.d, z3.d
        b.first .Lvecbody
```

**Set and read flags**

Generate the next iteration's predicate
Branch back to the header if first lane is active



ARM

# How do we target this architecture in LLVM?

# The goal

```
void loopfunc(int *a, int count) {
 for (int i = 0; i < count; ++i)
  a[i] = i;

}
```

```
loopfunc:
    cntw    x10
    mov     x8, xzr
    index   z1.s, #0, #1
    whilelo p0.s, xzr, x1
    mov     z0.s, w1
.Lvecbody:
    st1w    z1.s, p0, [x0, x8, lsl #2]
    incw    x8
    add     z1.s, z1.s, z0.s
    whilelo p0.s, x8, x1
    b.mi    .Lvecbody
```

# Fixed Length IR

```
void loopfunc(int *a, int count) {
 for (int i = 0; i < count; ++i)
  a[i] = i;

}
```

```
vbody:
%idx = phi i64 [ %next, %vbody ], [ 0, %vph ]
%1 = phi <4 x i32> [ %inc, %vbody ],
   [ <i32 0, i32 1, i32 2, i32 3>, %vph ]
%2 = getelementptr i32, i32* %a, i64 %idx
%3 = add <4 x i32> %1, %splat
%addr = bitcast i32* %3 to <4 x i32>*
store <4 x i32> %1, <4 x i32>* %addr
%next = add i64 %idx, 4
%exit = icmp eq i64 %next, %tc
br i1 %exit, label %exit.block, label %vbody
```

# Induction update

- Need to know exactly how many elements to increment induction variable
- Have to represent a parameterized runtime value in IR

```
vbody:
%idx = phi i64 [ %next, %vbody ], [ 0, %vph ]
%1 = phi <4 x i32> [ %inc, %vbody ],
   [ <i32 0, i32 1, i32 2, i32 3>, %vph ]
%2 = getelementptr i32, i32* %a, i64 %idx
%3 = add <4 x i32> %1, %splat
%addr = bitcast i32* %3 to <4 x i32>*
store <4 x i32> %1, <4 x i32>* %addr
%next = add i64 %idx, 4
%exit = icmp eq i64 %next, %tc
br i1 %exit, label %exit.block, label %vbody
```

# Induction update

- Need to know exactly how many elements to increment induction variable
- Have to represent a parameterized runtime value in IR
- We introduce a new instruction – `elementcount`
- Takes a vector, instructions will be planted to return the actual number of elements at runtime based on the type

```
vbody:
%idx = phi i64 [ %next, %vbody ], [ 0, %vph ]
%1 = phi <4 x i32> [ %inc, %vbody ],
   [ <i32 0, i32 1, i32 2, i32 3>, %vph ]
%2 = getelementptr i32, i32* %a, i64 %idx
%3 = add <4 x i32> %1, %splat
%addr = bitcast i32* %3 to <4 x i32>*
store <4 x i32> %1, <4 x i32>* %addr
%next = add i64 %idx, elementcount <4 x i32> %1
%exit = icmp eq i64 %next, %tc
br i1 %exit, label %exit.block, label %vbody
```

# Scalable “constants”

- Need to supply a value for each element in the vector
- But we don’t know how many elements there are… need a way to generate a sequence of element values

```
vbody:
%idx = phi i64 [ %next, %vbody ], [ 0, %vph ]
%1 = phi <4 x i32> [ %inc, %vbody ],
   [ <i32 0, i32 1, i32 2, i32 3>, %vph ]
%2 = getelementptr i32, i32* %a, i64 %idx
%3 = add <4 x i32> %1, %splat
%addr = bitcast i32* %3 to <4 x i32>*
store <4 x i32> %1, <4 x i32>* %addr
%next = add i64 %idx, elementcount <4 x i32> %1
%exit = icmp eq i64 %next, %tc
br i1 %exit, label %exit.block, label %vbody
```

# Scalable "constants"

- Need to supply a value for each element in the vector
- But we don't know how many elements there are… need a way to generate a sequence of element values
- New IR instruction – `seriesvector`
- Takes two integer inputs, start and step
- Creates a sequence of the form "elt[y] = start + step * y", where y is the index of the element
- Covers constant uses required for vectorization

```
vbody:
%idx = phi i64 [ %next, %vbody ], [ 0, %vph ]
%1 = phi <4 x i32> [ %inc, %vbody ],
   [ seriesvector i32 0, 1 as <4 x i32>, %vph ]
%2 = getelementptr i32, i32* %a, i64 %idx
%3 = add <4 x i32> %1, %splat
%addr = bitcast i32* %3 to <4 x i32>*
store <4 x i32> %1, <4 x i32>* %addr
%next = add i64 %idx, elementcount <4 x i32> %1
%exit = icmp eq i64 %next, %tc
br i1 %exit, label %exit.block, label %vbody
```

# Fully scalable IR

- Need a type to represent scalable vectors

```
vbody:
%idx = phi i64 [ %next, %vbody ], [ 0, %vph ]
%1 = phi <4 x i32> [ %inc, %vbody ],
   [ seriesvector i32 0, 1 as <4 x i32>, %vph ]
%2 = getelementptr i32, i32* %a, i64 %idx
%3 = add <4 x i32> %1, %splat
%addr = bitcast i32* %3 to <4 x i32>*
store <4 x i32> %1, <4 x i32>* %addr
%next = add i64 %idx, elementcount <4 x i32> %1
%exit = icmp eq i64 %next, %tc
br i1 %exit, label %exit.block, label %vbody
```

# Fully scalable IR

- Need a type to represent scalable vectors
- Change VectorType to use `{Min,Scalable}`
- If not scalable, then minimum element count is the total count
- If it is scalable, then the total number of elements is a multiple of the minimum
- Textual format is "`<n x Min x type>`"

```
vbody:
%idx = phi i64 [ %next, %vbody ], [ 0, %vph ]
%1 = phi <n x 4 x i32> [ %inc, %vbody ],
   [ seriesvector i32 0, 1 as <n x 4 x i32>, %vph ]
%2 = getelementptr i32, i32* %a, i64 %idx
%3 = add <n x 4 x i32> %1, %splat
%addr= bitcast i32* %3 to <n x 4 x i32>*
store <n x 4 x i32> %1, <n x 4 x i32>* %addr
%next = add i64 %idx, elementcount <n x 4 x i32> %1
%exit = icmp eq i64 %next, %tc
br i1 %exit, label %exit.block, label %vbody
```

# Scalable types in IR

- Scalable types work in a similar manner to fixed-length types:
  - `<n x 4 x i32>` and `<n x 4 x i8>` have the same number of elements
  - `<n x 4 x i32>` and `<n x 8 x i16>` have the same number of bytes
- Want to minimize code impact, and reuse existing optimizations
- Want to avoid accidentally dropping scalable flag, so aiming for single interface
- `ElementCount` struct has convenience methods and operators to make it easy to use for common operations (like halving/doubling the number of elements)

# Reductions

- Current approach in LLVM is to manually tree reduce with shuffles then pattern match in the backend to plant horizontal instructions
- **Doesn't work for SVE** – needs a loop, and have to hope that no other pass changes it enough to break optimization

```
reduction.block:
%rdx.shuf = shufflevector <4 x float> %lanes,
 <4 x float> undef,
 <4 x i32> <i32 2, i32 3, i32 undef, i32 undef>
%bin.rdx = fadd fast <4 x float> %lanes, %rdx.shuf
%rdx.shuf1= shufflevector <4 x float> %bin.rdx,
 <4 x float> undef,
 <4 x i32> <i32 1, i32 undef, i32 undef, i32 undef>
%bin.rdx1= fadd fast <4 x float> %bin.rdx,
 %rdx.shuf1
%rv = extractelement <4 x float> %bin.rdx1, i32 0
br label %loop.exit
```

# Reductions

- Current approach in LLVM is to manually tree reduce with shuffles then pattern match in the backend to plant horizontal instructions
- **Doesn't work for SVE** – needs a loop, and have to hope that no other pass changes it enough to break optimization
- Use intrinsics instead, obtained via TargetTransformInfo interface
- Reductions for integer and fp arithmetic, logical ops

```
reduction.block:
%rv = call float @llvm.aarch64.sve.addv.nxv4f32(
 <n x 4 x i1> shufflevector (<n x 4 x i1>
   insertelement (<n x 4 x i1> undef, i1 true, i32
                  0),
   <n x 4 x i1> undef,
   <n x 4 x i32> zeroinitializer),
 <n x 4 x float> %lanes)
br label %loop.exit
```

# Scalar tail performance impact

There are a number of benefits of eliminating scalar loop tails:

- Nested loop performance worse if the innermost loop has a small trip count
  - E.g. if an SVE implementation has 256B vectors, up to 255 elements could go straight to scalar tail
- Some short loops that NEON would execute mostly in vector code suddenly do everything in scalar tail
- Larger registers should not reduce vector throughput

# Predicated loop control

# Setting up our predicate

- Add a new phi to track the predicate
- Use `seriesvector` to generate the next iteration's predicate from the next induction value

```
vbody:
%idx = phi i64 [ %next, %vbody ], [ 0, %vph ]
%1 = phi <n x 4 x i32> [ %inc, %vbody ],
   [ seriesvector i32 0, 1 as <n x 4 x i32>, %vph ]
%prd = phi <n x 4 x i1> [ %pentry, % vph ],
   [ %pnext, %vbody ]
          …
store <n x 4 x i32> %1, <n x 4 x i32>* %addr
%next = add i64 %idx, elementcount <n x 4 x i32> %1
%vnext = seriesvector i64 %next, 1 as <n x 4 x i64>
%pnext = icmp ult %vnext, %tc.splat
%exit = icmp eq i64 %next, %tc
br i1 %exit, label %exit.block, label %vbody
```

# Avoiding memory faults

- Add a new phi to track the predicate
- Use `seriesvector` to generate the next iteration's predicate from the next induction value

```
vbody:
%idx = phi i64 [ %next, %vbody ], [ 0, %vph ]
%1 = phi <n x 4 x i32> [ %inc, %vbody ],
   [ seriesvector i32 0, 1 as <n x 4 x i32>, %vph ]
%prd = phi <n x 4 x i1> [ %pentry, % vph ],
   [ %pnext, %vbody ]
          …
store <n x 4 x i32> %1, <n x 4 x i32>* %addr
%next = add i64 %idx, elementcount <n x 4 x i32> %1
%vnext = seriesvector i64 %next, 1 as <n x 4 x i64>
%pnext = icmp ult %vnext, %tc.splat
%exit = icmp eq i64 %next, %tc
br i1 %exit, label %exit.block, label %vbody
```

# Avoiding memory faults

- Add a new phi to track the predicate
- Use `seriesvector` to generate the next iteration's predicate from the next induction value
- All loads and stores are now masked

```
vbody:
%idx = phi i64 [ %next, %vbody ], [ 0, %vph ]
%1 = phi <n x 4 x i32> [ %inc, %vbody ],
   [ seriesvector i32 0, 1 as <n x 4 x i32>, %vph ]
%prd = phi <n x 4 x i1> [ %pentry, % vph ],
   [ %pnext, %vbody ]
            …
call void @llvm.masked.store.nxv4i32(<n x 4 x i32>
%1, <n x 4 x i32>* %addr, i32 4, <n x 4 x i1> %prd)
%next = add i64 %idx, elementcount <n x 4 x i32> %1
%vnext = seriesvector i64 %next, 1 as <n as<nx 4 x
i64>
%pnext = icmp ult %vnext, %tc.splat
%exit = icmp eq i64 %next, %tc
br i1 %exit, label %exit.block, label %vbody
```

# Predicate based loop control

- Add a new phi to track the predicate
- Use `seriesvector` to generate the next iteration's predicate from the next induction value
- All loads and stores are now masked
- How to determine whether to continue?

```
vbody:
%idx = phi i64 [ %next, %vbody ], [ 0, %vph ]
%1 = phi <n x 4 x i32> [ %inc, %vbody ],
   [ seriesvector i32 0, 1 as <n x 4 x i32>, %vph ]
%prd = phi <n x 4 x i1> [ %pentry, % vph ],
   [ %pnext, %vbody ]
          …
call void @llvm.masked.store.nxv4i32(<n x 4 x i32>
%1, <n x 4 x i32>* %addr, i32 4, <n x 4 x i1> %prd)
%next = add i64 %idx, elementcount <n x 4 x i32> %1
%vnext = seriesvector i64 %next, 1 as <n as<nx 4 x
i64>
%pnext = icmp ult %vnext, %tc.splat
%exit = icmp eq i64 %next, %tc
br i1 %exit, label %exit.block, label %vbody
```

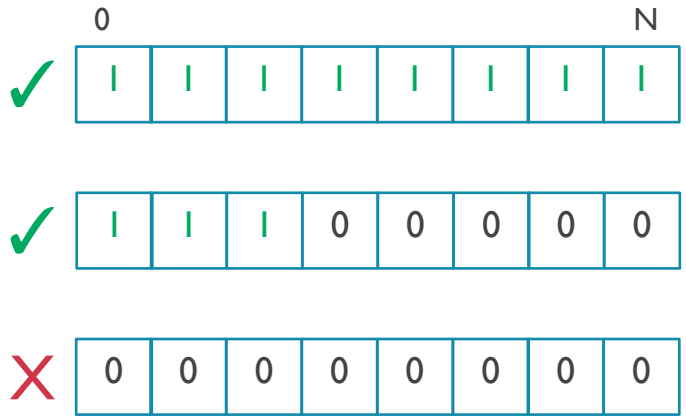
# Predicate based loop control

- Introduced a new instruction: `test`
- Flexible, can ask whether first/last/all/any elements are true or false
- All iterations performed in vector body if entered, scalar body only present if length/aliasing checks are required.

```
vbody:
%idx = phi i64 [ %next, %vbody ], [ 0, %vph ]
%1 = phi <n x 4 x i32> [ %inc, %vbody ],
   [ seriesvector i32 0, 1 as <n x 4 x i32>, %vph ]
%prd = phi <n x 4 x i1> [ %pentry, % vph ],
   [ %pnext, %vbody ]
            …
call void @llvm.masked.store.nxv4i32(<n x 4 x i32>
%1, <n x 4 x i32>* %addr, i32 4, <n x 4 x i1> %prd)
%next = add i64 %idx, elementcount <n x 4 x i32> %1
%vnext = seriesvector i64 %next, 1 as <n as<nx 4 x
i64>
%pnext = icmp ult %vnext, %tc.splat
%exit = test first true <n x 4 x i1> %pnext
br i1 %exit, label %exit.block, label %vbody
```

# Predicate based loop control

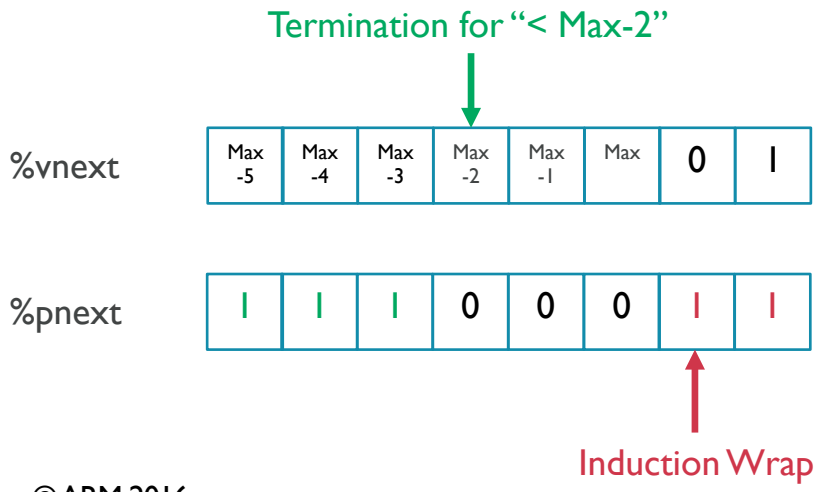
- Introduced a new instruction: `test`
- Flexible, can ask whether first/last/all/any elements are true or false
- All iterations performed in vector body if entered, scalar body only present if length/aliasing checks are required.




```
vbody:
%idx = phi i64 [ %next, %vbody ], [ 0, %vph ]
%1 = phi <n x 4 x i32> [ %inc, %vbody ],
   [ seriesvector i32 0, 1 as <n x 4 x i32>, %vph ]
%prd = phi <n x 4 x i1> [ %pentry, % vph ],
   [ %pnext, %vbody ]
            …
call void @llvm.masked.store.nxv4i32(<n x 4 x i32>
%1, <n x 4 x i32>* %addr, i32 4, <n x 4 x i1> %prd)
%next = add i64 %idx, elementcount <n x 4 x i32> %1
%vnext = seriesvector i64 %next, 1 as <n as<nx 4 x
i64>
%pnext = icmp ult %vnext, %tc.splat
%exit = test first true <n x 4 x i1> %pnext
br i1 %exit, label %exit.block, label %vbody
```

# What about wrapping?

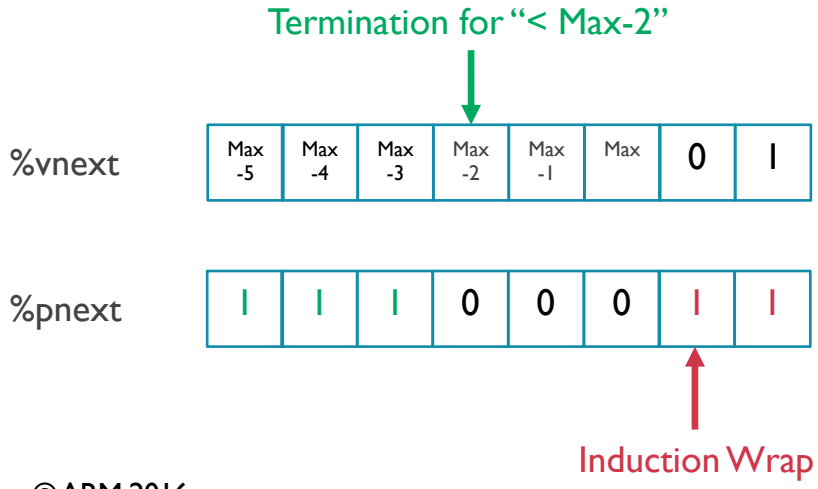
- Range of the vector loop's induction variable is larger than the original loop.
- Depending on the original loop's trip count %next or %vnext might wrap.




```
vbody:
%idx = phi i64 [ %next, %vbody ], [ 0, %vph ]
%1 = phi <n x 4 x i32> [ %inc, %vbody ],
   [ seriesvector i32 0, 1 as <n x 4 x i32>, %vph ]
%prd = phi <n x 4 x i1> [ %pentry, % vph ],
   [ %pnext, %vbody ]
          …
call void @llvm.masked.store.nxv4i32(<n x 4 x i32>
%1, <n x 4 x i32>* %addr, i32 4, <n x 4 x i1> %prd)
%next = add i64 %idx, elementcount <n x 4 x i32> %1
%vnext = seriesvector i64 %next, 1 as <n x 4 x i64>
%pnext = icmp ult <n x 4 x i64>, %vnext, %tc.splat

%exit = test first true <n x 4 x i1> %pnext
br i1 %exit, label %exit.block, label %vbody
```

# What about wrapping?

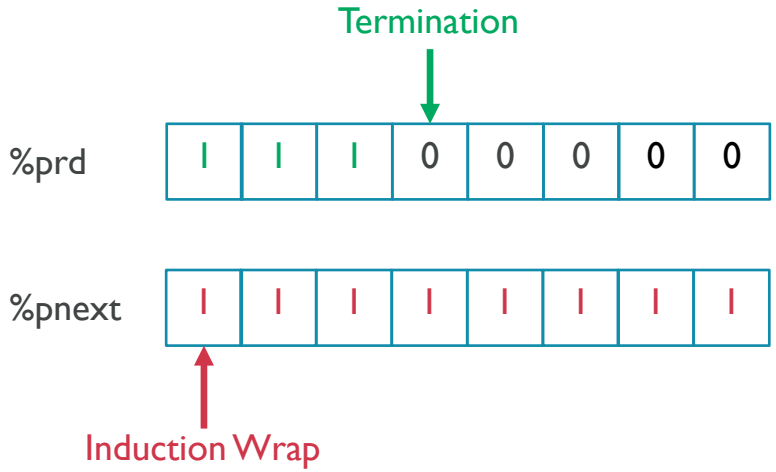
- Range of the vector loop's induction variable is larger than the original loop.
- Depending on the original loop's trip count %next or %vnext might wrap.
- %pnext becomes invalid.
- Next iteration corrupts memory.




```
vbody:
%idx = phi i64 [ %next, %vbody ], [ 0, %vph ]
%1 = phi <n x 4 x i32> [ %inc, %vbody ],
   [ seriesvector i32 0, 1 as <n x 4 x i32>, %vph ]
%prd = phi <n x 4 x i1> [ %pentry, % vph ],
   [ %pnext, %vbody ]
            …
call void @llvm.masked.store.nxv4i32(<n x 4 x i32>
%1, <n x 4 x i32>* %addr, i32 4, <n x 4 x i1> %prd)
%next = add i64 %idx, elementcount <n x 4 x i32> %1
%vnext = seriesvector i64 %next, 1 as <n x 4 x i64>
%pnext = icmp ult <n x 4 x i64>, %vnext, %tc.splat

%exit = test first true <n x 4 x i1> %pnext
br i1 %exit, label %exit.block, label %vbody
```

# What about wrapping?

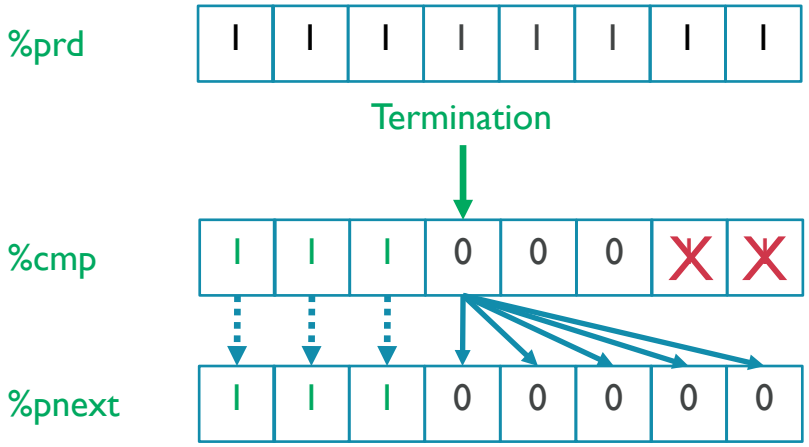
- Must ensure termination point is latched.
- The termination point can occur in either the current iterations predicate (%prd) or the next (%pnext).
- Both predicates must be evaluated to determine "real" active elements.




```
vbody:
%idx = phi i64 [ %next, %vbody ], [ 0, %vph ]
%1 = phi <n x 4 x i32> [ %inc, %vbody ],
   [ seriesvector i32 0, 1 as <n x 4 x i32>, %vph ]
%prd = phi <n x 4 x i1> [ %pentry, % vph ],
   [ %pnext, %vbody ]
            …
call void @llvm.masked.store.nxv4i32(<n x 4 x i32>
%1, <n x 4 x i32>* %addr, i32 4, <n x 4 x i1> %prd)
%next = add i64 %idx, elementcount <n x 4 x i32> %1
%vnext = seriesvector i64 %next, 1 as <n x 4 x i64>
%pnext = icmp ult <n x 4 x i64>, %vnext, %tc.splat

%exit = test first true <n x 4 x i1> %pnext
br i1 %exit, label %exit.block, label %vbody
```

# What about wrapping?

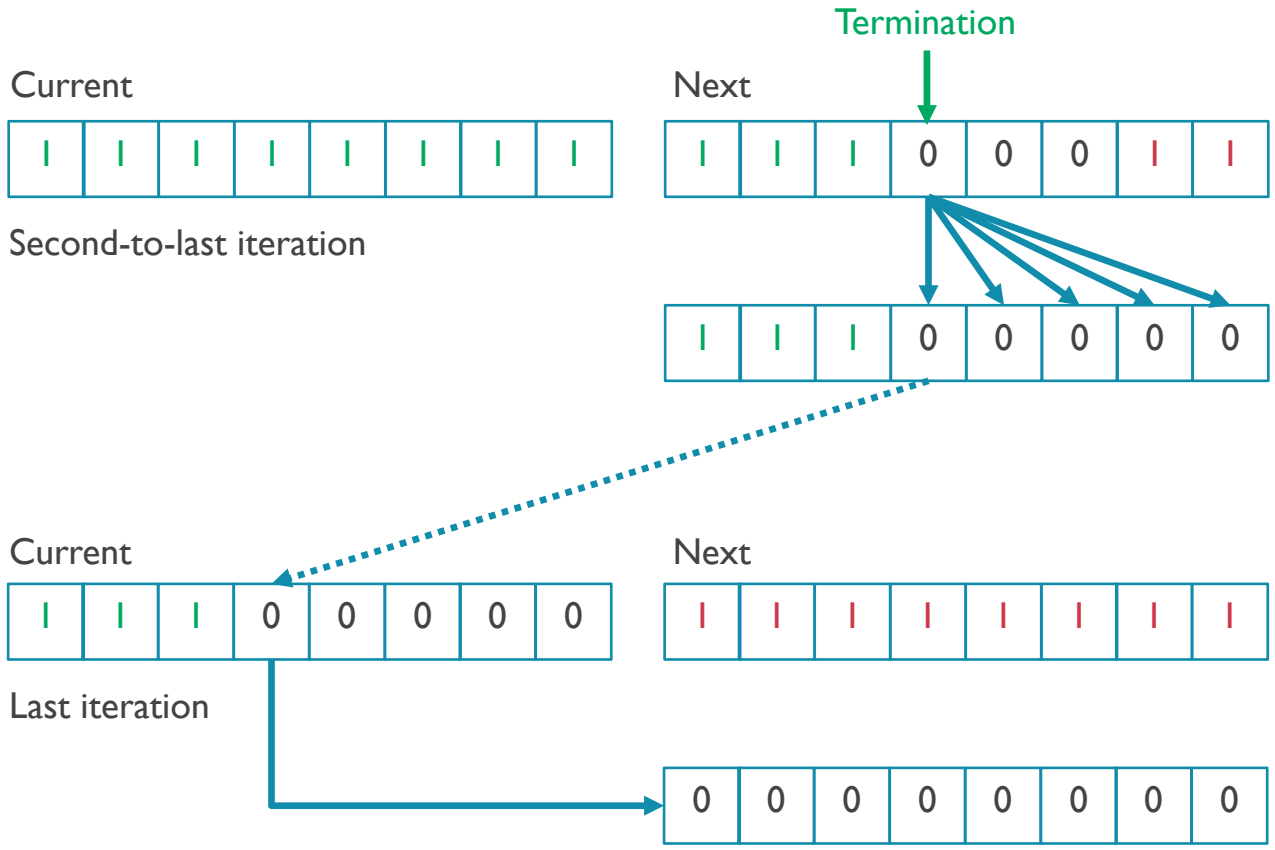
- New instruction – `propff`
- Propagates first false value to all subsequent elements




```
vbody:
%idx = phi i64 [ %next, %vbody ], [ 0, %vph ]
%1 = phi <n x 4 x i32> [ %inc, %vbody ],
   [ seriesvector i32 0, 1 as <n x 4 x i32>, %vph ]
%prd = phi <n x 4 x i1> [ %pentry, % vph ],
   [ %pnext, %vbody ]
          …
call void @llvm.masked.store.nxv4i32(<n x 4 x i32>
%1, <n x 4 x i32>* %addr, i32 4, <n x 4 x i1> %prd)
%next = add i64 %idx, elementcount <n x 4 x i32> %1
%vnext = seriesvector i64 %next, 1 as <n x 4 x i64>
%cmp = icmp ult <n x 4 x i64>, %vnext, %tc.splat
%pnext = propff <n x 4 x i1> %prd, %cmp
%exit = test first true <n x 4 x i1> %pnext
br i1 %exit, label %exit.block, label %vbody
```

# What about wrapping?

- `propff` takes values for current and next predicates
- If the indvar wrapped in the current iteration, next would be all true and we'd never terminate.
- Can be chained when loop unrolling.
- Minimum required capability – more advanced vector partitioning capabilities are desirable






ARM

# Compatibility

Our scalable vectorization with predication changes haven't modified the code generation of existing targets. We achieved this by:

- For fixed-length vector targets, `elementcount` and `seriesvector` IRBuilder methods emit constants as normal.
- LoopVectorize now has a separate `CreateEmptyLoopWithPredication()` method. `test` and `propff` are only planted for targets supporting predicate loop control.
- Existing reduction code still used for fixed-length vectors.

# Code Generation

# SelectionDAG – Types

- New scalable MVTs were created, one for each fixed-length MVT SimpleTy
- `<n x 4 x i32>` becomes `nxv4i32`
- MVT and EVT interfaces also use an `ElementCount` struct
- Types where "Min * EltSize" is 128b map directly to SVE data registers
- Other types must be legalized
  - Larger types will be split into multiple legal values
  - Smaller types require promotion of elements
  - Much the same as fixed-length

# SelectionDAG – ISD Opcodes

- New ISD node for each new instruction
- `BUILD_VECTOR` does not work for scalable types; introduced `SPLAT_VECTOR`
- Added `VECTOR_SHUFFLE_VAR` for non-constant-mask shufflevector
- `INSERT_SUBVECTOR` and `EXTRACT_SUBVECTOR` for scalable types:
  - Index is now treated as a multiple of 'n' for inputs of type <n x m x type>
  - e.g. `nxv2i64 = extract_subvector(nxv4i64, 2)` extracts upper half.

# Vectorizer & mid-end optimizations

# Improved analysis

- Scalar Evolution
  - Propagate no-wrap flags whenever possible
- Alias Analysis
  - Improved support for vectors of pointers
  - Include masked loads/stores (and gather/scatter) in capture tracking
- Loop Access Analysis
  - Better support for strided access
  - Size checking for invariant addresses
- Speculative Bounds Check
  - Used for loops where a uniform load used for loop termination is not hoisted
  - Version loop, speculatively hoists and creates AA/SCEV checks

# Improved loop vectorization

- Vectorization of GEPs, including those just used for pointer arithmetic
- Better memory cost model for gather/scatter ops
  - Improved target info on vector memory op cost
- Vectorization of some memsets within loops
- Support for ordered FP reductions

# Ordered reductions

- "Normal" horizontal reduction instructions process data in a tree: $O(\log_2(N))$
- Requires fastmath for floating point
- Ordered reductions are in-order: O(N)
- Take an additional input value for starting/continuing result
- Reduction occurs inside the vector body

```
OrderedFloatReduction:
        mov     x8, xzr
        whilelo p0.s, xzr, x9
        fmov    s0, wzr
.Lvecbody:
        ld1w    {z1.s}, p0/z, [x0, x8, lsl #2]
        incw    x8
        fadda   s0, p0, s0, z1.s
        whilelo p0.s, x8, x9
        b.mi    .Lvecbody
```

# SLV - Search Loop Vectorization

- Vectorizing loops without known bounds (e.g. strlen)
- Usually 'searching' for a value and terminating
- May have multiple exits (e.g. strnlen), needs to track induction and reduction values for each possible exit
- Speculative loads for safety
- Uses a vector tail instead of performing all work in body

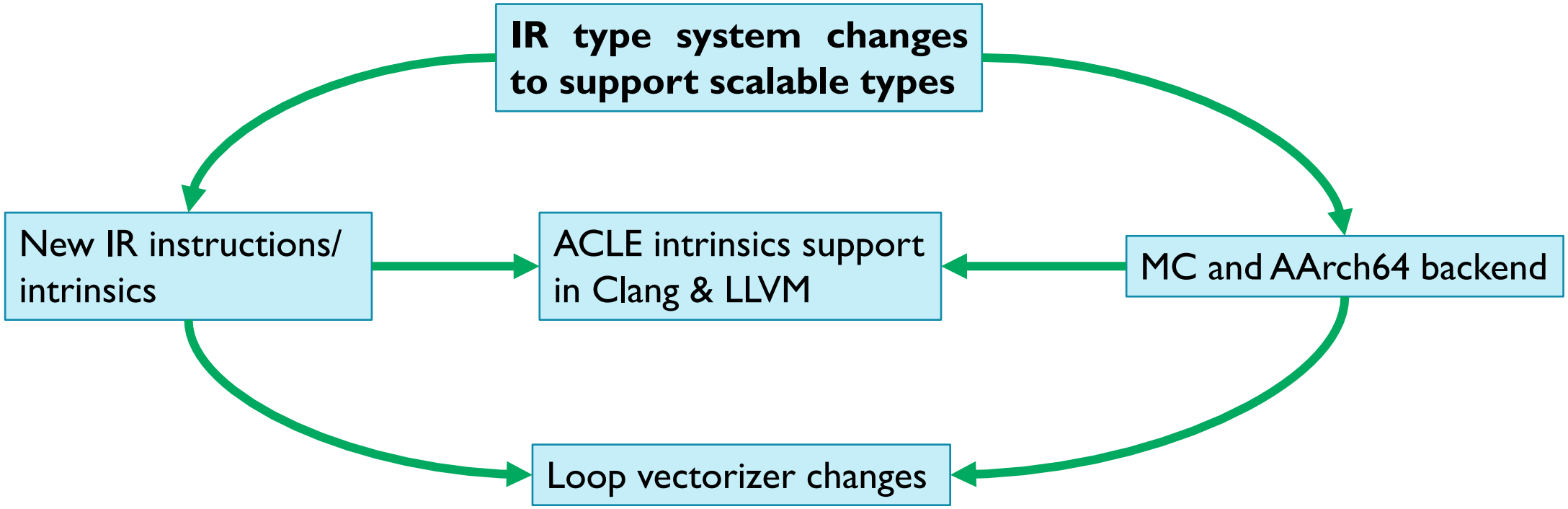
```
int Search(int *a, int count, int val) {
     int i;

     for (i = 0; i <= count; ++i)
          if(a[i] == val)
               break;

     return i;
}
```

# The road ahead…

# Upstreaming plans

**Contributing our changes back to community is a high priority.**

If our design choices are accepted at a high level:



# Access to code

All of our code has been made public as a branch on a public github.

- Available now: https://github.com/ARM-Software/LLVM-SVE
- Serves as reference for upstreaming discussions.

# How we can help each other

There are areas discussed today that need the help of the community to decide on a direction.

- Predicated floating point operations
  - To preserve FP exception semantics after vectorization
- Loop strength reduce
  - Causes problems for code-generation for addressing modes
- Generic reduction intrinsics specification
  - Intrinsics are essential for SVE, should we choose a new canonical form?
- Improved API for separate Vectorization Analysis pass
  - Provide list of barriers to vectorization to earlier passes

# Thank you

## To summarize:

- SVE vectorization enables further extraction of parallelism from sequential loops than previously possible, in an efficient and scalable way.
- A few additional features to the LLVM IR enables the use of this in the loop vectorizer.

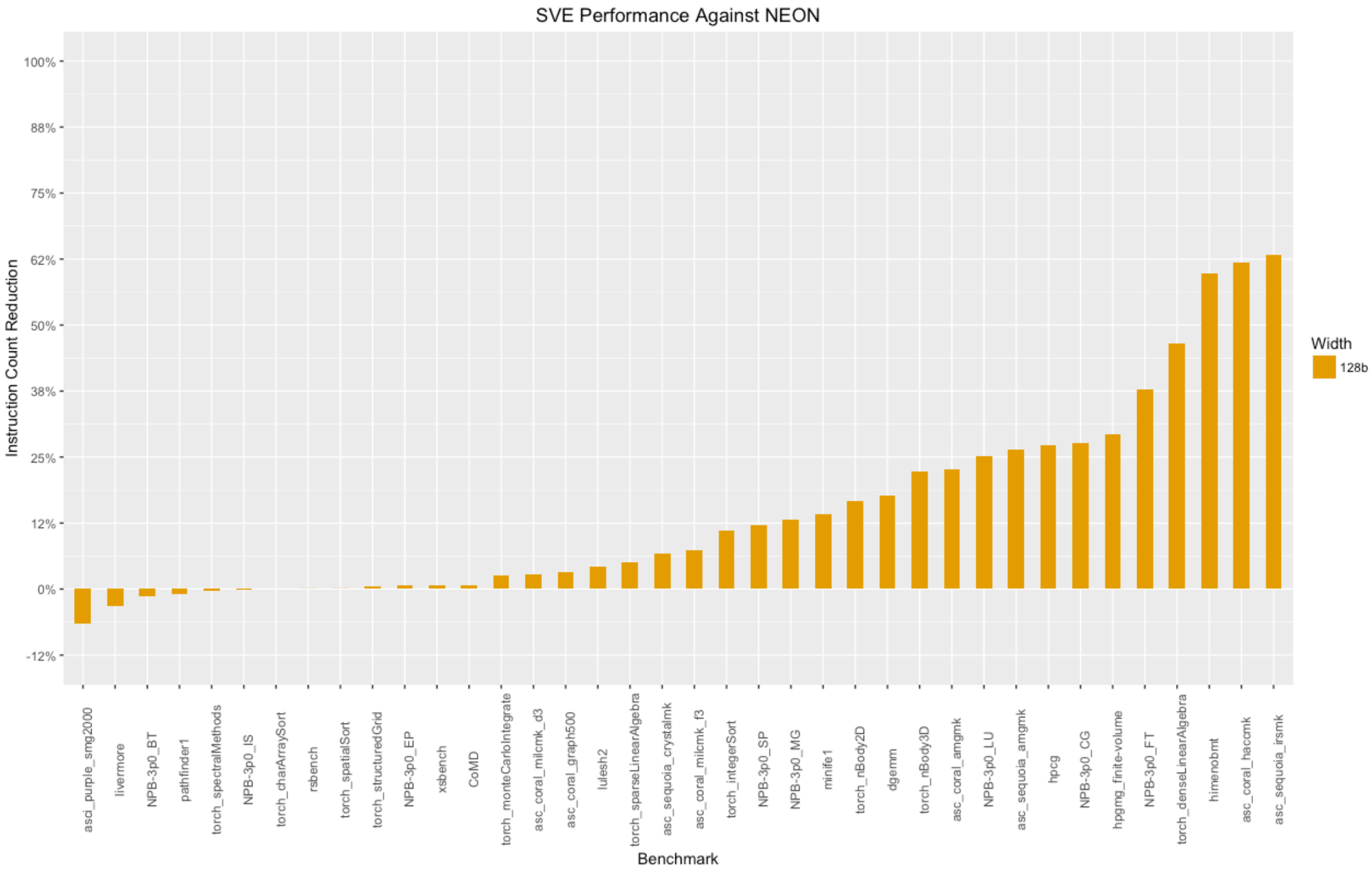
## Additional resources:

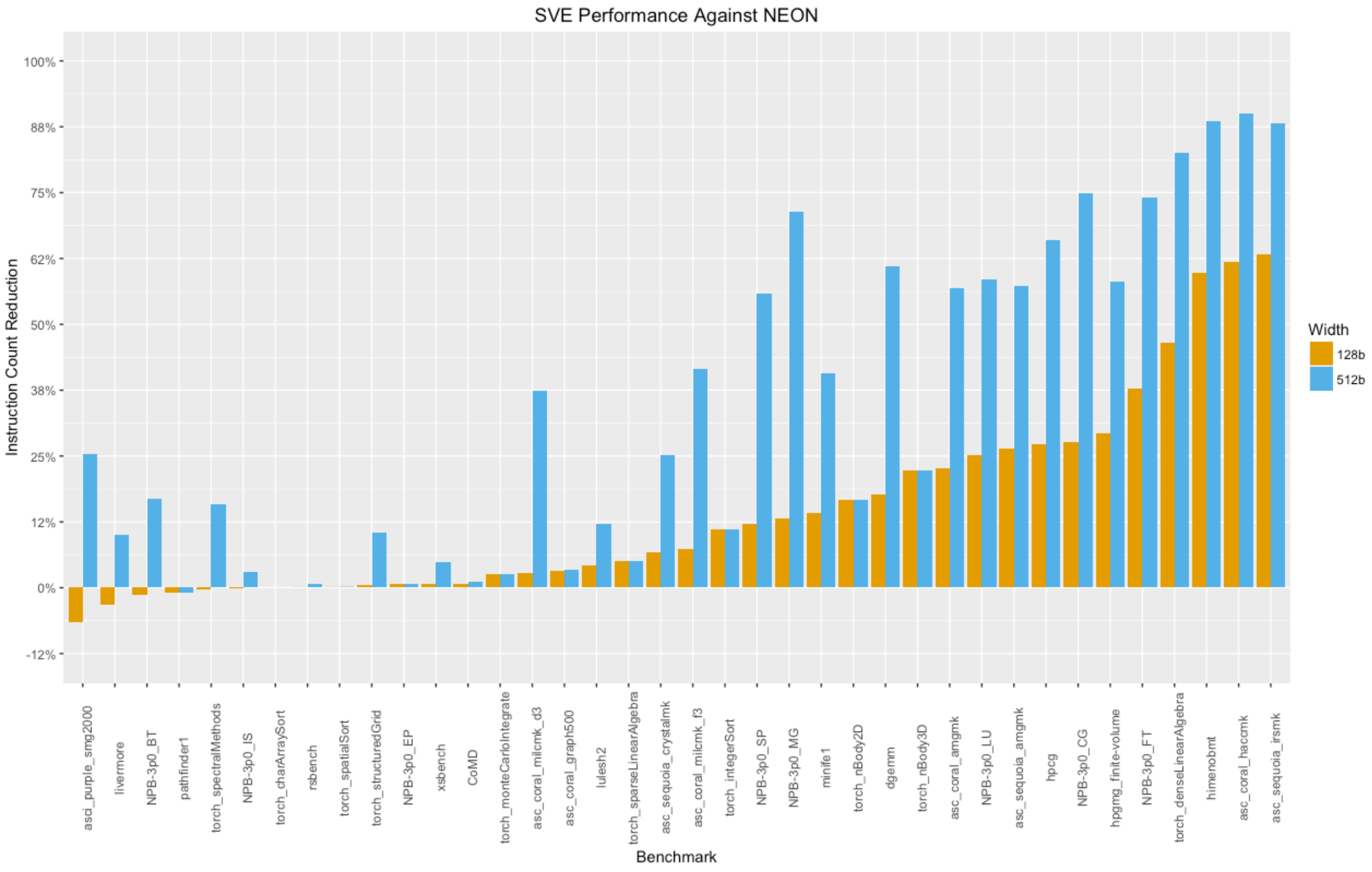
- SVE architecture blog post:
  - https://community.arm.com/groups/processors/blog/2016/08/22/technology-update-the-scalable-vector-extension-sve-for-the-armv8-a-architecture
- SVE & VLA (vector-length-agnostic) whitepaper:
  - http://developer.arm.com/hpc/a-sneak-peek-into-sve-and-vla-programming

ARM

# Backup content

# Performance


SVE Performance Against NEON
Instruction Count Reduction
100%
88%
75%
62%
50%
38%
25%
12%
0%
-12%
asci_purple_smg2000
livermore
NPB-3p0_BT
pathfinder1
torch_spectralMethods
NPB-3p0_IS
torch_charArraySort
rsbench
torch_spatialSort
torch_structuredGrid
NPB-3p0_EP
xsbench
CoMD
torch_monteCarloIntegrate
asc_coral_milcmk_d3
asc_coral_graph500
lulesh2
torch_sparseLinearAlgebra
asc_sequoia_crystalmk
asc_coral_milcmk_f3
torch_integerSort
NPB-3p0_SP
NPB-3p0_MG
minife1
torch_nBody2D
dgemm
torch_nBody3D
asc_coral_amgmk
NPB-3p0_LU
asc_sequoia_amgmk
hpcg
NPB-3p0_CG
hpgmg_finite-volume
NPB-3p0_FT
torch_denseLinearAlgebra
himenobmt
asc_coral_haccmk
asc_sequoia_irsmk
Benchmark
Width
128b


SVE Performance Against NEON
Instruction Count Reduction
100%
88%
75%
62%
50%
38%
25%
12%
0%
-12%
asci_purple_smg2000
livermore
NPB-3p0_BT
pathfinder1
torch_spectralMethods
NPB-3p0_IS
torch_charArraySort
rsbench
torch_spatialSort
torch_structuredGrid
NPB-3p0_EP
xsbench
CoMD
torch_monteCarloIntegrate
asc_coral_milcmk_d3
asc_coral_graph500
lulesh2
torch_sparseLinearAlgebra
asc_sequoia_crystalmk
asc_coral_milcmk_f3
torch_integerSort
NPB-3p0_SP
NPB-3p0_MG
minife1
torch_nBody2D
dgemm
torch_nBody3D
asc_coral_amgmk
NPB-3p0_LU
asc_sequoia_amgmk
hpcg
NPB-3p0_CG
hpgmg_finite-volume
NPB-3p0_FT
torch_denseLinearAlgebra
himenobmt
asc_coral_haccmk
asc_sequoia_irsmk
Benchmark
Width
128b
512b

# Miscellaneous

# New SVE-specific backend passes

- Pass to recognize 'while' sequence
  - Compare induction seriesvector against termination value
  - Propff
  - Test first true
- Addressing(Loop Strength Reduce mitigation)
- Gather/Scatter ➔ structured loads

# Reduction Example

## Source

```
int reduction(int *a, int count) {
  int res = 0;
  for (int i = 0; i < count; ++i) {
    res += a[i];
  }

  return res;
}
```

## IR before vectorization

```
define i32 @reduction(i32* nocapture readonly %a, i32 %count) #0 {
entry:
  %cmp6 = icmp sgt i32 %count, 0
  br i1 %cmp6, label %for.body.preheader, label %for.cond.cleanup
for.body.preheader:
  br label %for.body
for.cond.cleanup.loopexit:
  %add.lcssa = phi i32 [ %add, %for.body ]
  br label %for.cond.cleanup
for.cond.cleanup:
  %res.0.lcssa = phi i32 [ 0, %entry ], [ %add.lcssa,
%for.cond.cleanup.loopexit ]
  ret i32 %res.0.lcssa
for.body:
  %indvars.iv = phi i64 [ %indvars.iv.next, %for.body ], [ 0,
%for.body.preheader ]
  %res.07 = phi i32 [ %add, %for.body ], [ 0, %for.body.preheader ]
  %arrayidx = getelementptr inbounds i32, i32* %a, i64 %indvars.iv
  %0 = load i32, i32* %arrayidx, align 4, !tbaa !1
  %add = add nsw i32 %0, %res.07
  %indvars.iv.next = add nuw nsw i64 %indvars.iv, 1
  %lftr.wideiv = trunc i64 %indvars.iv.next to i32
  %exitcond = icmp eq i32 %lftr.wideiv, %count
  br i1 %exitcond, label %for.cond.cleanup.loopexit, label %for.body
}
```

# Reduction in scalable vectorized IR

## Fully optimized IR prior to codegen

```
define i32 @SimpleReduction(i32* nocapture readonly %a, i32 %count) #0 {
entry:
 %cmp6 = icmp sgt i32 %count, 0
 br i1 %cmp6, label %min.iters.checked, label %for.cond.cleanup

min.iters.checked:                                ; preds = %entry
 %0 = add i32 %count, -1
 %1 = zext i32 %0 to i64
 %2 = add nuw nsw i64 %1, 1
 %wide.end.idx.splatinsert = insertelement <n x 4 x i64> undef, i64 %2, i32 0
 %wide.end.idx.splat = shufflevector <n x 4 x i64> %wide.end.idx.splatinsert, <n x
4 x i64> undef, <n x 4 x i32> zeroinitializer
 %3 = icmp ugt <n x 4 x i64> %wide.end.idx.splat, seriesvector (i64 0, i64 1)
 %predicate.entry = propff <n x 4 x i1> shufflevector (<n x 4 x i1> insertelement
(<n x 4 x i1> undef, i1 true, i32 0), <n x 4 x i1> undef, <n x 4 x i32>
zeroinitializer), %3
 br label %vector.body

vector.body:                                      ; preds = %vector.body,
%min.iters.checked
 %index = phi i64 [ 0, %min.iters.checked ], [ %index.next, %vector.body ]
 %predicate = phi <n x 4 x i1> [ %predicate.entry, %min.iters.checked ], [
%predicate.next, %vector.body ]
 %4 = phi i64 [ 0, %min.iters.checked ], [ %6, %vector.body ]
 %vec.phi = phi <n x 4 x i32> [ zeroinitializer, %min.iters.checked ], [ %10,
%vector.body ]
 %5 = icmp ult i64 %index, 4294967296
 call void @llvm.assume(i1 %5)
 %6 = add i64 %4, elementcount (<n x 4 x i64> undef)
 %7 = getelementptr inbounds i32, i32* %a, i64 %4
 %8 = bitcast i32* %7 to <n x 4 x i32>*
 %wide.masked.load = call <n x 4 x i32> @llvm.masked.load.nxv4i32(<n x 4 x i32>*
%8, i32 4, <n x 4 x i1> %predicate, <n x 4 x i32> undef), !tbaa !1
 %9 = select <n x 4 x i1> %predicate, <n x 4 x i32> %wide.masked.load, <n x 4 x
i32> zeroinitializer
 %10 = add nsw <n x 4 x i32> %vec.phi, %9
 %index.next = add nuw nsw i64 %index, elementcount (<n x 4 x i64> undef)
 %11 = add nuw nsw i64 %index, elementcount (<n x 4 x i64> undef)
 %12 = seriesvector i64 %11, 1 as <n x 4 x i64>
 %13 = icmp ult <n x 4 x i64> %12, %wide.end.idx.splat
 %predicate.next = propff <n x 4 x i1> %predicate, %13
 %14 = test first true <n x 4 x i1> %predicate.next
 br i1 %14, label %vector.body, label %middle.block, !llvm.loop !5

middle.block:                                     ; preds = %vector.body
 %15 = call i64 @llvm.aarch64.sve.uaddv.nxv4i32(<n x 4 x i1> shufflevector (<n x 4
x i1> insertelement (<n x 4 x i1> undef, i1 true, i32 0), <n x 4 x i1> undef, <n x
4 x i32> zeroinitializer), <n x 4 x i32> %10)
 %16 = trunc i64 %15 to i32
 br label %for.cond.cleanup

for.cond.cleanup:                                 ; preds = %middle.block, %entry
 %res.0.lcssa = phi i32 [ 0, %entry ], [ %16, %middle.block ]
 ret i32 %res.0.lcssa
}
```

# Reduction – C to NEON and SVE

## Source

```
int rdx(int *a, int count) {
 int res = 0;
 for (int i = 0; i < count; ++i)
 {
  res += a[i];
 }

 return res;
}
```

## NEON

```
.LBB0_6: // %vector.body
         ldr      q1, [x8], #16
         sub      x11, x11, #4
         add      v0.4s, v1.4s, v0.4s
         cbnz     x11, .LBB0_6
// BB#7: // %middle.block
         addv     s0, v0.4s
         fmov     w8, s0
         cmp      x10, x9
         b.ne     .LBB0_9
         b        .LBB0_11
.LBB0_8: // %for.body.preheader12
         mov      w8, wzr
.LBB0_9: // %for.body.preheader12
         add      x10, x0, x9, lsl #2
         sub      w9, w1, w9
.LBB0_10: // %for.body
         ldr      w11, [x10], #4
         sub      w9, w9, #1
         add      w8, w11, w8
         cbnz     w9, .LBB0_10
.LBB0_11: // %for.cond.cleanup
         mov      w0, w8
         ret
```

## SVE

```
.LBB0_2: // %vector.body
         ld1w     {z1.s}, p1/z, [x0, x8, lsl
#2]
         incw     x8
         add      z0.s, p1/m, z0.s, z1.s
         whilelo  p1.s, x8, x9
         b.mi     .LBB0_2
// BB#3: // %middle.block
         uaddv    d0, p0, z0.s
         fmov     w0, s0
         ret
.LBB0_4:
         mov      w0, wzr
         ret
```

# SLV - Search Loop Vectorization

- Currently implemented in a separate pass
- Applicable to older vector ISAs (e.g. NEON), though cost model will be different
- Started splitting out analysis into a separate pass so that better decisions can be made by prior passes
- Future work
  - Support more cases (e.g. return from inside if statement)
  - Multiple backedges

# Speculative bounds checking pass

- Loop termination condition requires a load inside loop
- Load is from invariant address, but wasn't hoisted
- Speculatively hoist the first load and perform aliasing/SCEV checks based on that value.
- Feels very similar to LV-LICM
- Would need to expand LoopVersioning API to merge together

# BOSCC pass

- Branch on Superword Condition Code (Shin et al)
- Reintroduce branches to vectorized loops
- Good if large amount of code is only executed rarely
- FDO extremely useful for using this transformation with larger vector sizes
- May well be useful for other targets

# SVE Addressing Modes - Loop Strength Reduce

- In contrast to SVE gather/scatter instructions, the addressing modes for contiguous loads/stores are only available in scaled variants
- Scalar + Imm
  - Scaled by #elements * sizeof(vector-lane), e.g.

```
ld1w {z0.s}, [x12, #1 mul vl]
```

    for a 128bit vector packing 4 32bit elements,
    this loads from byte address `x12 + 16`
- Scalar + Scalar
  - Scaled by sizeof(vector-lane), e.g.

```
ld1w {z0.s}, [x12, x8, #lsl 2]
```

- Unscaled addresses need to be explicitly materialized and used with Scalar + Imm addressing mode with immediate #0 mul vl.

# SVE Addressing Modes - Loop Strength Reduce

- LLVM optimizes for:
  - Constant offsets being cheap to implement as immediates
    - Cost model for Loop Strength Reduce (LSR)
    - SeparateConstantOffsetsFromGEP pass
  - Unscaled addressing modes – pushing all scaling down to the pointer increment itself

- SVE supports VL-scaled immediates (useful for e.g. unrolled loops), but for non-VL-scaled immediates, they are better folded into the index or base address

- SVE has no unscaled (scalar) reg + reg addressing modes for contiguous vector loads/stores, so we need to push the scaling back to its uses

# SVE Addressing Modes - Loop Strength Reduce

We have a pass that reverts LSR

```
vector.body:
  %idx = phi i64 [ %idx.n, %vector.body ],
                 [ 8,      %vector.ph   ]
  %1 = bitcast double* %0 to i8*
  %uglygep580 = getelementptr i8, i8* %1,
                                  i64 %idx
  ...
  %idx.n = add i64 %idx,
             mul (i64 elementcount (
                 <n x 2 x i64> undef), i64 8)

  br i1 %cond, label vector.body
```

```
vector.body:
  %idx = phi i64 [ %idx.n, %vector.body ],
                 [ 1,      %vector.ph   ]
  %idx.new = mul i64 %idx, 8
  %1 = bitcast double* %0 to i8*
  %uglygep580 = getelementptr i8, i8* %1,
                                  i64 %idx.new
  ...
  %idx.n = add i64 %idx,
       i64 elementcount (<n x 2 x i64> undef)

  br i1 %cond, label vector.body
```

# SVE Stack Support

- How to lay out the stack when the size of a register is only known at runtime?
- Related discussion on mailing list for variable sized registers and stack support
  - "RFC: Implement variable-sized register classes"
- SVE has instructions to increment a scalar by (a multiple of) the VL that can be used to allocate stack space

```
addvl sp, sp, #-8
  ; allocates 8 vector registers on the stack
```

- SVE spill/fill instructions have VL-scaled immediate addressing mode, e.g.

```
str z0, sp, #7, mul vl
  ; spills vector register z0 to stack at (sp + 7*sizeof(z0))
```

- Because we need special instructions and addressing modes to allocate, deallocate and access SVE objects on the stack, we do not want to mix unscaled and VL-scaled objects

# SVE Stack Support

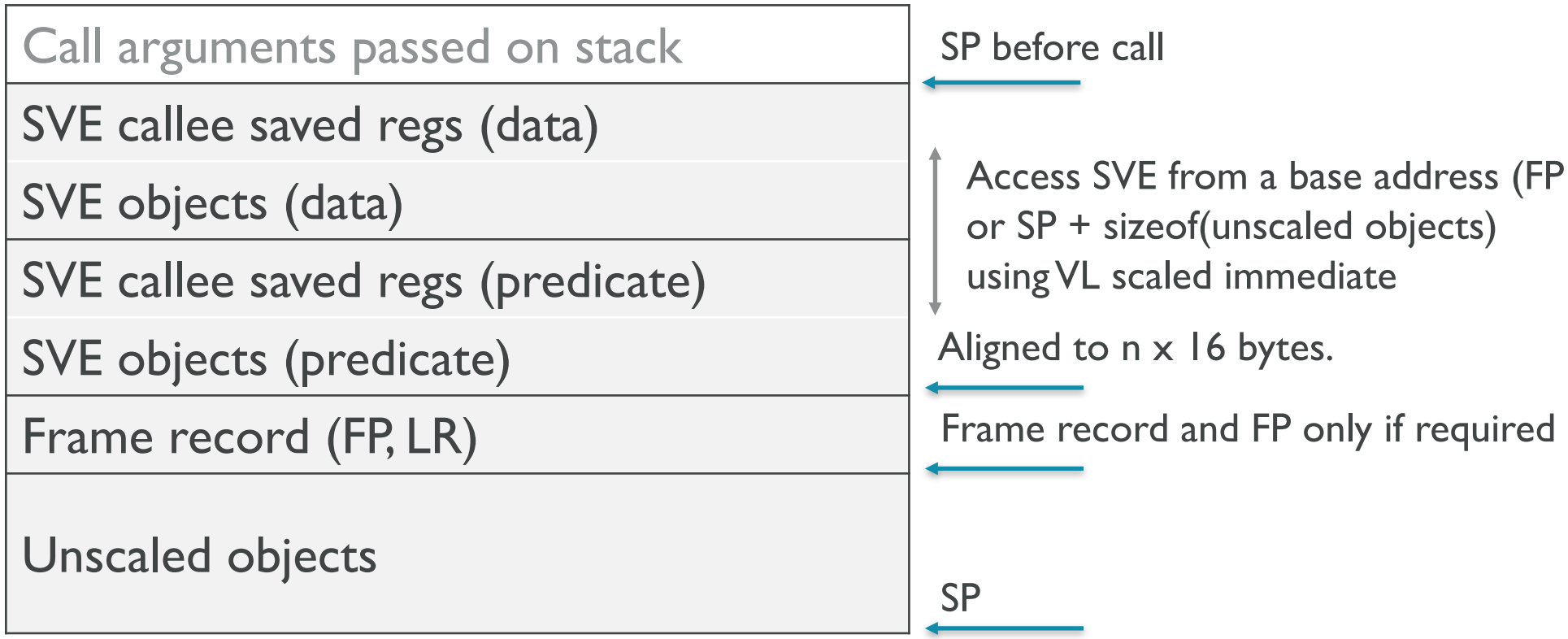
- We separate SVE objects (predicate and data) from objects with known size by registering them to a separate region on the stack
- A StackRegion handles its layout, allocation and deallocation separately from objects not associated with any StackRegion (the default)
- We added a new CodeGen pass called 'InitStackRegions', which queries the target if it wants to register custom regions, and iterates over all object allocations and CSRs to assign them to custom stack regions
- Passes like 'StackSlotColoring' and 'LocalStackSlotAllocation' should make sure not to combine objects of different StackRegions

# SVE Stack Support

- The scaling of an object is determined by the region it is associated with
- AArch64FrameLowering is responsible for laying out the predicate and vector objects by aligning the data and assigning SP-relative offsets
- SVE has a region for its predicate vectors and a region for its data vectors
- Since predicate vectors are much smaller than data vectors (‘n times 2 bytes’ vs to ‘n times 16 bytes’), we allocate these as close as possible to the Stack Pointer and align these to the data region, so we can easily ‘jump’ 8 predicate registers to access the first SVE data vector.

# SVE Stack Support

- AArch64 Stack Frame with SVE



# SVE Stack Support

- Stack Regions may be useful for other targets as well
  - No need to define multiple register classes to implement ‘variable sized registers’ for the purpose of spilling/filling
- Existing targets are not affected and layout of existing types and register classes is unchanged
- We currently have InitStackRegions as a CodeGen pass - This means that we cannot make the distinction between different regions during instruction lowering (we may want to create stack objects to legalize e.g. INSERT_VECTOR_ELT)