# Optimizing code for scalable vector architectures

Sander de Smalen

# Scalable Vectors

- Allows the same binary to run on CPUs with different vector-lengths.
  - Performance scales with number of lanes implemented by the µarch.
- Two targets in LLVM implement support for scalable vectors
- LLVM supports increasing levels of vector-length agnostic auto-vectorization.
- The AArch64 target additionally supports:
  - SVE C/C++ types and intrinsics (Arm C Language Extensions for SVE)
  - Use of vector-length specific types for SVE, enabled by `-msve-vector-bits=<width>`

# LLVM IR Type

Scalable vector type:

<vscale x N x eltty>

vscale is:

- An integer value unknown at compile-time.
- Constant for all vectors in the program.
- Greater than 0 and in the range specified by vscale_range(min[,max])

<vscale x N x eltty> is a more expressive type than <vscale x eltty> since it allows to represent that one vector is wider than another.

The types cannot be used in arrays or as global values, and structs of scalable vectors cannot be stored.

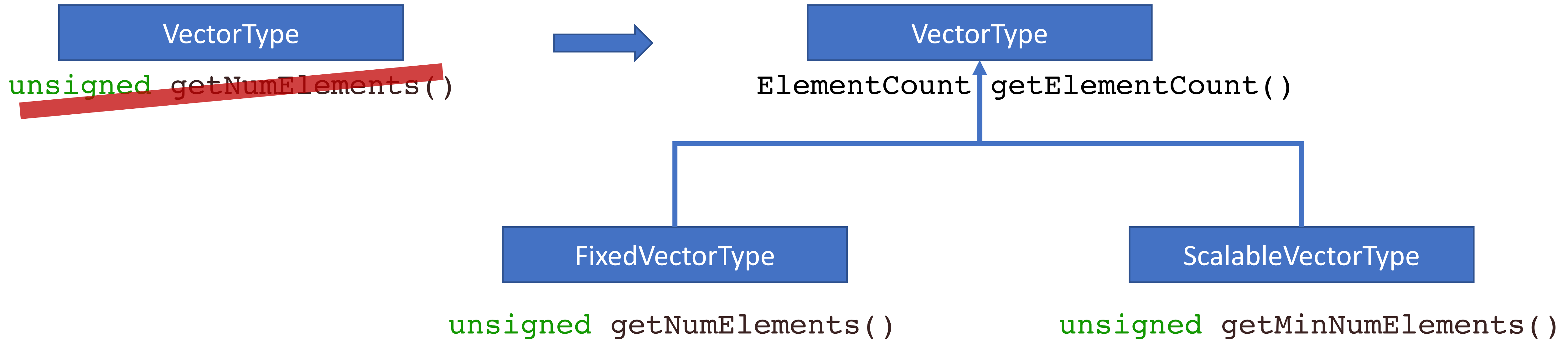
New Function Attribute:

vscale_range(min[,max])

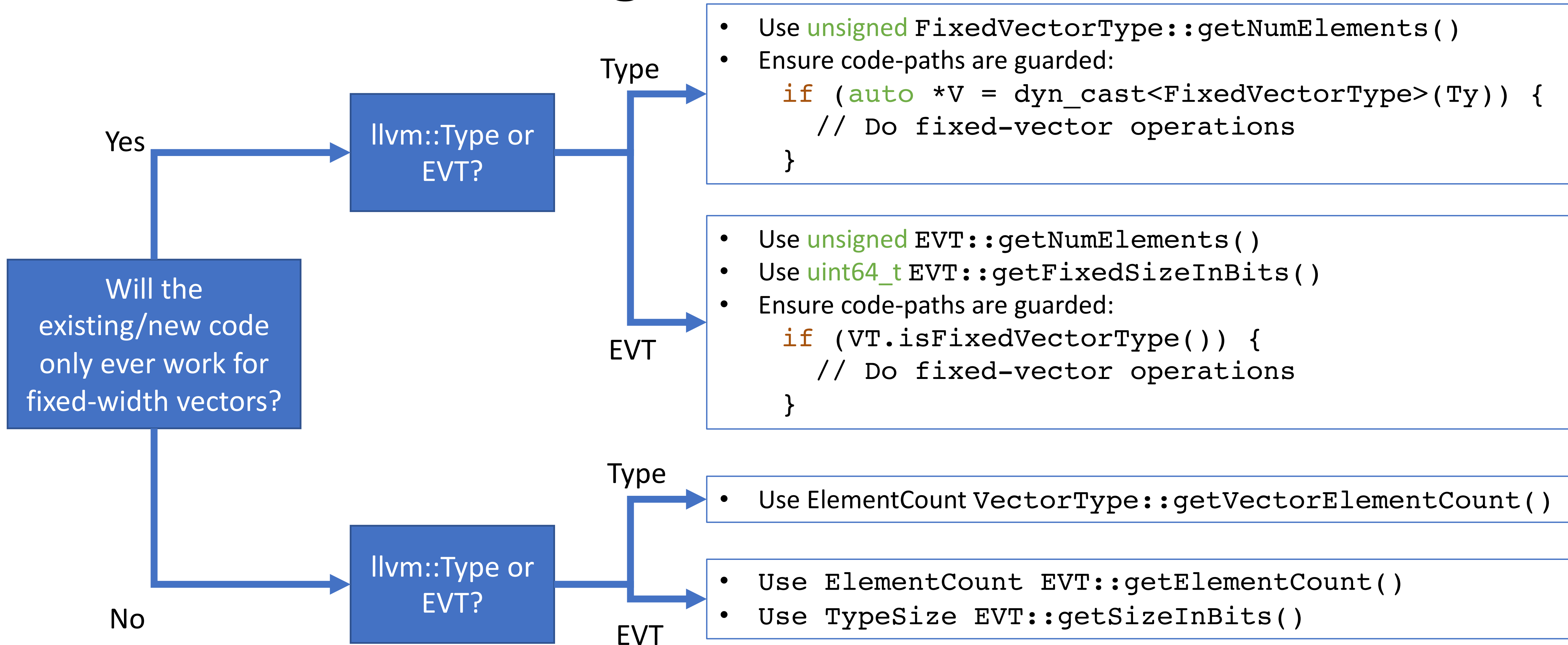
- LLVM guarantees that the compiled program can run correctly on a machine with vscale in the range: min <= vscale <= max
- If max=0, the maximum is unbounded.
- If vscale_range is not specified, it is entirely unbounded (1, infinity)

# Changes in LLVM

- `sizeof(<vscale x N x eltty>)` is not fully known at compile-time.
  - LLVM previously represented sizes as `uint64_t` ➔ now `class TypeSize`.
  - LLVM previously represented element counts as `unsigned` ➔ now `struct ElementCount`.
- These structures represent a value that is either fixed or scalable.
- `VectorType` is split up into `FixedVectorType` and `ScalableVectorType`



# Guidance for Using New Interfaces



# Guidance for Using New Interfaces

- `Typesize` and `ElementCount` have overloaded operators for addition, subtraction, multiplication and division.
- Work on `ElementCount` and `TypeSize` directly.
  - Avoid querying "KnownMin” values where possible.

- Please be so kind to consider writing a test for scalable vectors as well ☺

# Enabling Scalable Loop Vectorization

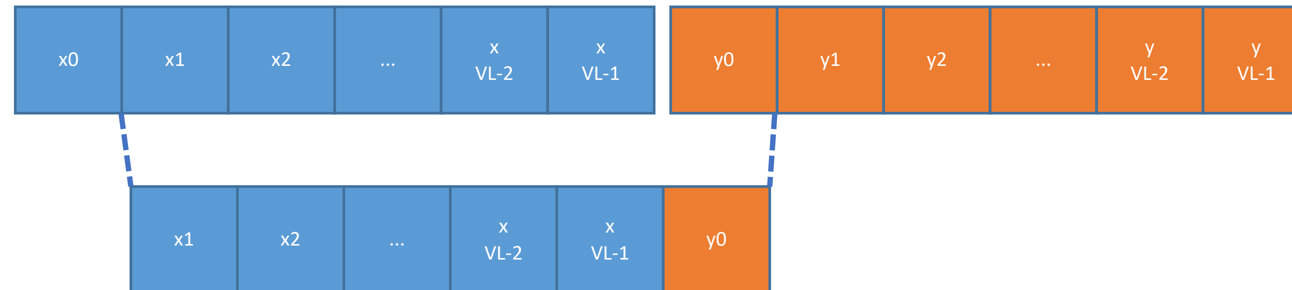
# Enabling Scalable Loop Vectorization

- Experimental scalable auto-vectorization enabled under a flag:
  `-scalable-vectorization=<off|on|preferred>`
  - Guarded by buildbots (building and running on AArch64 SVE hardware)
- Issues we had to solve:
  - Representing scalable vector shuffles
  - Representing induction variables as a vector
  - Cost-modelling for unknown number of lanes
  - Extending the LoopVectorizer to use scalable VFs

# Representing Scalable Vector Shuffles

- Splats can use `shufflevector`
  - `zeroinitializer` is the only scalable vector Constant that LLVM can represent, which means we can use `shufflevector` similar to fixed-width vectors.
- Reverse (new intrinsic `@llvm.experimental.vector.reverse`)
- Splice (new intrinsic `@llvm.experimental.vector.splice`)

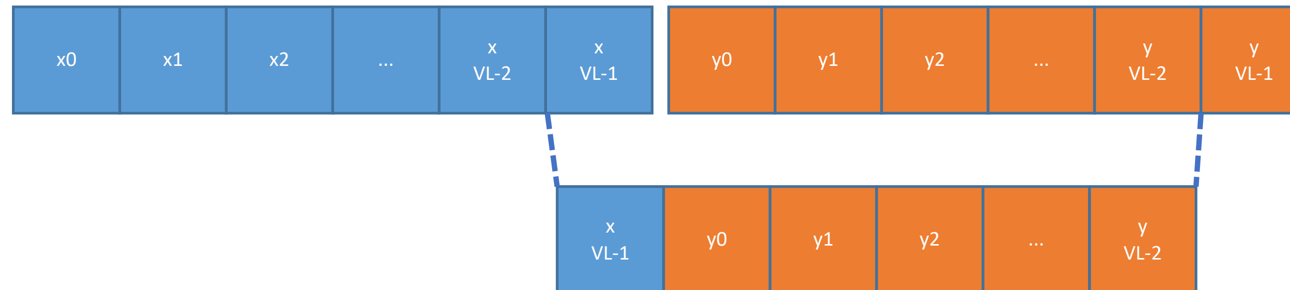
```
@llvm.experimental.vector.splice(<vscale x N x eltty> %x,
                                 <vscale x N x eltty> %y, i32 1)
```



# Representing Scalable Vector Shuffles

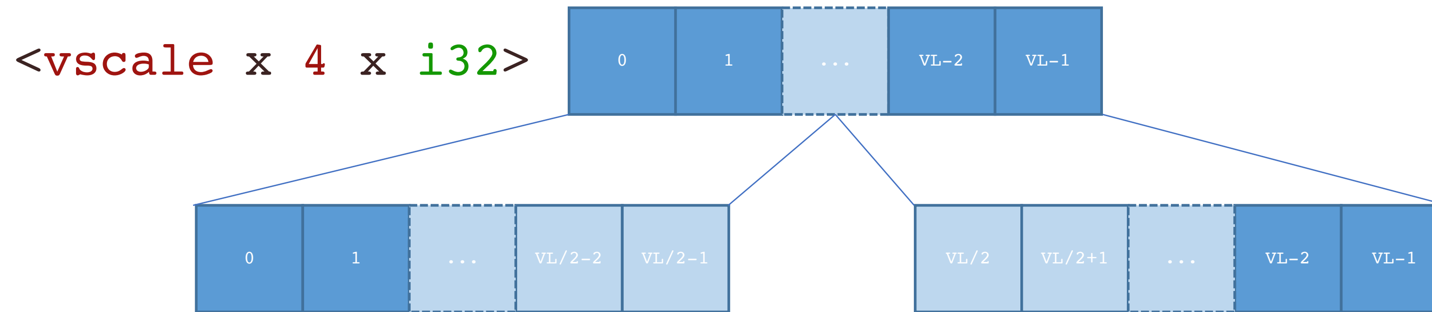
- Splats can use `shufflevector`
  - `zeroinitializer` is the only scalable vector Constant that LLVM can represent, which means we can use `shufflevector` similar to fixed-width vectors.
- Reverse (new intrinsic `@llvm.experimental.vector.reverse`)
- Splice (new intrinsic `@llvm.experimental.vector.splice`)

```
@llvm.experimental.vector.splice(<vscale x N x eltty> %x,
                                 <vscale x N x eltty> %y, i32 -1)
```



# Subvector Extract & Insert

- Changed definition of `ISD::INSERT/EXTRACT_SUBVECTOR` to implicitly scale index by `vscale`.
- Made the nodes available as LLVM IR intrinsics.
  - `subvecty @llvm.experimental.vector.extract(vecty %in, i64 %idx)`
  - `vecty @llvm.experimental.vector.insert(vecty %in, subvecty %sub, i64 %idx)`




Multiplied by `vscale`

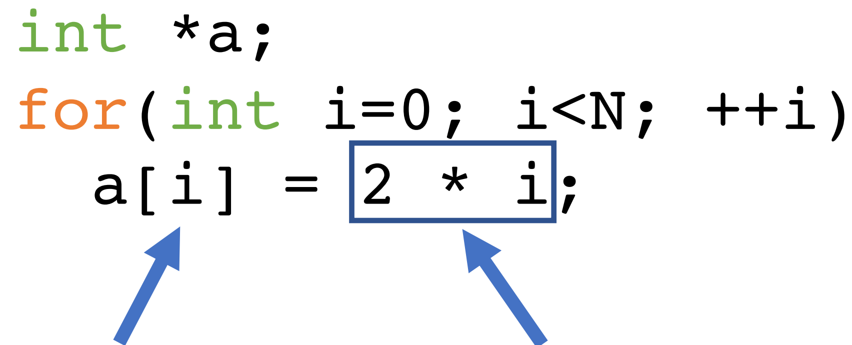
```
<vscale x 2 x i32>
@llvm.vector.extract(<vscale x 4 x i32>, i64 0)
```

Extract low half

```
<vscale x 2 x i32>
@llvm.vector.extract(<vscale x 4 x i32>, i64 2)
```

Extract top half

# Induction Variables as a Vector




`i` is sequential and used in address arithmetic for a contiguous store. Expression can remain scalar.

`2*i` is expanded and stored. This requires expanding to an actual vector `<0, 2, 4, .., 2*(VL-1)>`

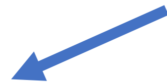
```
<vscale x 4 x i32> @llvm.experimental.stepvector.nxv4i32()
```

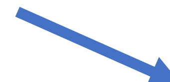
Returns a vector <0, 1, 2, ..., VL-1> of requested integer vector type. Standard `mul` and `add` instructions can be used to get a vector with different stride and start offset.

# Cost-modelling for unknown number of lanes

- LLVM now uses `struct InstructionCost` for cost types, instead of `unsigned` and `int`.
- For costs of individual operations, the number of lanes doesn't matter, except when the operation is ordered (such as strict FP reductions)
- The overall cost gets divided by the number of lanes.
  - Use –mtune=<cpu> to tune for specific `vscale` for that CPU.
  - Code remains compatible with any `vscale` in the specified `vscale_range`.

```
-march=armv8.x-a+sve -mtune=neoverse-v1
```





Compatible with any armv8.x CPU that has SVE

Optimized cost-model for vscale = 2

# Loop Vectorization Legality

- There is no scalarization fallback for scalable vectors, so avoid vectorization if scalarization is required.
- Is the dependence distance within maximum vscale range?

```
int *a;
for(int i=0; i<N; ++i)
  a[i] = a[i+32] + 42;
```

Dependence distance is 128bytes.

For `vscale_range(1, 16)`, the loop can be vectorized with at most `VF=vscale x 2` elements.

⇔ 16 x 2 x 4bytes = 128 bytes

# Loop Vectorization Example

```
int * restrict a, *b;
  int Val, N;
  for(int i=0; i<N; ++i)
    a[i] = b[i] + Val;
```

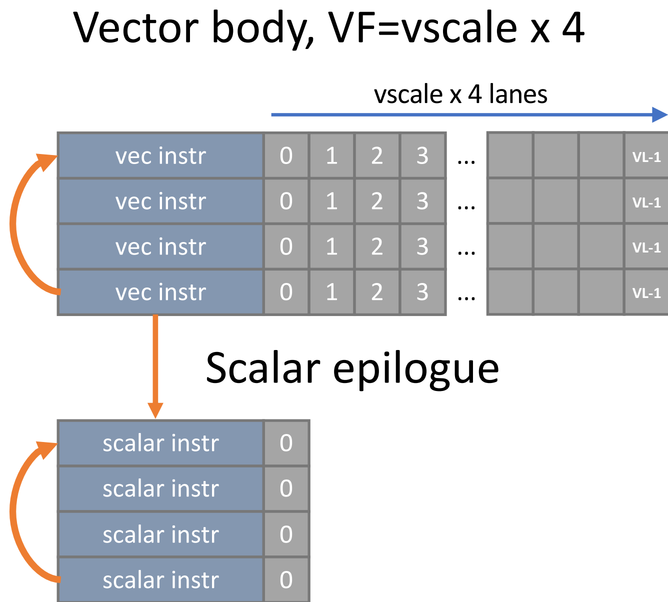
```
vector.ph:                                        ; preds = %for.body.preheader
  %2 = call i64 @llvm.vscale.i64()
  %3 = shl nuw nsw i64 %2, 2
  %n.mod.vf = urem i64 %wide.trip.count, %3
  %n.vec = sub nsw i64 %wide.trip.count, %n.mod.vf
  %broadcast.splatinsert = insertelement <vscale x 4 x i32> poison, i32 %Val, i32 0
  %broadcast.splat =
        shufflevector <vscale x 4 x i32> %broadcast.splatinsert,
                      <vscale x 4 x i32> poison, <vscale x 4 x i32> zeroinitializer
  %4 = call i64 @llvm.vscale.i64()
  %5 = shl nuw nsw i64 %4, 2
  br label %vector.body

vector.body:                                      ; preds = %vector.body, %vector.ph
  %index = phi i64 [ 0, %vector.ph ], [ %index.next, %vector.body ]
  %6 = getelementptr inbounds i32, i32* %b, i64 %index
  %7 = bitcast i32* %6 to <vscale x 4 x i32>*
  %wide.load = load <vscale x 4 x i32>, <vscale x 4 x i32>* %7, align 4, !tbaa !8
  %8 = add nsw <vscale x 4 x i32> %wide.load, %broadcast.splat
  %9 = getelementptr inbounds i32, i32* %a, i64 %index
  %10 = bitcast i32* %9 to <vscale x 4 x i32>*
  store <vscale x 4 x i32> %8, <vscale x 4 x i32>* %10, align 4, !tbaa !8
  %index.next = add nuw i64 %index, %5
  %11 = icmp eq i64 %index.next, %n.vec
  br i1 %11, label %middle.block, label %vector.body, !llvm.loop !12
```

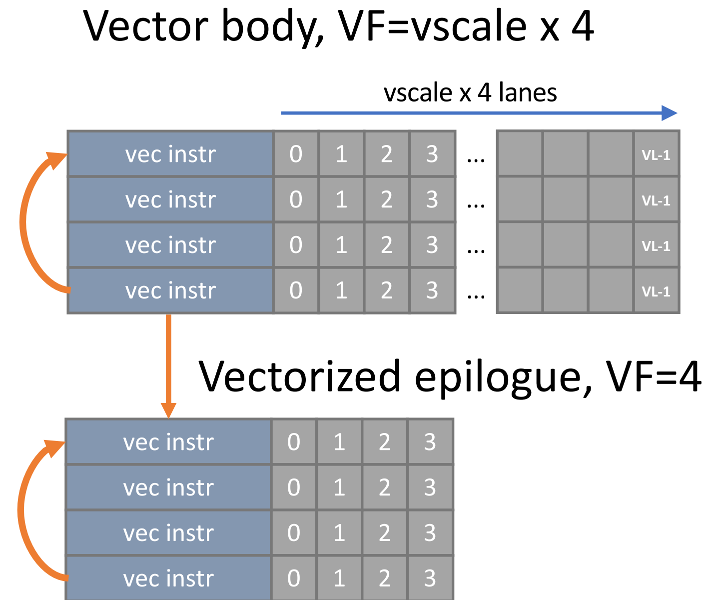
Increments loop induction var by vscale x 4 elements (VF=vscale x 4)

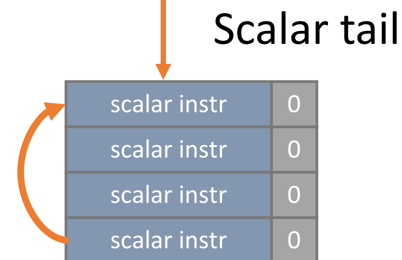
# Vectorization Styles

1. Unpredicated vector body + scalar epilogue
2. Unpredicated vector body + vectorized epilogue
3. Predicated vector body




1




2

Predicated vector body, VF=vscale x 4
No epilogue

vscale x 4 lanes

loop predicate 1 1 1 1 ... 1 0 0 0

while i < N

vec instr 0 1 2 3 ... VL-1

vec instr 0 1 2 3 ... VL-1

vec instr 0 1 2 3 ... VL-1

vec instr 0 1 2 3 ... VL-1

3

# Generating code for fixed-width vectors for SVE/SVE2

# Fixed-width Code Generation for SVE/SVE2

- If the vscale_range minimum allows, use SVE/SVE2 for wide fixed-width vectors.
- We can use SVE/SVE2 instructions for fixed-width vectors as well, without the need to re-implement all TableGen patterns.

```
v8i32 [v0|v1|v2|v3|v4|v5|v6|v7] = LOAD i64 %ptr
```

```
         nxv4i1 [1|1|1|1|1|1|1|1|0|...|0|0] %mask = PTRUE(8 elements)
    nxv4i32 [v0|v1|v2|v3|v4|v5|v6|v7|0|...|0|0] %load = MLOAD i64%ptr, nxv4i1 %mask
v8i32 [v0|v1|v2|v3|v4|v5|v6|v7] = EXTRACT_SUBVECTOR(nxv4i32 %load, i64 0)
```

```
STORE i64 %ptr, v8i32 %val [v0|v1|v2|v3|v4|v5|v6|v7]
```

```
        nxv4i32 [v0|v1|v2|v3|v4|v5|v6|v7|?|...|?|?] %val.bc = INSERT_SUBVECTOR(
                                                        nxv4i32 UNDEF, %val, i64 0)
        nxv4i1 [1|1|1|1|1|1|1|1|0|...|0|0] %mask = PTRUE(8 elements)
MSTORE i64 %ptr, nxv4i32 %val.bc, nxv4i1 %mask
```

# Fixed-width Code Generation

```
define void @add_v8i32(<8 x i32>* %a,
                       <8 x i32>* %b) #0 {
  %op1 = load <8 x i32>, <8 x i32>* %a
  %op2 = load <8 x i32>, <8 x i32>* %b
  %res = add <8 x i32> %op1, %op2
  store <8 x i32> %res, <8 x i32>* %a
  ret void
}

attributes #0 = { vscale_range(2,16) }
```

```
add_v8i32:
// %bb.0:
        ptrue   p0.s, vl8
        ld1w    { z0.s }, p0/z, [x0]
        ld1w    { z1.s }, p0/z, [x1]
        add     z0.s, p0/m, z0.s, z1.s
        st1w    { z0.s }, p0, [x0]
        ret
```

The code-generator knows that vscale is at least 2, so can use the (2(or more) x 128bit) SVE vectors.

# SLP Vectorization

We get SLP Vectorization with SVE for free when compiling with Clang using:

`-msve-vector-bits=512+`

```
void SLP_add(double * restrict a,
             double *b, double Val) {
  a[0] = b[0] + Val;
  a[1] = b[1] + Val;
  a[2] = b[2] + Val;
  a[3] = b[3] + Val;
  a[4] = b[4] + Val;
  a[5] = b[5] + Val;
  a[6] = b[6] + Val;
  a[7] = b[7] + Val;
}
```

```
SLP_add:
    ptrue   p0.d, vl8
    ld1d    { z1.d }, p0/z, [x1]
    mov     z0.d, d0
    fadd    z0.d, p0/m, z0.d, z1.d
    st1d    { z0.d }, p0, [x0]
    ret
```

# What’s next?

- Interleaved accesses (requires new shuffle intrinsics)
- Predicated vector epilogue block.
- Making use of llvm.vp intrinsics (Vector Predication).
- Scalable Auto-Vec Enabled by default for Arm cores in LLVM 14.

# Thank you!