Bringing Armv8-a Scalable Vector Extension into OpenJDK

Ningsheng Jian

Outline

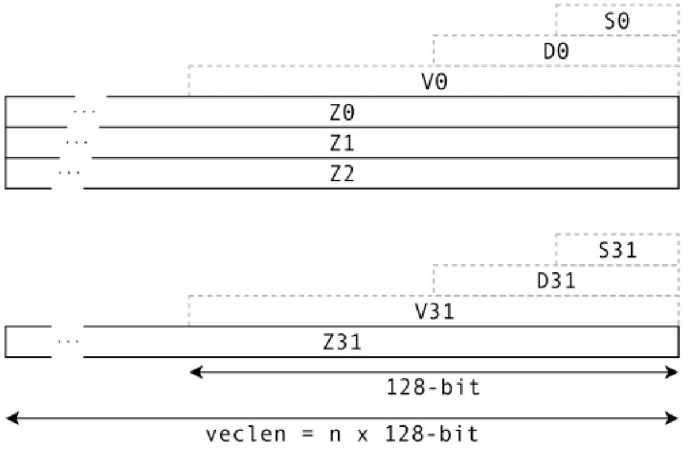
- Scalable Vector Extension (SVE) Introduction
- JEP 338: Vector API
- SVE Implementation Status in OpenJDK
- Future work

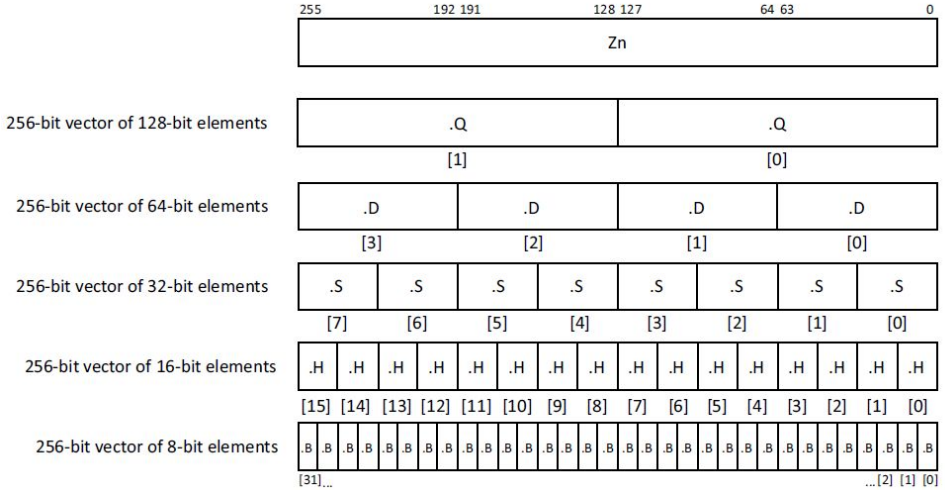
Scalable Vector Extension (SVE) Overview

- Scalable Vector Extension (SVE) is a next generation of SIMD extension to Armv8-A
  - Scalable vector length, implementation defined multiple of 128 bits, up to 2048 bits
  - Per-lane Predication
  - Predicate-driven loop control and management
  - Gather-load and scatter-store
  - Horizontal operations
- 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

Scalable Vectors

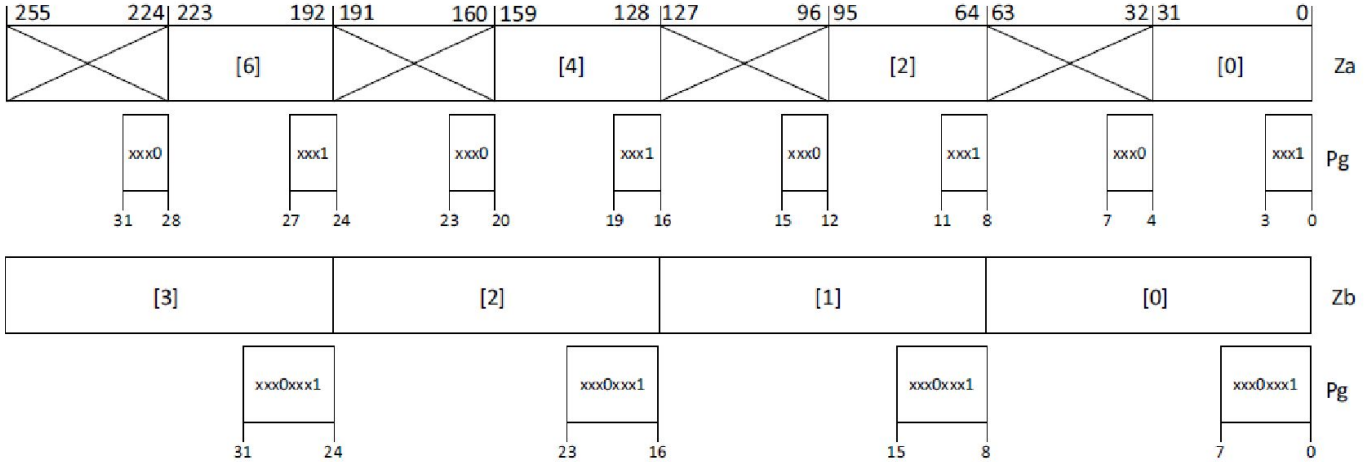
- 32 new scalable vector registers, Z0-Z31
  - Length of 128-2048bit, decided by implementation
  - Bottom 128bits are shared with AArch64 SIMD&FP registers
  - There is no preferred vector length
  - The Vector Length Agnostic (VLA) code generation style makes code portable across all possible vector lengths, no need to recompile.





Per-lane predication

- 16 scalable predicate registers, P0-P15.
  - 1 bit of predicate register is mapped to 1 byte of vector register
    - Predicate register is one-eighth of the size of a Vector register
    - Predicate register is an IMPLEMENTATION DEFINED multiple of 16 bits
  - P0-P7 is the Governing predicate - lane mask to determine active elements
  - Supporting Zeroing(`/Z`) or Merging(`/M`) to the inactive vector elements



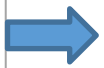
Vector Length Agnostic (VLA) Programming

- SVE implementations may have different actual vector lengths
- VLA and SVE allow the same program binary to any implementations
- VLA is made possible by the per-lane predication, predicate-driven loop control, vector partitioning and software-managed speculation features of SVE
- An example for SVE and VLA to be shown

A Simple Example

- A simple C function and it's typical NEON vectorization code.

```
void array_add(int *restrict a, int *restrict b,
               int *restrict c, int length) {
  for (int i = 0; i < length; i++) {
    a[i] = b[i] + c[i];
  }
}
```



```
    // ... Header
.L4: ldr    q0, [x1, x4]
     ldr    q1, [x2, x4]
     add    v0.4s, v0.4s, v1.4s
     str    q0, [x0, x4]
     add    x4, x4, 16
     cmp    x4, x5
     bne    .L4
     and    w4, w3, -4
     cmp    w3, w4
     beq    .L1
.L3: sxtw   x6, w4
     add    w5, w4, 1
     cmp    w3, w5
     lsl    x5, x6, 2
     ldr    w7, [x1, x6, lsl 2]
     ldr    w8, [x2, x6, lsl 2]
     add    w7, w7, w8
     str    w7, [x0, x6, lsl 2]
     ble    .L1
     add    x6, x5, 4
     add    w4, w4, 2
     cmp    w3, w4
     ldr    w3, [x1, x6]
     ldr    w4, [x2, x6]
     add    w3, w3, w4
     str    w3, [x0, x6]
     ble    .L1
     add    x3, x5, 8
     ldr    w2, [x2, x3]
     ldr    w1, [x1, x3]
     add    w2, w2, w1
     str    w2, [x0, x3]
.L1:
     ...
```

main loop

tail loop

A Simple Example - cont.

- The SVE code:

```
        cmp   w3, 0
        ble   .L1
        mov   x4, 0
        sxtw  x3, w3
        whilelo       p0.s, xzr, x3
.L3:
        ld1w  z1.s, p0/z, [x1, x4, lsl 2]
        ld1w  z0.s, p0/z, [x2, x4, lsl 2]
        add   z0.s, z0.s, z1.s
        st1w  z0.s, p0, [x0, x4, lsl 2]
        incw  x4
        whilelo       p0.s, x4, x3
        bne   .L3       // b.any
.L1:
        ret
```

Scalar induction variable (i)

Setup entry predication

Load data based on predication

Store result based on predication

Increment induction variable by vector element counts

Generate predication for the next iteration

If there's any active elements, loop back

No tail loop

SVE Enablement on OpenJDK

- Enablement work in Hotspot VM
  - Assembler
  - Intrinsic
  - c2 compiler
    - Register allocator
    - Backend codegen
    - A new auto-vectorizer to support VLA programming mode
    - ...
- What about API/core-libs?
  - Is JEP-338 Vector API friendly to SVE?
    - To be improved
  - Can Vector API help for SVE backend enablement work?
    - Yes

Vector API

- JEP 338: provide an initial iteration of an incubator module, jdk.incubator.vector, to express vector computations that reliably compile at runtime to optimal vector hardware instructions on supported CPU architectures and thus achieve superior performance to equivalent scalar computations.
  - To make good use of CPU SIMD ability by using pure java code.
  - Expose data parallel operations through a set of cross-platform APIs.
  - With a well defined Vector APIs, we can easily map high level Java API to low level SIMD instructions.
- Developed by Oracle and Intel.

Vector API Example

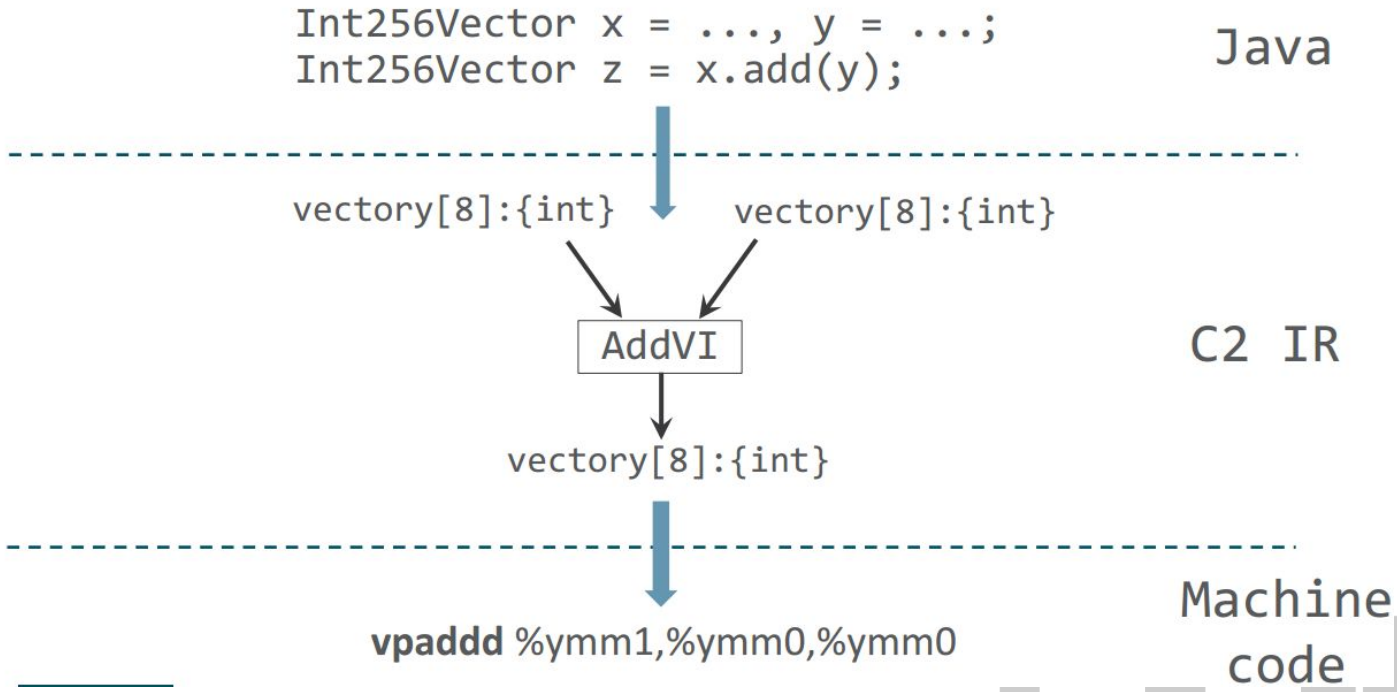
```
public void vectorAdd(int[] a, int[] b, int[] c) {
    IntVector.IntSpecies<Shapes.S128Bit> species = IntVector.species(Shapes.S_128_BIT);
    int i = 0;
    for (; (i + species.length()) < c.length; i += species.length()) {
        IntVector<Shapes.S128Bit> va = species.fromArray(a, i);
        IntVector<Shapes.S128Bit> vb = species.fromArray(b, i);
        va.add(vb).intoArray(c, i);
    }
    for (; i < c.length; i++) { // tail loop
        c[i] = a[i] + b[i];
    }
}
```

Fixed vector length

```
<S extends Shape>
void vectorAdd(int[] a, int[] b, int[] c, IntVector.IntSpecies<S> species) {
    int i = 0;
    for (; (i + species.length()) < c.length; i += species.length()) {
        var va = species.fromArray(a, i);
        var vb = species.fromArray(b, i);
        va.add(vb).intoArray(c, i);
    }
    for (; i < a.length; i++) { // tail loop
        c[i] = a[i] + b[i];
    }
}
IntVector.IntSpecies<?> species = IntVector.preferredSpecies();
vectorAdd(intA, intB, intC, species);
```

More portable version

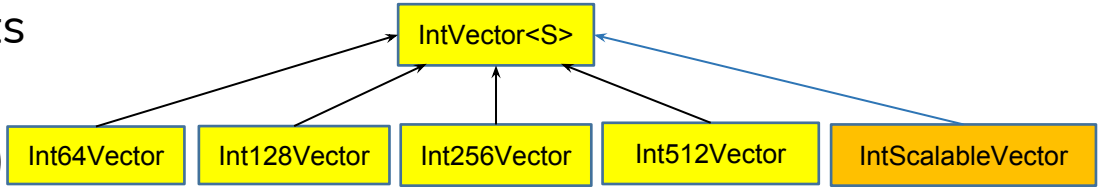
How does it work - From Java to Machine Code (x86)



From Vladimir Ivanov: http://cr.openjdk.java.net/~vlivanov/talks/2017_VectorAPI_C2_Enhancements.pdf

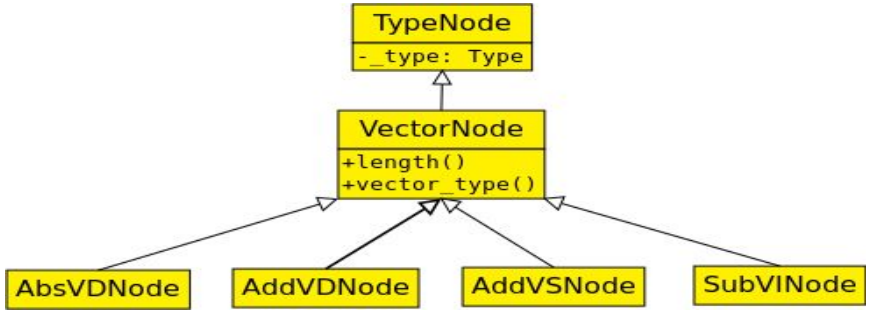
Scalable Vector Support in Vector API

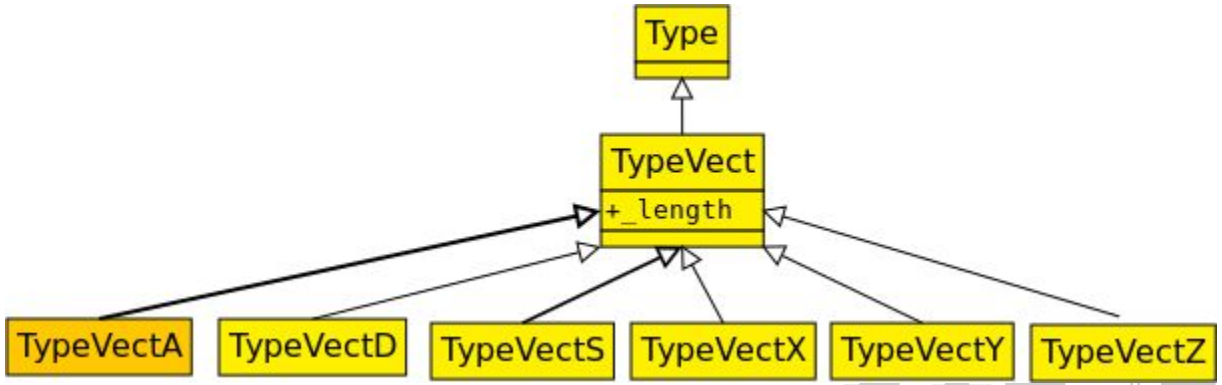
- Currently Vector API supports Intel biased 64bits, 128bits(xmm), 256bits(ymm) and 512bits(zmm)
- Supporting for Scalable Vector Sizes
- With scalable vector shapes, we can reuse current middle-end code easily
- It is actually a known vector size represented as Scalable for all SVE supported lengths




```
public static final SScalableBit S_Scalable_BIT = new SScalableBit();
public static final class SScalableBit extends Vector.Shape {
    @Override
    public int bitSize() {
        Unsafe u = Unsafe.getUnsafe();
        return u.getMaxVectorSize(byte.class) * 8;
    }
}
```

VectorNode and Types in Hotspot C2 Compiler





Add a new length Agnostic vector type for SVE

Scalable Vector Register Definition

- Register Masks for Scalable Vectors
  - Use 8 mask bits to represent a scalable vector register – double of NEON bits.
  - Special handle in register allocation code for these bits.

```
reg_class vectora_reg (
    V0, V0_H, V0_J, V0_K, V0_L, V0_M, V0_N, V0_O,
    V1, V1_H, V1_J, V1_K, V1_L, V1_M, V1_N, V1_O,
    ...
```

```
const RegMask _VECTORA_REG_mask( 0x0, 0x0, 0xffffffff, 0xffffffff, 0xffffffff,
const RegMask _VECTORD_REG_mask( 0x0, 0x0, 0x3030303, 0x3030303, 0x3030303, 0x
const RegMask _VECTORX_REG_mask( 0x0, 0x0, 0xf0f0f0f, 0xf0f0f0f, 0xf0f0f0f, 0x
```

- Real size of vector_a reg is actually known to JIT compiler

Hotspot C2 Codegen for SVE

- Only unpredicate version for now. Predicate support will also be added.

```
operand vecA()
%{
  constraint(ALLOC_IN_RC(vectora_reg));
  match(VecA);
  op_cost(0);
  format %{ %}
  interface(REG_INTER);
%}

instruct vaddAI(vecA dst, vecA src1, vecA src2)
%{
  predicate(UseSVE && n->as_Vector()->length() > 4);
  match(Set dst (AddVI src1 src2));
  ins_cost(SVE_COST);
  format %{ "add $dst, $src1, $src2\t # sve" %}
  ins_encode %{
    __ addv(as_FloatRegister($dst$$reg), __ S,
            as_FloatRegister($src1$$reg),
            as_FloatRegister($src2$$reg));
  %}
  ins_pipe(vdop128);
%}
```

Future work

- Predicate and Masks supports for Vector API
  - Generic Mask for computation
  - Better vector loop shapes for SVE
- Auto-vectorization

Note: Our work is still in development, and may be changed when upstreaming. And the initial patch has been sent out: http://cr.openjdk.java.net/~njian/sve/

Thanks!

Backup

```
@SuppressWarnings("unchecked")
public static <E> Vector.Species<E, ?> preferredSpecies(Class<E> c) {
    Unsafe u = Unsafe.getUnsafe();

    int vectorLength = u.getMaxVectorSize(c);
    int vectorBitSize = bitSizeForVectorLength(c, vectorLength);
    Shape s = shapeForVectorBitSize(vectorBitSize);
    return species(c, s);
}

private static Shape shapeForVectorBitSize(int bitSize) {
    switch (bitSize) {
        case 64:
            return Shapes.S_64_BIT;
        case 128:
            return Shapes.S_128_BIT;
        case 256:
            return Shapes.S_256_BIT;
        case 512:
            return Shapes.S_512_BIT;
        default:
            if ((bitSize > 0) && (bitSize <= 2048) && (bitSize % 128 == 0)) {
                return Shapes.S_Scalable_BIT;
            } else {
                throw new IllegalArgumentException("Bad vector bit size: " + bitSize);
            }
    }
}
```