JavaOne™

Data Parallel Programming in Java Using the Vector API

Adam Pocock, Oracle
Paul Sandoz, Oracle

JavaOne™

Overview

Motivation

Use cases

Concepts

Examples

Guidance

Status

Motivation

- Most modern CPUs have
  - Special registers called vector registers that can hold multiple data
  - Many vector instructions that operate on those registers
- A **S**ingle **I**nstruction can operate on **M**ultiple **D**ata (**SIMD**)
  - We can use these instructions to implement highly performant data parallel algorithms
- How can we leverage all those instructions using Java?

Motivation

- OpenJDK's HotSpot runtime compiler can leverage some of these instructions
  - Some Java loops can be transformed into data-parallel loops when generating machine code. This is process is called **auto-vectorization**
  - HotSpot is very good at this, but the technique is limited
- How can we overcome the limitations of HotSpot's auto-vectorization?
  - To reliably express a broad set of data parallel algorithms?

Vector API

- Enables a developer to reliably express cross-platform data parallel algorithms leveraging a significant set of vector instructions
  - A **W**hat **Y**ou **S**ee is **W**hat **Y**ou **G**et (**WYSIWYG**) API, almost
- HotSpot reliably compiles code written using the Vector API to vector instructions
  - On Intel, ARM, Power CPUs, and soon RISC-V CPUs
  - Hiding the various differences in CPU architectures

Use cases

- Daniel Lemire's blog and publications present some excellent use cases and highly performant algorithms
  - Lemire's JavaFastPFOR project recently integrated an implementation using the Vector API
- In OpenJDK
  - String encoding, BASE64 encoding, array/string comparison, array hash codes, sorting, cryptography
  - Some are currently supported with explicit assembler code

Example use cases

- Filtering
- Machine learning
- (Lanewise arc-tangent)

Concepts

- A `Vector` is a sequence of elements. Each element is held in a lane
  - Many operations can be performed on vectors
- A `VectorMask` is a sequence of booleans
  - Masks can control if an operation is applied to a lane
- A `VectorShuffle` is a sequence of lane indexes
  - Shuffles can rearrange the elements of a vector

- All sequences are **logical**, fixed in size, ordered, and immutable

Concepts

- A `VectorSpecies` manages vectors, masks, and shuffles
  - Species can instantiate vectors with a specific element type and shape that together determines the length of the vector (or number of lanes)
- A `VectorShape` selects a particular implementation of vectors, masks and shuffles
  - A shape of 256 bits selects vectors whose elements can be stored in a vector register of size 256 bits or greater
  - A species with a 256 bits shape and an element type of `float` will instantiate vectors with 8 lanes

Concepts

- Lanewise operations
  - Unary, binary, ternary, unary test, compare (binary test)
- Cross-lane operations
  - Reduction, rearranging, slicing, compressing/expanding
- Reinterpret operations
  - Conversion between vectors of different element types and shapes

Compute the sum of all positive elements of an array

```java
@Benchmark
public float scalarf() {
    float sum = 0.0f;
    for (int i = 0; i < fvalues.length; i++) {
        float v = fvalues[i];
        if (v > 0) sum += v;
    }

    return sum;
}
```

Data parallel code using the Vector API

```
static final VectorSpecies<Float> F_S = FloatVector.SPECIES_PREFERRED;

@Benchmark
public double vectorf() {
    FloatVector vsum = FloatVector.broadcast(F_S, 0.0f);
    int i = 0;
    for (; i < F_S.loopBound(fvalues.length); i += F_S.length()) {
        FloatVector v = FloatVector.fromArray(F_S, fvalues, i);
        VectorMask<Float> m = v.compare(VectorOperators.GT, 0.0f);
        vsum = vsum.add(v, m);
    }
    float sum = vsum.reduceLanes(VectorOperators.ADD);
    for (; i < fvalues.length; i++) {
        float v = fvalues[i];
        if (v > 0) sum += v;
    }
    return sum;
}
```

Data parallel code using the Vector API

```
species: shape = 256 bits, element type = float
       ∴ lanes = 256 / 32 = 8

// VectorMask<Float> m = v.compare(VectorOperators.GT, 0.0f);
v[1.0, -2.0, 3.0, -4.0, 5.0, -6.0, 7.0, -8.0]
    >     >    >     >    >     >    >     >
 [0.0,  0.0, 0.0,  0.0, 0.0,  0.0, 0.0,  0.0])
    =     =    =     =    =     =    =     =
m[  T,    F,   T,    F,   T,    F,   T,    F]

// vsum = vsum.add(v, m);
vsum[1.0,  1.0,  1.0,  1.0, 1.0,  1.0, 1.0,  1.0]
   m[  T,    F,    T,    F,   T,    F,   T,    F]
       +           +          +          +
   v[1.0, -2.0,  3.0, -4.0, 5.0, -6.0, 7.0, -8.0]
       =           =          =          =
vsum[2.0,  1.0,  4.0,  1.0, 6.0,  1.0, 8.0,  1.0]
```

Generated code is good

```
// 2.6 GHz 6-Core Intel Core i7
// Produced using JMH's dtraceasm profiler
// -XX:-TieredCompilation -XX:LoopUnrollLimit=0

  0.14%  ↗││   0x000000011e9726b0:   vmovdqu 0x10(%rsi,%rdi,4),%ymm1
 10.73%  │││   0x000000011e9726b6:   vaddps %ymm1,%ymm0,%ymm3
 35.68%  │││   0x000000011e9726ba:   vcmpgt_oqps %ymm2,%ymm1,%ymm1
  1.00%  │││   0x000000011e9726bf:   vblendvps %ymm1,%ymm3,%ymm0,%ymm0
 42.19%  │││   0x000000011e9726c5:   add    $0x8,%edi
  2.14%  │││   0x000000011e9726c8:   cmp    %r10d,%edi
         ╰││   0x000000011e9726cb:   jl     0x000000011e9726b0
```


Comparing source with generated code (WYSIWYG)

```
for (;
     // cmp    %r10d,%edi
     i < F_S.loopBound(dvalues.length);
     // add    $0x8,%edi
     i += F_S.length()) {

    // vmovdqu
    FloatVector v = FloatVector.fromArray(F_S, fvalues, i);
    // vcmpgt_oqps
    VectorMask<Float> m = v.compare(VectorOperators.GT, 0.0f);
    // vaddps
    // vblendvps
    vsum = vsum.add(v, m);
}
```

Performance is good (lower is better)

```
// 2.6 GHz 6-Core Intel Core i7
// -XX:-TieredCompilation
Benchmark  (size)  Mode  Cnt    Score    Error  Units
scalarf      1024  avgt    5  982.786 ± 44.711  ns/op
vectorf      1024  avgt    5  169.881 ±  4.465  ns/op
```

Example: filtering

- Imagine a table arranged as an in-memory columnar data structure

```
position,  value,    x,    y,    z
      p0      v0   ...,  ...,  ...
      p1      v1
      p2      v2
      p3      v3
      p4      v4
      p5      v5
      p6      v6
      p7      v7
      ...    ...
```

- The first column represents the position or identifier of rows in the table

Filtering

- How can we filter the rows whose `values` are within some upper and lower bound?

  e.g.,

```
positions[p0, p1, p2, p3, p4, p5, p6, p7]
   values[-1,  0,  5,  4,  3,  2, 10, 12]
          0 <= v <= 4
 filtered[p1, p3, p4, p5]
```

Sequential code

```java
public boolean testInt(int value) {
    return value >= lower && value <= upper;
}

@Benchmark
public void scalari() {
    int matchCount = 0;
    for (int i = 0; i < values.length; i++) {
        if (testInt(values[i])) {
            filtered[matchCount++] = positions[i];
        }
    }
}
```


Data parallel code using the Vector API

- The Vector API in JDK 19 has a new cross-lane operation, `compress` (and its inverse `expand`)
  - This operation is optimized on supporting hardware e.g. Intel CPUs supporting the AVX-512 instruction set
- We can convert the sequential code to data parallel code using this new operation

Data parallel code using the Vector API

```
public VectorMask<Integer> testIntVector(IntVector values) {
    return values.compare(GE, lower).and(values.compare(LE, upper));
}

@Benchmark
public void vectori() {
    int matchCount = 0, i = 0;
    for (; i < I_S.loopBound(values.length); i += I_S.length()) {
        var v = IntVector.fromArray(I_S, values, i);
        VectorMask<Integer> match = testIntVector(v);

        var pV = IntVector.fromArray(I_S, positions, i);
        var fV = pV.compress(match);

        fV.intoArray(filtered, matchCount);
        matchCount += match.trueCount();
    }
    ...
```


Data parallel code using the Vector API

```
// VectorMask<Integer> match = testIntVector(v);
values[-1,  0,  5,  4,  3,  2, 10, 12]
       0 <= v <= 4
 match[ F,  T,  F,  T,  T,  T,  F,  F]

// var fV = pV.compress(match);
positions[p0, p1, p2, p3, p4, p5, p6, p7]
    match[ F,  T,  F,  T,  T,  T,  F,  F]
          compress
 filtered[p1, p3, p4, p5,  0,  0,  0,  0]
```


Performance is good (higher is better)

```
// On Intel AVX-512 system
Benchmark (selectivity)   Mode   Cnt         Score         Error  Units
scalari               128  thrpt     3  1294956.909 ±    1463.924  ops/s
vectori               128  thrpt     3  7574101.344 ±   30222.336  ops/s
```


Sorting

- We can use `compress` to optimize Quicksort
  - Quicksort recursively partitioning elements around a pivot point
  - See blog and paper by Google engineers
- Applicable to Java's sorting of primitive arrays

Example: machine learning

- Machine Learning is a heavy consumer of FLOPS, which the Vector API makes much more accessible in Java
- There are a few basic operations:
  - Matrix multiply
  - Convolution
  - Non-linear element-wise functions like max(0,x), sigmoid, or tanh
- We'll look at the 7x7 convolution operation used as the first step in most image recognition or object detection systems

Convolution

- The convolution operation multiplies each “patch” of an image with a learned weight matrix - the “convolution matrix”
- We move the learned matrix across the image one pixel at a time, multiplying each element of the convolution with the covered image pixel and summing them to produce an output pixel
- It is essentially a per pixel dot product, with some fancy indexing

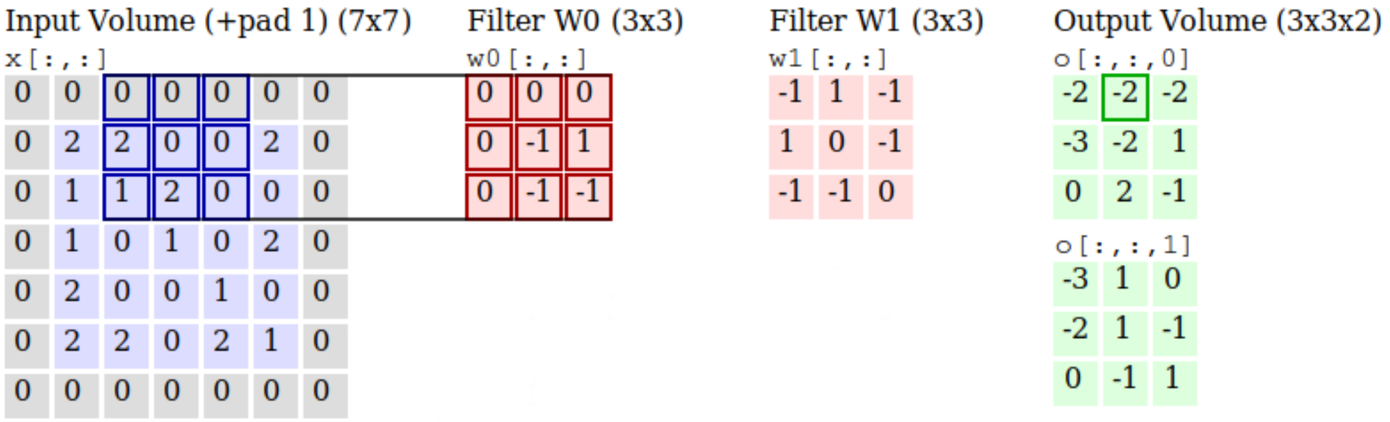

Figure adapted from Stanford’s CS231n course

Scalar convolution

```
// Compute convolution
for (int i = 0; i < output.x; i++) {
    for (int j = 0; j < output.y; j++) {
        double accumulator = 0.0;
        int iOffset = 0;
        int jOffset = 0;
        for (int k = 0; k < convFilter.values.length; k++) {
            final int newIdx = ((i+iOffset) * y) + (j+jOffset);
            accumulator = Math.fma(values[newIdx], convFilter.values[k], accumulator);
            jOffset++;
            if (jOffset == convFilter.y) {
                iOffset++;
                jOffset = 0;
            }
        }
        output.add(i,j,accumulator);
    }
}
```


SIMD convolution

```java
static final VectorSpecies<Double> D_S = DoubleVector.SPECIES_PREFERRED;

// Compute convolution
for (int i = 0; i < output.x; i++) {
    for (int j = 0; j < output.y; j++) {
        fillBuffer(buffer, i, j, convFilter.x, convFilter.y);
        int k = 0;
        var accumVec = DoubleVector.zero(D_S);
        for (; k < D_S.loopBound(buffer.length); k += D_S.length()) {
            var data = DoubleVector.fromArray(D_S, buffer, k);
            var conv = DoubleVector.fromArray(D_S, convFilter.values, k);
            accumVec = data.fma(conv,accumVec);
        }

        double accumulator = accumVec.reduceLanes(VectorOperators.ADD);
        for (; k < buffer.length; k++) {
            accumulator = Math.fma(buffer[k], convFilter.values[k], accumulator);
        }
        output.add(i, j, accumulator);
    }
}
```

Performance is ok (higher is better)

```
// On 8 core VM using an Intel Xeon Platinum 8358 (Ice Lake, AVX-512)
Benchmark                                  Mode  Cnt  Score    Error   Units
Benchmarks.scalarBufferedConv             thrpt    5  0.245 ±  0.002  ops/ms
Benchmarks.scalarConv                     thrpt    5  0.428 ±  0.001  ops/ms
Benchmarks.simdBufferedConv               thrpt    5  0.459 ±  0.002  ops/ms
```

- Performance is not optimal due to the copy needed to line up the image
- The buffered SIMD implementation is twice as fast as the buffered scalar one
  - But only slightly faster than the unbuffered scalar implementation with more complex indexing
- If we know the convolution size we can hand craft code for just that operation
- Nvidia do this in cuDNN (their GPU machine learning primitives library)
- With the Vector API, we can do the same thing in Java

Specialized 7x7 SIMD convolution

```java
// Load filter into registers
final boolean[] sevenLaneMask = new boolean[]{true,true,true,true,true,true,true,false};
final var mask = VectorMask.fromArray(D_S, sevenLaneMask, 0);
final var one = DoubleVector.fromArray(D_S, convFilter.values, 0, mask);
final var two = DoubleVector.fromArray(D_S, convFilter.values, 7, mask);
...
final var seven = DoubleVector.fromArray(D_S, convFilter.values, 42, mask);

// Compute convolution
for (int i = 0; i < output.x; i++) {
    for (int j = 0; j < output.y; j++) {
        var outputVec = DoubleVector.zero(D_S);

        var input = DoubleVector.fromArray(D_S, values, (i*y) + j);
        outputVec = input.fma(one, outputVec);
        input = DoubleVector.fromArray(D_S, values, ((i+1)*y) + j);
        outputVec = input.fma(two, outputVec);
        ...
        input = DoubleVector.fromArray(D_S, values, ((i+6)*y) + j, mask);
        outputVec = input.fma(seven, outputVec);

        output.add(i, j, outputVec.reduceLanes(VectorOperators.ADD));
    }
}
```

Performance is improved

```
// On 8 core VM using an Intel Xeon Platinum 8358 (Ice Lake, AVX-512)
Benchmark                                Mode  Cnt  Score    Error   Units
Benchmarks.scalarBufferedConv           thrpt    5  0.245 ±  0.002  ops/ms
Benchmarks.scalarConv                   thrpt    5  0.428 ±  0.001  ops/ms
Benchmarks.simdBufferedConv             thrpt    5  0.459 ±  0.002  ops/ms
Benchmarks.simdSpecializedConv          thrpt    5  1.379 ±  0.012  ops/ms
```

- The loop is unrolled and specialized to convolution size & vector width
- We lose 1/8th of our computation each time as the 8th lane is masked
  - But C2 can promote the convolution matrix into vector registers
- So we load the convolution matrix once, and reuse it for each pixel
  - Resulting in a 3x faster implementation, at the cost of more complicated code

Example: lanewise arc-tangent

- The JDK bundles some of the routines from Intel's Short Vector Math Library for lanewise transcendental operations
  - Same accuracy as the scalar methods in `java.lang.Math`
- We can increase performance for reduced accuracy using a different implementation
- A good implementation that balances performance with accuracy can be found in the Cephes Mathematical Library
  - https://github.com/jeremybarnes/cephes/blob/master/single/atanf.c

Cephes atanf algorithm pseudo code

```
// Range reduction to interval [0, tan(pi/8)]
neg = false
if (x < 0) {
    neg = true
    x = −x
}

A0 = 0
if (x > tan(3pi/8)) {
    x = −(1 / x)
    A0 = pi/2
} else (if x > tan(pi/8)) {
    x = (x − 1) / (x + 1)
    A0 = pi/4
}
y = A9 ∗ x^9 − A7 ∗ x^7 + A5 ∗ x^5 − A3 ∗ x^3 + x + A0

if (neg) y = −y
```


Absolute value and extract the sign bit

```
IntVector sign_bit = x.reinterpretAsInts();
// Take the absolute value
x = sign_bit.and(INV_SIGN_MASK_I).reinterpretAsFloats();
// Extract the sign bit (upper one) to be applied to the result
sign_bit = sign_bit.and(SIGN_MASK_I);
```

Comparisons producing masks for blending

```java
// x > tan(3pi/8)
VectorMask<Float> cmpHi = x.compare(VectorOperators.GT, ATANF_RANGE_HI_F);
// x > tan(pi/8)
VectorMask<Float> cmpLow = x.compare(VectorOperators.GT, ATANF_RANGE_LOW_F);
// x > tan(pi/8) && !( x > tan(3pi/8) )
VectorMask<Float> cmpLowHi = cmpLow.andNot(cmpHi);
```

Blend in x and A0 according to range of x

```
// x0 = -1.0 / x
FloatVector x0 = FloatVector.broadcast(F_S, -1.0f).div(x);
// x1 = (x - 1.0) / (x + 1.0)
FloatVector x1 = x.sub(1.0f).div(x.add(1.0f));

// Blend in -1 / x and (x - 1) / (x + 1) and x to x
// x0 if x > tan(3pi/8)
x = x.blend(x0, cmpHi);
// x1 if x > tan(pi/8) && !( x > tan(3pi/8) )
x = x.blend(x1, cmpLowHi);

// Blend in pi/2 and pi/4 and 0 to A0 coefficient
// pi/2 if x > tan(3pi/8)
FloatVector y0 = FloatVector.zero(F_S).blend(HALF_PI_F, cmpHi);
// pi/4 if x > tan(pi/8) && !( x > tan(3pi/8) )
FloatVector y = y0.blend(QUATER_PI_F, cmpLowHi);
```

Compute the polynomial

```
// A9 * x^9 - A7 * x^7 + A5 * x^5 - A3 * x^3 + x + A0
FloatVector zz = x.mul(x);
FloatVector acc = zz.fma(
        FloatVector.broadcast(F_S, ATANF_COF_P0_F),
        FloatVector.broadcast(F_S, -ATANF_COF_P1_F));
acc = acc.fma(
        zz,
        FloatVector.broadcast(F_S, ATANF_COF_P2_F));
acc = acc.fma(
        zz,
        FloatVector.broadcast(F_S, -ATANF_COF_P3_F));
acc = acc.mul(zz);
acc = acc.fma(x, x);
// Add A0 coefficient
y = y.add(acc);
```

Restore the sign

```
y = y.reinterpretAsInts()
        .lanewise(VectorOperators.XOR, sign_bit)
        .reinterpretAsFloats();
```


Performance is good (lower is better)

```
// 2.6 GHz 6-Core Intel Core i7
// -XX:-TieredCompilation
Benchmark  (size)  Mode  Cnt      Score      Error  Units
scalar       1024  avgt   10   3437.683 ±   65.388  ns/op
vector       1024  avgt   10    494.722 ±    3.962  ns/op
```

Guidance

- Measure using a benchmark tool such as JMH
  - Use perfasm (on Linux) or dtraceasm (on Mac) to look at generated code
- Store species in static final fields
  - This will ensure the runtime compiler fold away constant expressions
- Avoid storing vectors in instance fields or arrays
  - Keep use to local variables and method arguments and return values
  - Static fields can be used for vector constants
- Avoid identity sensitive operations on vectors or assigning them to `null`
- Take care when splitting an algorithm into multiple methods
  - Method calls need to inline
  - No support vector calling conventions

Status

- JEP 426: Vector API (Fourth Incubator) delivered in JDK 19
  - Compile and run with `--add-modules=jdk.incubator.vector`
  - Solid progress made on each delivery to optimize the implementation
- Work has started to align with value classes and types from Project Valhalla
  - Vectors are a special kind of value
- Experimental support for vectors of 16 bit floating point numbers
  - See https://github.com/openjdk/panama-vector/tree/vectorIntrinsics+fp16
  - This also requires value classes and types from Project Valhalla

JavaOne™