White paper

Vector API: Writing own-vector algorithms in OpenJDK* for faster performance

In this paper, we discuss insights into Vector API, which is being developed as part of OpenJDK under Project Panama. First, we'll go over some Vector API fundamentals, basic functionalities, and tips. We'll then show you some code samples of vector algorithms for standard Machine Learning routines and financial benchmarks, and go over some ways to increase performance. These examples should give you some guidelines and best practices for vector programming in Java*, to help you to write successful vector versions of your own compute-intensive algorithms.*

AUTHOR

Rahul Kandu
Software and Services Group
Intel Corporation

ACKNOWLEGMENTS
Software and Services Group
Intel Corporation:

Shirish Aundhe
Mahesh Bhatt
Vivek Deshpande
Chris Elford
Ian Graves
Anil Kumar
Razvan Lupusoru
Sandhya Viswanathan

INTRODUCTION

Modern microprocessors offer parallelism at various granularities in order to deliver high performance. These granularities are at the thread, instruction, data, and pipeline level. Thread- and Instruction-level parallelism is delivered by the CPU itself (via hyper-threading and out-of-order execution). Data-level parallelism is achieved using Single Instruction Multiple Data (SIMD) instructions. Processors can leverage application-level parallelization by simultaneously processing the same operation on multiple data items. Application-level parallelization allows for higher performance speeds by orders of magnitude.

CHALLENGES: LIMITED SIMD SUPPORT IN JAVA*

SIMD in Java* opens up ways for developers to explore new opportunities in areas like high performance computing (HPC), machine learning (ML) linear algebra-based algorithms, deep learning (DL) and artificial intelligence (AI) frameworks, DL training workloads, and financial services that operate on huge data sets.

Right now there is limited SIMD support in Java, with some new optimizations in the Java Virtual Machine 9 (JVM 9). However, even though SIMD optimization is supported in Java (via auto-vectorization and Super-word optimizations), that support is currently limited to simple expressions and/or loops. Fortunately, these optimizations don't require you to do additional work. However, you do need to write your Java code in certain ways, in order to take advantage of auto-vectorization.

Because of this, big data applications typically use Unsafe or Java Native Interface (JNI) wrapper packages, in order to have access to faster native libraries. Such big data applications include Apache Flink*, Apache Spark* ML libraries, and BigDL (big data distributed deep learning on Spark). Unfortunately, JNI remains hard to develop, hard to maintain, and adds performance overhead.

Vector API and Project Panama

To address these challenges, Project Panama offers a Vector API. Vector API supports the more complex expressions used in ML and DL (such as basic linear algorithm subprogram, or BLAS), as well as in the financial services industry (FSI) and HPC programs.

```
import jdk.incubator.vector.FloatVector;
import jdk.incubator.vector.Vector;
import jdk.incubator.vector.Shapes;
```

Code sample 1. *To begin using Vector API, import these lines into your program.*

```
private static final FloatVector.FloatSpecies<Shapes.S256Bit> species =
(FloatVector.FloatSpecies<Shapes.S256Bit>) Vector.speciesInstance (Float.class, Shapes.S_256_BIT);
```

Code sample 2. *Create a primordial instance, then create other vectors of that particular size and shape.*

Vector API makes it possible to develop compute-intensive machine and deep learning algorithms, financial algorithms, and training workloads all in Java. Even better, Vector API enables this without needing non-portable native code or incurring JNI overhead. It introduces a set of methods for data-parallel operations on sized vector-types for programming directly, typically without requiring knowledge of an underlying CPU. APIs are further efficiently mapped to SIMD instructions on modern CPUs by the JVM JIT. Also, BLAS algorithms can run up to 3x to 4x faster when implemented using Vector API.

How to use Vector API in Java

A vector interface is bundled as part of the com.oracle.vector package. To begin using Vector API, you need to import the following lines into your program. Depending on the Vector type, you can choose to import *FloatVector, IntVector* etc. See code sample 1, above.

Vector is an Immutable interface. It has been defined this way because of the advantages of concurrency and multi-threading that immutable objects offer.

The Vector type (Vector<E, S>) takes two parameters:

- E The element type, broadly supporting the int, float, and double primitive types.
- S Specifies the shape or bitwise size of the vector.

Vector operations live as virtual methods of a vector instance. Before using vector operations, you must create a primordial instance which captures the element type and vector shape. Using that instance, you can then create other vectors of that particular size and shape. See code sample 2, above.

Once you have created the primordial instance, you can create vector instances of the *FloatVector<Shapes.S256Bit>* type. If necessary, your program should also create first instances of other vector elements and shape types, such as *DoubleVector<Shapes.S512Bit>*.

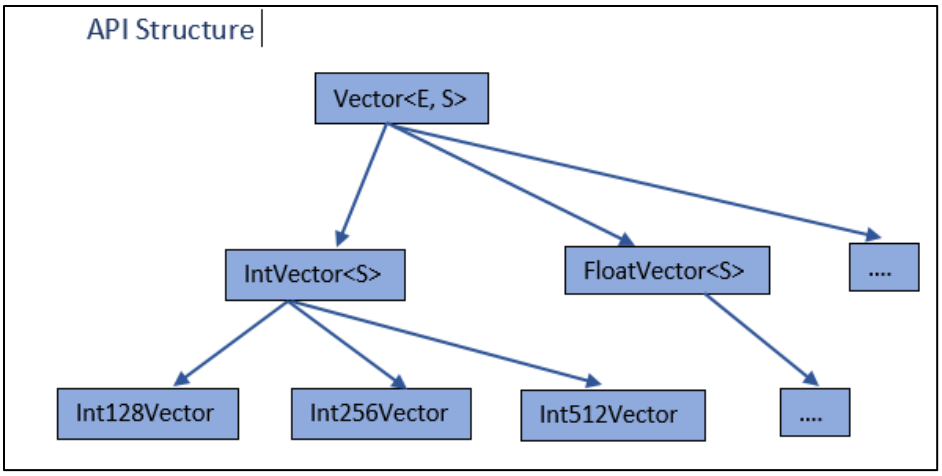

Figure 1. *Vector API class hierarchy*

```
public interface Vector<E, S extends Vector.Shape<Vector<?, ?>>>  {
 Vector<E, S> add (Vector<E, S> v2);
}
```

Code sample 3. *Prototype implementation for a Vector interface.*

```
public abstract class DoubleVector <S extends Vector.Shape<Vector<?, ?>>> implements
Vector<Double, S> {
  DoubleVector() {}
  public DoubleVector<S> add (Vector<Double, S> o) {..}
  public DoubleVector<S> mul (Vector<Double, S> o) {..}
}
```

Code sample 4. *Starting to specialize with an IntVector, with a shape that has yet to be determined.*

```
final class Double512Vector extends DoubleVector<Shapes.S512Bit> {
..
}
```

Code sample 5. *Vector classes specific to the Element types (E) and Shapes (S).*

```
public abstract class DoubleVector<S extends Vector.Shape<Vector<?,?>>> implements
Vector<Double,S> {
 Vector<Double, S> add (Vector<Double, S> v2);
Vector<Double,S> add (Vector<Double, S> o, Mask<Double, S> m);
Vector<Double, S> mul (Vector<Double, S> v2);
Vector<Double, S> mul (Vector<Double, S> o, Mask<Double, S> m);
….
Vector<Double, S> sin ();
Vector<Double, S> sin (Mask<Double, S> m);
Vector<Double, S> sqrt (),
…
}
```

Code sample 6. *Vector classes specific to the Element types (E) and Shapes (S).*

Vector class hierarchy

First, let's look at the Vector API class hierarchy. At the top level, the Vector interface *Vector<E, S>* takes the parameters Element and Shape of the vector. Further down we have abstract classes *IntVector<S>*, *FloatVector<S>*, and so on. The abstract classes specialize the Element type to Double, Integer, and so on. These abstract classes are inherited into classes like *Int128Vector*, *Float512Vector.* In turn, those classes specialize the Shape (bitwise size) of the vectors for each Element type for concrete vectors where vector operations reside.

Vector API has broad support for Float, Integer, and Double types, since most SIMD features revolve around these primitive data types. See Figure 1.

A prototype implementation for a Vector interface is shown in code sample 3.

At the top level, there is a vector interface. Moving down through the hierarchy, we start to specialize with an IntVector, with a shape that has yet to be determined. See code sample 4.

When we drill down to the bottom of the hierarchy, we start getting vector classes which are specific to the particular Element types (E) and Shapes (S), as shown in code sample 5.

Vector-Vector functionality

As we mentioned earlier, Vector is an immutable interface. All the methods in the vector interface provide the result in a new object. They do not modify the input's objects.

The Vector interface supports most of the basic arithmetic and trigonometric operations in both masked and unmasked form. You can see this support in code sample 6.

```
public abstract class IntVector<S extends Vector.Shape<Vector<?,?>>> implements Vector<Integer,S>
{
int sumAll ();
void intoArray(int[] a, int ix);
void intoArray (int [] is, int ix, Mask<Integer, S> m);
Vector<Integer, S> fromArray (int [] fs, int ix);
Vector<Integer, S> blend (Vector<Integer, S> o, Mask<Integer, S> m);
Vector<Integer, S> shuffle (Vector<Integer, S> o, Shuffle<Integer, S> s);
Vector<Integer, S> fromByte (byte f);
…
}
```

Code sample 7. *Supported advanced vector operations.*

```
public static void AddArrays (float [] left, float [] right, float [] res) {
  FloatVector.FloatSpecies<Shapes.S256Bit> species = (FloatVector.FloatSpecies<Shapes.S256Bit>)
Vector.speciesInstance (Float.class, Shapes.S_256_BIT);
       FloatVector<Shapes.S256Bit> l  = species.fromArray (left, 0);
       FloatVector<Shapes.S256Bit> r  = species.fromArray (right, 0);
       FloatVector<Shapes.S256Bit> lr = l.add(r);
       lr.intoArray (res, 0);
  }
```

Code sample 8. *Supported advanced vector operations.*

```
private final FloatVector.FloatSpecies<S> spec;
  FloatVector<S> av = spec.fromArray (a, i);
  FloatVector<S> bv = spec.fromArray (b, i);
```

Code sample 9. *Using the spec vector instance to create vector instances that are length-agnostic.*

```
for (int i = 0; i < C.length; i++) {  //original scalar loop kernel
    C[i] = A[i] + B[i];
}

public class AddClass<S extends Vector.Shape<Vector<?, ?>>> {
     private final FloatVector.FloatSpecies<S> spec;
     AddClass (FloatVector.FloatSpecies<S> v) {spec = v; }
     //vector routine for add
      void add (float [] a, float [] b, float [] c) {
       int i=0;
       for (; i+spec.length ()<a.length;i+=spec.length ()) {
          FloatVector<S> av = spec.fromArray (a, i);
          FloatVector<S> bv = spec.fromArray (b, i);
          av.add (bv).intoArray(c, i);
       }
      //clean up loop
       for (;i<a.length;i++) c[i]=a[i]+b[i];
```

Code sample 10. *Loop kernel for the vector addition of two arrays.*

Support for advanced vector operations

Advanced vector operations are also supported. These include horizontal reductions, broadcasting primitives, blend/shuffle operations, masked comparisons, and array load/stores. You can see the supported operations in code sample 7.

The next code sample shows the vector addition of two arrays. *AddArrays* creates two Vectors of Float Element type and 256-bit Shape (size). In this example, we use the *fromArray (array, index)* operation to load the vectors from arrays *left []* and *right [],* followed by the vector *add ()* operation. We also use the *intoArray (array, index)* operation to store the result of the computation into *res []*. See code sample 8.

Write loop kernels independent of vector size.

Vector API lets you write loop kernels independent of vector size. While writing loop kernels, you can simply query the Vector API for the vector size *species.length ()*. You can then use that size for striding through arrays and byte-buffers. This allows you to make your code portable across multiple vector sizes.

In code sample 9, you can see that user-defined classes must extend *Vector.Shape<Vector<?, ?>>*. Within the class definition, you must include a primordial instance *spec* as *field* of *final* type. Once you do that, you can then use the *spec* vector instance to create vector instances that are length-agnostic.

The next code sample (sample 10) shows a loop kernel for the vector addition of two arrays. In this example, the Shape of the vector is parameterized by *FloatVector<S>* in the loop kernel.

```
FloatVector.FloatSpecies<Shapes.S256Bit> species = (FloatVector.FloatSpecies<Shapes.S256Bit>)
Vector.speciesInstance (Float.class, Shapes.S_256_BIT);
AddClass<Shapes.S256Bit> myAddObject = new AddClass<> (species);
```

Code sample 11. *Explicitly specifying the Vector shape.*

```
for (int i = 0; i < SIZE; i++) {
        float res = b[i];
        if (a[i] > 1.0) {
            res = res * a[i];
        }
        c[i] = res;
    }
```

Code sample 12. *Scalar code for loops with conditional statements.*

```
public void useMask (float [] a, float [] b, float [] c, int SIZE) {
 FloatVector.FloatSpecies<Shapes.S256Bit> species = (FloatVector.FloatSpecies <Shapes.S256Bit>)
Vector.speciesInstance Float.class, Shapes.S_256_BIT);
 FloatVector<Shapes.S256Bit> tv = species.broadcast (1.0f); int i = 0;
 for (; i+species.length() < SIZE; i+ = species.length()){
   FloatVector<Shapes.S256Bit> rv = species.fromArray (b,i);
   FloatVector<Shapes.S256Bit> av = species.fromArray (a,i);
   Vector.Mask<Float,Shapes.S256Bit> mask = av.greaterThan (tv);
   rv.mul (av, mask).intoArray(c,i);
 }
  //tail processing
}
```

Code sample 13. *Vector loop.*

Note that, after writing the vectorized version of the loop, you still need to iterate over the remaining array elements in a scalar fashion. The example above shows this in the *tail cleanup loop*.

While creating objects of *AddClass* type, you will need the primordial vector instance *species* specialized by Shape. As part of the object definition, you must explicitly specify the Vector shape as *AddClass<Shapes.S256Bit>*. See code sample 11.

Loops with conditional statements

Loops with conditional statements can also be written as vector versions using masked operations. In this next example, we first generate *Vector.Mask<Float, Shapes.S256Bit>* using the *greaterThan ()* operation. Mask is used for the *mul ()* operation.

Code sample 12 shows the scalar code.

Code sample 13 shows the vector loop.

We'll show you several other vector programming tips and tricks using code samples in the next discussion.

```
static void VecDdot (double [] a, int a_offset, double [] b, int b_offset) {
  DoubleVector.DoubleSpecies<Shapes.S512Bit> spec =
(DoubleVector.DoubleSpecies<Shapes.S512Bit>)    Vector.speciesInstance (Double.class,
Shapes.S_512_BIT);
  int i = 0; double sum = 0;
  for (; i + spec.length () < a.length; i += spec.length ()) {
       DoubleVector<Shapes.S512Bit> l = spec.fromArray (a, i + a_offset);
       DoubleVector<Shapes.S512Bit> r = spec.fromArray (b, i + b_offset);
       sum+=l.mul(r).sumAll();
   }
   for (; i < a.length; i++) sum += a[i + a_offset] * b[i + b_offset]; //tail
}
```

Code sample 14. *The Vector program VecDdot multiplies two vectors and performs the horizontal reduction sumAll () on the result of the multiplication*

```
static void VecDaxpy (double [] a, int a_offset, double [] b, int b_offset, double alpha) {
   DoubleVector.DoubleSpecies<Shapes.S512Bit> spec =
(DoubleVector.DoubleSpecies<Shapes.S512Bit>)    Vector.speciesInstance (Double.class,
Shapes.S_512_BIT);
   DoubleVector<Shapes.S512Bit> alphaVec = spec.broadcast (alpha); int i = 0;
       for (; (i + spec.length ()) < a.length; i += spec.length ()) {
          DoubleVector<Shapes.S512Bit> bv = spec.fromArray (b, i + b_offset);
          DoubleVector<Shapes.S512Bit> av = spec.fromArray (a, i + a_offset);
          bv.add (av.mul (alphaVec)).intoArray (b, i + b_offset);
       }
       for (; i < a.length; i++) b[i + b_offset] += alpha * a[i + a_offset];          //tail of the loop
    }
```

Code sample 15. *Using the DAXPY algorithm to broadcast the alpha scalar value outside the loop using broadcast ().*

Code samples: BLAS

The BLAS sub-program contains a set of low-level linear algebra routines. These routines can take advantage of both Vector operations and SIMD instructions. BLAS algorithms are used in most common ML algorithms and utilities. For example, BLAS-3 GEMM is a ubiquitous algorithm used in DL and neural networks.

This discussion focuses on how to implement Vector API for few of the BLAS routines. Most of the BLAS algorithms can be fully expressed using Vector API. You can find more code samples here.

The BLAS-1 DDOT routine

Our first example uses the BLAS-1 DDOT routine. This routine computes the dot product of two vectors or arrays (A*B). Its product is returned as a scalar value.

The Vector program *VecDdot* uses the vector *mul ()* routine to multiply both vectors and perform the horizontal reduction *sumAll ()* on the result of the multiplication. The vector loop will need a tail/post loop to iterate over the remaining scalar values until it reaches the end of the arrays.

When implemented using vectors and when run on a platform that supports Intel® Advanced Vector Extensions 2 (Intel® AVX2), the performance of the SDOT routine (which operates on two float vectors) can be increased by up to 3.25x. See code sample 14.

Horizontal reductions are platform-specific

Horizontal reductions like *sumAll ()* and *subAll ()* are platform-specific operations. This is because different CPUs may have different instruction support at the hardware level.

You must be careful while handling horizontal reductions for Float and Double-based custom types in Java applications. The order of operations matters, and your implementation must be explicit for your specific reduction.

The DAXPY algorithm in vector loops

DAXPY is a BLAS level-1 algorithm used to compute a constant multiplied by a vector, plus a vector. When writing vector loops, you should broadcast all constant scalars into vectors before entering the loop kernel for better performance. (This helps you avoid generating unnecessary garbage.)

In this next example, we broadcast the alpha scalar value outside the loop using *broadcast ()*. The result is computed from two arithmetic operations *add ()* and *mul ()*. The result is stored in array *b [],* using *intoArray ()*. See code sample 15.

```
void VecDgemm (String transa, String transb, int m, int n, int k, double alpha, double[] a, int a_offset,
int lda, double[] b, int b_offset, int ldb, double beta, double[] c, int c_offset, int ldc, boolean nota,
boolean notb) {
DoubleVector.DoubleSpecies<Shapes.S512Bit> spec=
(DoubleVector.DoubleSpecies<Shapes.S512Bit>)    Vector.speciesInstance (Double.class,
Shapes.S_512_BIT);
double temp;
 if (notb && nota) {              for (j = 0; j < n; j++) {
              for (l = 0; l < k; l++) {
                 if (b[l + j * ldb + b_offset] != 0.0) {
                    temp = alpha * b [l + j * ldb + b_offset];
                    DoubleVector<Shapes.S512Bit> tv = spec.broadcast (temp);
                    for (i = 0; (i + spec.length ()) < m; i += spec.length ()) {
                       DoubleVector<Shapes.S512Bit> av = spec.fromArray (a, i + l * lda + a_offset);
                       DoubleVector<Shapes.S512Bit> cv = spec.fromArray (c, i + j * ldc + c_offset);
                       tv.fma (av, cv).intoArray (c, i+j*ldc+c_offset);
                    }
                    for (; i < m; i++)
                       c[i + j * ldc + c_offset] = c[i + j * ldc + c_offset] + temp * a[i + l * lda + a_offset];
                 }
              }
          }
      }
}
```

Code sample 16. *DGEMM: C = alpha*A*B+beta*C. Here, A and B are non-zero matrices that are not transposed; while alpha is a non-zero scalar, and beta scalars as 1.*

```
public double getOptionPrice (double Sval, double Xval, double T)  {
  double val=0.0 , val2=0.0;
  double VBySqrtT = volatility * Math.sqrt (T);
  double MuByT = (riskFree - 0.5 * volatility * volatility) * T;
  //Simulate Paths
  for (int path = 0; path < numberOfPaths; path++) {
   double callValue = Sval * Math.exp (MuByT + VBySqrtT * z[path]) - Xval;
   callValue = (callValue > 0) ? callValue: 0;
   val += callValue;
   val2 += callValue * callValue;
  }
  double optPrice=0.0;
  optPrice = val / numberOfPaths;
  return (optPrice);
 }
```

Code sample 17. *Vector version of trigonometric math exp () operation on the vectors created from VBySqrtT, MuByT and z[path.*

Vector-matrix and matrix-matrix operations

BLAS-2 and BLAS-3 routines, such as DSYR and DGEMM respectively, perform vector-matrix or matrix-matrix operations. Code sample 16 showcases DGEMM: *C = alpha*A*B+beta*C.*

In this example, *A* and *B* are non-zero matrices that are not transposed; while *alpha* is a non-zero scalar, and *beta* scalars as 1.

You can find a VecDgemm example of this here.

In the presence of nested loops, it is possible to broadcast scalars in the outer loop — provided the inner compute loop doesn't modify the scalar value. In our example with BLAS-3, *temp* is broadcast outside, since it is unchanged inside the compute kernel.

When you use vector implementation with Intel® AVX2 support, you can boost the performance of BLAS-II routines such as SSYR by up to 2.5x, and SSPR by up to 4x. You can boost the performance of typical BLAS-III routines like SGEMM by about 4.25x.

The example above also uses the *fma ()* vector (fused multiply-add) operation. You could express the original vector operation *cv.add (av.mul (tv))* as *tv.fma (av, cv),* using the FMA library function in Java. To speed up performance, you should leverage the Intel AVX FMA instructions. Several BLAS-2 and BLAS-3 routines can be written using these vector operations.

Code examples: Financial services applications

Vector API is highly applicable to benchmarks and use cases in the financial services industry. For example, the *getOptionPrice ()* program is used for Monte Carlo European simulations to get the option price. In this program, loop-independent scalar values (like *VBySqrtT, MuByT, Sval,* and *Xval*) can be broadcast first. Within the loop, the *z[path*] array is loaded into a vector. That is followed by a trigonometric math *exp ()* operation on the vectors that were created from *VBySqrtT, MuByT* and *z[path].*

You can write a vector version of this over a number of paths. Code sample 17 shows the code in scalar form.

```
double callValue = Sval * Math.exp (MuByT + VBySqrtT * z [path]) - Xval; //scalar expression
  MuVec.add (VByVec).mul (zv).exp (); // wrong operator precedence
  MuVec.add (VByVec.mul (zv)).exp (); // correct operator precedence
  DoubleVector<Shapes.S512Bit> callValVec=SvalVec.mul (MuVec.add (VByVec.mul (zv)).exp
()).sub (XvalVec);
```

Code sample 18. *When writing mathematical expressions, be careful to maintain operator precedence.*

```
public static double VecGetOptionPrice (double Sval, double Xval, double T, double[] z, int
numberOfPaths, double riskFree, double volatility) {
  DoubleVector.DoubleSpecies<Shapes.S512Bit> spec=
  (DoubleVector.DoubleSpecies<Shapes.S512Bit>)  Vector.speciesInstance (Double.class,
Shapes.S_512_BIT);
   double val = 0.0, val2 = 0.0;
   double VBySqrtT = volatility * Math.sqrt (T);
   double MuByT = (riskFree - 0.5 * volatility * volatility) * T;
   //broadcast happens here
    DoubleVector<Shapes.S512Bit> VByVec = spec.broadcast (VBySqrtT);
    DoubleVector<Shapes.S512Bit> MuVec = spec.broadcast (MuByT);
    DoubleVector<Shapes.S512Bit> SvalVec = spec.broadcast (Sval);
    DoubleVector<Shapes.S512Bit> XvalVec = spec.broadcast (Xval);
    DoubleVector<Shapes.S512Bit> zeroVec =spec.broadcast (0.0D);
    //Simulate Paths
    int path = 0;
    for (; (path + spec.length()) < numberOfPaths; path += spec.length()) {
      DoubleVector<Shapes.S512Bit> zv = spec.fromArray (z, path);
      DoubleVector<Shapes.S512Bit> tv = MuVec.add (VByVec.mul (zv)).exp ();
      DoubleVector<Shapes.S512Bit> callValVec = SvalVec.mul (tv).sub (XvalVec);
      callValVec = callValVec.blend(zeroVec, callValVec.greaterThan (zeroVec));
      val += callValVec.sumAll ();
      val2 += callValVec.mul (callValVec).sumAll ();
       }
       //tail loop goes here
   }
```

Code sample 19. *Vector loop kernel.*

```
callValVec = callValVec.blend (zeroVec, callValVec.greaterThan (zeroVec));
```

Code sample 20. *Extracted line from previous code sample for scalar results in val and val2 variables.*

```
void BinomialOptions (double[] stepsArray, int STEPS_CACHE_SIZE, double vsdt, double x, double
s, int numSteps, int NUM_STEPS_ROUND, double pdByr, double puByr) {
  for (int j=0; j< STEPS_CACHE_SIZE; j++) {
    double profit = s*Math.exp (vsdt* (2.0D* j- numSteps))- x;
    stepsArray[j] = profit > 0.0D? profit: 0.0D;
       }
   for (int j=0; j<numSteps; j++) {
     for (int k=0; k<NUM_STEPS_ROUND; ++k) {
       stepsArray[k] = pdByr * stepsArray[k+1] + puByr * stepsArray[k];
     }
   }
 }
```

Code sample 21. *Scalar algorithm for a binomial lattice.*

Note that, when writing mathematical expressions, you must be careful to maintain operator precedence. See code sample 18.

Code sample 19 shows the vector loop kernel.

Java ternary operations: Use the blend () routine

The Vector API provides a *blend ()* routine for Java ternary operations (such as *maxVal=a>0? a: 0)* of two Vectors.

If you look at the previous code sample, *blend ()* takes a second operand as *zero vector instance zeroVec,* and the masked output from *greaterThan (zeroVec)* of the first vector. This is then followed by a horizontal reduction *sumAll ()*. In turn, that accumulates the scalar results in the *val* and *val2* variables.

Code sample 20 shows the relevant line from the previous code sample.

Using a Binomial Lattice

The *BinomialOptions ()* FSI algorithm uses a binomial lattice model (Cox, Ross and Rubenstein method) to price the European call option. At every step, the value of stock “S” can go up by u*S or down by v*S. This leads to a simple loop going over all leaf nodes in order to calculate the payoff at the expiry.

Code sample 21 shows the scalar algorithm.

```
DoubleVector<Shapes.S512Bit> sv = spec.broadcast(s);
DoubleVector<Shapes.S512Bit> vsdtVec = spec.broadcast (vsdt);
DoubleVector<Shapes.S512Bit> xv = spec.broadcast (x);
DoubleVector<Shapes.S512Bit> pdv = spec.broadcast (pdByr);
DoubleVector<Shapes.S512Bit> puv = spec.broadcast (puByr);
DoubleVector<Shapes.S512Bit> zv = spec.broadcast (0.0D);
IntVector<Shapes.S512Bit> inc = ispec.fromArray (new int [] {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13,
14, 15}, 0);
IntVector<Shapes.S512Bit> nSV = ispec.broadcast (numSteps);
```

Code sample 22. *Broadcasting scalar constants and/or function parameters outside the loop.*

```
2.0D * j – numSteps; //scalar form (implicit cast)
Vector<Double, Shapes.S512Bit> tv = jv.add (inc).cast (Double.class).mul (spec.broadcast (2.0D)).sub
(nSV.cast (Double.class));
```

Code sample 23. *Addressing the challenge of type casting.*

```
int j;
for (j = 0; (j + spec.length()) < STEPS_CACHE_SIZE; j += spec.length()) {
  IntVector<Shapes.S512Bit> jv = ispec.broadcast (j);
  Vector<Double, Shapes.S512Bit> tv = jv.add (inc).cast (Double.class).mul (spec.broadcast
(2.0D)).sub (nSV.cast (Double.class));
  DoubleVector<Shapes.S512Bit> pftVec = sv.mul (vsdtVec. mul (tv).exp ()).sub (xv);
  pftVec.blend (zv, pftVec.greaterThan (zv)).intoArray (stepsArray,j);
 }
for (; j < STEPS_CACHE_SIZE; j++) { //tail processing
   double profit = s * Math.exp (vsdt * (2.0D * j - numSteps)) - x;
   stepsArray[j] = profit > 0.0D? profit : 0.0D;
 }
```

Code sample 24. *Vector version of the first loop that computes profit.*

```
for (j = 0; j < numSteps; j++) {
  int k;
  for (k = 0; k + spec.length() < NUM_STEPS_ROUND; k += spec.length()) {
    DoubleVector<Shapes.S512Bit> sv0 = spec.fromArray (stepsArray, k);
    DoubleVector<Shapes.S512Bit> sv1 = spec.fromArray (stepsArray, k + 1);
     pdv.mul (sv1).add (puv.mul (sv0)).intoArray (stepsArray, k);
   //sv0 = pdv.fma (sv1, puv.mul (sv0)); sv0.intoArray (stepsArray, k);
         }
         for (; k < NUM_STEPS_ROUND; ++k) {
            stepsArray[k] = pdByr * stepsArray[k + 1] + puByr * stepsArray[k];
         }
      }
```

Code sample 26. *Expressing the second loop (that computes profit) in vector form, using fma ().*

As shown in our previous examples, we broadcast the scalar constants and/or function parameters outside the loop. See code sample 22.

Addressing the challenge of type casting

You can write a vector version of the algorithm over the number of nodes. In this case, one of the challenges of writing the vector code is type casting.

We use *cast* (*Double.class*) to explicitly type cast Vector<Integer, Shapes.S512Bit> to Vector<Double, S512Bit>. Here, our program uses *blend ()* for ternary operations on profit values. We store values greater than zero into *stepsArray*.

See code sample 23.

Code sample 24 shows a vector version of the first loop that computes profit.

The second loop can also be expressed in vector form. Moreover, you can write the expression using *fma ()*. Look at code sample 25.

You can see the Vector API implementation for these algorithms at https://software.intel.com/en-us/articles/vector-api-developer-program-for-java.

Conclusion

Big data applications, distributed deep learning programs, and artificial intelligence solutions can run directly on top of existing Spark or Apache Hadoop clusters, and can benefit from efficient scale out. The OpenJDK Project Panama enables data parallelism, by including a rich set of Vector API methods to enrich machine learning and deep learning support.

Using Vector API, you can boost the performance of several BLAS algorithms (up to Level 3 routines) by 3x to 4x when running them on Intel® AVX-enabled platforms. Using this API, you can now write your own vector algorithms in Java itself, to achieve higher performance and leverage the advanced SIMD features provided by modern CPUs.

Although the Vector API is available for use, it is an evolving project, and JVM support for Vector API is still in progress. Currently, Vector API provides a comprehensive set of methods for basic arithmetic and advanced vector operations. It is available for use under experimental Panama and JVM flags, and will be published soon once more progress is made on the implementation.

Where to get started with Vector API

You can download Vector API under the experimental flags in Project Panama. We suggest you download the latest Project Panama Vector API sources using Mercurial* source-control, by cloning http://hg.openjdk.java.net/panama/panama/. JVM flags for real-world use are expected to be published soon.

You can find Vector interface implementations in the vector-draft-spec directory, under com/oracle/java. Several vector API examples are available inside the src/test/java folder.

We recommend using the JetBrains IntelliJ IDEA* community edition as the integrated development environment (IDE) for related development. You can find detailed instructions for using IntelliJ IDEA at the Vector API developer Program webpage.

About the Author

Rahul Kandu is a senior software engineer in the Intel Software and Services Group (SSG). He focuses on Java runtime and SOC performance analysis and Hotspot compiler optimizations.

To learn more about using Vector API to write your own vector algorithms, visit the **Vector API developer program site**

Printed in USA XXXX/XXX/XXX/XX/XX Please Recycle XXXXXX-001US