JEP 338: Vector API (Incubator)

Authors	Vladimir Ivanov, Razvan Lupusoru, Paul Sandoz, Sandhya Viswanathan
Owner	Paul Sandoz
Type	Feature
Scope	JDK
Status	Closed / Delivered
Release	16
Component	hotspot / compiler
Discussion	panama dash dev at openjdk dot java dot net
Effort	M
Duration	M
Relates to	JEP 414: Vector API (Second Incubator)
Reviewed by	John Rose, Maurizio Cimadamore, Yang Zhang
Endorsed by	John Rose, Vladimir Kozlov
Created	2018/04/06 22:58
Updated	2021/08/28 00:15
Issue	8201271

Summary

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.

Goals

- *Clear and concise API:* The API shall be capable of clearly and concisely expressing a wide range of vector computations consisting of a sequence of vector operations often composed within loops and possibly with control flow. It should be possible to express a computation that is generic to vector size (or the number of lanes per vector) thus enabling such computations to be portable across hardware supporting different vector sizes (as detailed in the next goal).
- *Platform agnostic:* The API shall be architecture agnostic, enabling support for runtime implementations on multiple CPU architectures that support vector hardware instructions. As is usual in Java APIs, where platform optimization and portability conflict, the bias will be to making the Vector API portable, even if some platform-specific idioms cannot be directly expressed in portable code. The next goal of x64 and AArch64 performance is representative of appropriate performance goals on all platforms where Java is supported. The ARM Scalable Vector Extension (SVE) is of special interest in this regard to ensure the API can support this architecture, even though as of writing there are no known production hardware implementations.
- *Reliable runtime compilation and performance on x64 and AArch64 architectures:* The Java runtime, specifically the HotSpot C2 compiler, shall compile, on capable x64 architectures, a sequence of vector operations to a corresponding sequence of vector hardware instructions, such as those supported by Streaming SIMD Extensions (SSE) and Advanced Vector Extensions (AVX) extensions, thereby generating efficient and performant code. The programmer shall have confidence that the vector operations they express will reliably map closely to associated hardware vector instructions. The same shall also apply to capable ARM AArch64 architectures compiling to a sequence of vector hardware instructions supported by Neon.
- *Graceful degradation:* If a vector computation cannot be fully expressed at runtime as a sequence of hardware vector instructions, either because an architecture does not support some of the required instructions or because another CPU architecture is not supported, then the Vector API implementation shall degrade gracefully and still function. This may include issuing warnings to the developer if a vector computation cannot be sufficiently compiled to vector hardware instructions. On platforms without vectors, graceful degradation shall yield code competitive with manually-unrolled loops, where the unroll factor is the number of lanes in the selected vector.

Non-Goals

- It is not a goal to enhance the auto-vectorization support in HotSpot.
- It is not a goal for HotSpot to support vector hardware instructions on CPU architectures other than x64 and AArch64. Such support is left for later JEPs. However, it is important to state, as expressed in the goals, that the API must not rule out such implementations. Further, work performed may naturally leverage and extend existing abstractions in HotSpot for auto-vectorization vector support making such a task easier.
- It is not a goal to support the C1 compiler in this or future iterations. We expect the Graal compiler to be supported in future work.
- It is not a goal to support strict floating point calculations as defined by the Java strictfp keyword. The results of floating point operations performed on floating point scalars may differ from equivalent floating point operations performing on vectors of floating point scalars. However, this goal does not rule out options to express or control the desired precision or reproducibility of floating point vector computations.

Motivation

Vector computations consist of a sequence of operations on vectors. A vector comprises a (usually) fixed sequence of scalar values, where the scalar values correspond to the number of hardware-defined vector lanes. A binary operation applied to two vectors with the same number of lanes would, for each lane, apply the equivalent scalar operation on the corresponding two scalar values from each vector. This is commonly referred to as Single Instruction Multiple Data (SIMD).

Vector operations express a degree of parallelism that enables more work to be performed in a single CPU cycle and thus can result in significant performance gains. For example, given two vectors each covering a sequence of eight integers (eight lanes), then the two vectors can be added together using a single hardware instruction. The vector addition hardware instruction operates on sixteen integers, performing eight integer additions, in the time it would ordinarily take to operate on two integers, performing one integer addition.

HotSpot supports auto-vectorization where scalar operations are transformed into superword operations, which are then mapped to vector hardware instructions. The set of transformable scalar operations are limited and fragile to changes in the code shape. Furthermore, only a subset of available vector hardware instructions might be utilized limiting the performance of generated code.

A developer wishing to write scalar operations that are reliably transformed into superword operations needs to understand HotSpot's auto-vectorization support and its limitations to achieve reliable and sustainable performance.

In some cases it may not be possible for the developer to write scalar operations that are transformable. For example, HotSpot does not transform the simple scalar operations for calculating the hash code of an array (see the Arrays::hashCode method implementations in the JDK source code), nor can it auto-vectorize code to lexicographically compare two arrays (which is why an intrinsic was added to perform lexicographical comparison, see 8033148).

The Vector API aims to address these issues by providing a mechanism to write complex vector algorithms in Java, using pre-existing support in HotSpot for vectorization, but with a user model which makes vectorization far more predictable and robust. Hand-coded vector loops can express high-performance algorithms (such as vectorized hashCode or specialized array comparison) which an auto-vectorizer may never optimize. There are numerous domains where this explicitly vectorizing API may be applicable such as machine learning, linear algebra, cryptography, finance, and usages within the JDK itself.

Description

A vector will be represented by the abstract class Vector<E>. The type variable E corresponds to the boxed type of scalar primitive integral or floating point element types covered by the vector. A vector also has a *shape* which defines the size, in bits, of the vector. The shape of the vector will govern how an instance of Vector<E> is mapped to a vector hardware register when vector computations are compiled by the HotSpot C2 compiler (see later for a mapping from instances to x64 vector registers). The length of a vector (number of lanes or elements) will be the vector size divided by the element size.

The set of element types (E) supported will be Byte, Short, Integer, Long, Float and Double corresponding to the scalar primitive types byte, short, int, long, float and double, respectively.

The set of shapes supported will correspond to vector sizes of 64, 128, 256, and 512 bits. A shape corresponding to a size of 512 bits can pack bytes into 64 lanes or pack ints into 16 lanes, and a vector of such a shape can operate on 64 bytes at a time, or 16 ints at a time.

> *Note:* We believe that these simple shapes are generic enough to be useful on all platforms supporting the Vector API. However, as we experiment during the incubation of this JEP with future platforms, we may further modify the design of the shape parameter. Such work is not in the early scope of this JEP, but these possibilities partly inform the present role of shapes in the Vector API. See the Future Work section, below.

The combination of element type and shape determines the vector's species, represented by VectorSpecies<E>

An instance of Vector<E> is immutable and is a value-based type that retains, by default, object identity invariants (see later for relaxation of these invariants).

Operations on vectors can be classified as lane-wise and cross-lane. Lane-wise operations can be further classified as unary, binary, ternary, and comparison. Cross-lane operations can be classified as permutation, conversion, and reduction. To reduce the surface of the API, we will define collective methods for each class of operation which then take an operator as input. The supported operators are instances of Operator class and are defined as static final fields in the VectorOperators class. Some common operations (e.g., add, mul), called full-service operations, will have dedicated methods which can be used in place of the generic methods.

Certain operations on vectors, such lane-wise cast and reinterpret, can be said to be inherently *shape-changing*. Having shape-changing operations in a vector computation could have unintended effects on portability and performance. For this reason, wherever applicable, the API will define an additional shape-invariant flavor of such an operation. Users are encouraged to write shape-invariant code using the shape-invariant flavor of operations. Additionally, shape-changing operations will be clearly called out in the Javadoc.

Vector<E> declares a set of methods for common vector operations supported by all element types. To support operations specific to an element type there are six abstract sub-classes of Vector<E>, one for each supported element type: ByteVector, ShortVector, IntVector, LongVector, FloatVector, and DoubleVector. These sub-classes define additional operations which are bound to the element type since the method signature refers to the element type (or the equivalent array type), such as reduction operations (e.g., sum all elements to a scalar value) or storing the vector elements to an array. They also define additional full-service operations that are specific to the integral sub-types such as bitwise operations (e.g., logical or), and operations specific to the floating point types, such as mathematical operations (e.g., transcendental functions such as pow()).

These classes are further extended by concrete sub-classes defined for different shapes (size) of Vectors.

The concrete sub-classes are non-public since there is no need to provide operations specific to the type and shape. This reduces the API surface to a sum of concerns rather than a product. As a result, instances of concrete Vector classes cannot be constructed directly. Instead, instances are obtained via factories methods defined in the base Vector<E> class and its type-specific sub-classes. These methods take as input the species of the desired vector instance. The factory methods provide different ways to obtain vector instances, such as the vector instance whose elements are initiated to default values (the zero vector), or a vector from an array, in addition to providing the canonical support for converting between vectors of different types or shapes (e.g., casting).

To support control flow, relevant vector operations will optionally accept masks represented by the public abstract class VectorMask<E>. Each element in a mask, a boolean value or bit, corresponds to a vector lane. When a mask is an input to an operation it governs whether the operation is applied to each lane; the operation is applied if the mask bit for the lane is set (is true). Alternative behavior occurs if the mask bit is not set (is false). Similar to vectors, instances of VectorMask<E> are instances of (private) concrete sub-class defined for each element type and length combination. The instance of VectorMask<E> used in an operation should have the same type and length as the instance(s) of Vector<E> involved in the operation. Comparison operations produce masks, which can then be input to other operations to selectively disable the operation on certain lanes and thereby emulate flow control. Another way for creating masks is using static factory methods in VectorMask<E>.

We anticipate that masks will likely play an important role in the development of vector computations that are generic to shape. (This expectation is based on the central importance of predicate registers, the equivalent of masks, in the ARM Scalable Vector Extensions as well as in Intel's AVX-512.)

Example

Here is a simple scalar computation over elements of arrays:

```
void scalarComputation(float[] a, float[] b, float[] c) {
   for (int i = 0; i < a.length; i++) {
        c[i] = (a[i] * a[i] + b[i] * b[i]) * -1.0f;
   }
}
```

(We assume that the array arguments will be of the same size.)

An explicit way to implement the equivalent vector computation using the Vector API is as follows:

```
// Example 1

static final VectorSpecies<Float> SPECIES = FloatVector.SPECIES_256;

void vectorComputation(float[] a, float[] b, float[] c) {

    for (int i = 0; i < a.length; i += SPECIES.length()) {
        var m = SPECIES.indexInRange(i, a.length);
                // FloatVector va, vb, vc;
        var va = FloatVector.fromArray(SPECIES, a, i, m);
        var vb = FloatVector.fromArray(SPECIES, b, i, m);
        var vc = va.mul(va).
                    add(vb.mul(vb)).
                    neg();
        vc.intoArray(c, i, m);
    }
}
```

In this example, a species for 256-bit wide vectors of floats is obtained from FloatVector. The species is stored in a static final field so the runtime compiler will treat the field's value as a constant and therefore be able to better optimize the vector computation.

The vector computation features a main loop kernel iterating over the arrays in strides of vector length (i.e., the species length). The static method fromArray() loads float vectors of the given species from arrays a and b at the corresponding index. Then the operations are performed, fluently, and finally the result is stored into array c.

We use masks, generated by indexInRange(), to prevent reading/writing past the array length. The first floor(a.length / SPECIES.length()) iterations will have a mask with all lanes set. Only the final iteration, if a.length is not a multiple of SPECIES.length(), will have a mask with first a.length % SPECIES.length() lanes set.

Since a mask is used in all iterations, the above implementation may not achieve optimal performance for large array lengths. The same computation can be implemented without masks as follows:

```
// Example 2

static final VectorSpecies<Float> SPECIES = FloatVector.SPECIES_256;

void vectorComputation(float[] a, float[] b, float[] c) {
    int i = 0;
    int upperBound = SPECIES.loopBound(a.length);
    for (; i < upperBound; i += SPECIES.length()) {
        // FloatVector va, vb, vc;
        var va = FloatVector.fromArray(SPECIES, a, i);
        var vb = FloatVector.fromArray(SPECIES, b, i);
        var vc = va.mul(va).
                    add(vb.mul(vb)).
                    neg();
        vc.intoArray(c, i);
    }

    for (; i < a.length; i++) {
        c[i] = (a[i] * a[i] + b[i] * b[i]) * -1.0f;
    }
}
```

The *tail* elements, the length of which is smaller than the species length, are processed using the scalar computation after the vector computation. Another way to process the tail elements is to use a single masked vector computation.

When operating on large arrays, the implementation above achieves optimal performance.

For this second example, the HotSpot compiler should generate machine code similar to the following on an Intel x64 processor supporting AVX:

```
0.43%  / |  0x0000000113d43890: vmovdqu 0x10(%r8,%rbx,4),%ymm0
  7.38%  | |  0x0000000113d43897: vmovdqu 0x10(%r10,%rbx,4),%ymm1
  8.70%  | |  0x0000000113d4389e: vmulps %ymm0,%ymm0,%ymm0
  5.60%  | |  0x0000000113d438a2: vmulps %ymm1,%ymm1,%ymm1
 13.16%  | |  0x0000000113d438a6: vaddps %ymm0,%ymm1,%ymm0
 21.86%  | |  0x0000000113d438aa: vxorps -0x7ad76b2(%rip),%ymm0,%ymm0
  7.66%  | |  0x0000000113d438b2: vmovdqu %ymm0,0x10(%r9,%rbx,4)
 26.20%  | |  0x0000000113d438b9: add    $0x8,%ebx
  6.44%  | |  0x0000000113d438bc: cmp    %r11d,%ebx
         \ |  0x0000000113d438bf: jl     0x0000000113d43890
```

This is actual output from a JMH micro-benchmark for the example code under test using a prototype of the Vector API and implementation (the vectorIntrinsics branch of Project Panama's development repository). This shows the hot areas of C2-generated machine code. There is a clear translation to vector registers and vector hardware instructions. (Loop unrolling was disabled to make the translation clearer, otherwise HotSpot should be able to unroll using existing C2 loop optimization techniques.) All Java object allocations are elided.

It is an important goal to support more complex non-trivial vector computations that translate clearly into generated machine code.

There are, however, a few issues with this particular vector computation:

1. The loop is hardcoded to a concrete vector shape, so the computation cannot adapt dynamically to a maximal shape supported by the architecture, which may be smaller or larger than 256 bits. Therefore the code is less portable and may be less performant.
2. Calculation of the loop upper bounds, although simple here, can be a common source of programming error.
3. A scalar loop is required at the end, duplicating code.

We will address the first two issues in this JEP. A preferred species can be obtained whose shape is optimal for the current architecture, the vector computation can then be written with a generic shape, and a method on the species can round down the array length, for example:

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

void vectorComputation(float[] a, float[] b, float[] c,
        VectorSpecies<Float> species) {
    int i = 0;
    int upperBound = species.loopBound(a.length);
    for (; i < upperBound; i += species.length()) {
        //FloatVector va, vb, vc;
        var va = FloatVector.fromArray(species, a, i);
        var vb = FloatVector.fromArray(species, b, i);
        var vc = va.mul(va).
                    add(vb.mul(vb)).
                    neg();
        vc.intoArray(c, i);
    }

    for (; i < a.length; i++) {
        c[i] = (a[i] * a[i] + b[i] * b[i]) * -1.0f;
    }
}

vectorComputation(a, b, c, SPECIES);
```

The third issue will not be fully addressed by this JEP and will be the subject of future work. As shown in the first example, you can use masks to implement vector computation without tail processing. We anticipate that such masked loops will work well for a range of architectures, including x64 and ARM, but will require additional runtime compiler support to generate maximally efficient code. Such work on masked loops, though important, is beyond the scope of this JEP.

HotSpot C2 compiler details

The Vector API has two implementations in order to achieve this JEP's goals. The first implements operations in Java, thus it is functional but not optimal. The second makes intrinsic, for the HotSpot C2 compiler, those operations with special treatment for Vector API types. This allows for proper translation to hardware registers and instructions for the case where architecture support and implementation for translation exists.

To avoid an explosion of intrinsics added to C2, a set of intrinsics will be defined that correspond to operation kinds such as binary, unary, comparison, and so on, where constant arguments are passed describing operation specifics. Approximately twenty new intrinsics will be needed to support the intrinsification of all parts of the API.

Vector instances are value-based, i.e., morally values where identity-sensitive operations should be avoided. Further, although vector instances are abstractly composed of elements in lanes, those elements are not scalarized by C2. The vector value is treated as a whole unit, like int or long, that maps to a hardware vector register of the appropriate size. Inline types will require some related enhancements to ensure that a vector value is treat as a whole unit.

Until inline types are available, Vector instances will be treated specially by C2 to overcome limitations in escape analysis and avoid boxing. As such, identity sensitive operations on vectors should be avoided.

Future Work

The Vector API will benefit significantly from value types once ready (see Project Valhalla). Instances of a Vector<E> can be values, whose concrete classes are inline types. This will make it easier to optimize and express vector computations. Sub-types of Vector<E> for specific types, such as IntVector, may not be required with generic specialization over inline types and type-specific method declaration.

Therefore, a future version of the Vector API will make use of inline types and enhanced generics, as noted above. As a result, we will incubate the API over multiple releases of the JDK and will adapt as inlines types become available.

We will enhance the API to load and store vectors using features of JEP 370 Foreign-Memory Access API, when that API transitions from an incubating API. Further, memory layouts to describe vector species may prove useful, for example to stride over a memory segment comprised of elements.

We anticipate enhancing the implementation in the following ways:

- Include support for vectorized transcendental operations (such as logarithm, and the trigonometric functions),
- Improve the optimization of loops containing vectorized code,
- Optimize masked vector operations on supporting platforms, and
- Make adjustments for large vector sizes (e.g., as supported by ARM SVE).

Performance work will be ongoing as we make incremental improvements to the implementation.

Alternatives

HotSpot's auto-vectorization is an alternative approach, but it would require significant work. It would, moreover, likely still be fragile and limited compared to using the Vector API, since auto-vectorization with complex control flow is very hard to perform.

In general, and even after decades of research (especially for FORTRAN and C array loops), it seems that auto-vectorization of scalar code is not a reliable tactic for optimizing ad-hoc user-written loops unless the user pays unusually careful attention to unwritten contracts about exactly which loops a compiler is prepared to auto-vectorize. It's too easy to write a loop that fails to auto-vectorize, for a reason that the optimizer but no human reader can detect. Years of work on auto-vectorization, even in HotSpot, have left us with lots of optimization machinery that works only on special occasions. We want to enjoy the use of this machinery more often!

Testing

We will develop combinatorial unit tests to ensure coverage for all operations, for all supported types and shapes, over various data sets.

We will also develop performance tests to ensure that performance goals are met and vector computations map efficiently to vector hardware instructions. This will likely consist of JMH micro-benchmarks, but more realistic examples of useful algorithms will also be required. Such tests may initially reside in a project specific repository. Curation is likely required before integration into the main repository given the proportion of tests and how they are generated.

As a backup to performance tests, we may create white-box tests to force the JIT to report to us that vector API source code did, in fact, trigger vectorization.

Risks and Assumptions

There is a risk that the API will be biased to the SIMD functionality supported on x64 architectures but this is mitigated with support for AArch64. This applies mainly to the explicitly fixed set of supported shapes, which bias against coding algorithms in a shape-generic fashion. We consider the majority of other operations of the Vector API to bias toward portable algorithms. To mitigate that risk we will take other architectures into account, specifically the ARM Scalar Vector Extension architecture whose programming model adjusts dynamically to the singular fixed shape supported by the hardware. We welcome and encourage OpenJDK contributors working on the ARM-specific areas of HotSpot to participate in this effort.

The Vector API uses box types (such as Integer) as proxies for primitive types (such as int). This decision is forced by the current limitations of Java generics, which are hostile to primitive types. When Project Vahalla eventually introduces more capable generics the current decision will seem awkward, and may need changing. We assume that such changes will be possible without excessive backward incompatibility.