```
OpenJDK
                   JEP 417: Vector API (Third Incubator)
                              Owner Paul Sandoz
Contributing
Sponsoring
                                Type Feature
Developers' Guide
                              Scope JDK
Vulnerabilities
                              Status Closed / Delivered
JDK GA/EA Builds
Mailing lists
                             Release 18
Wiki · IRC
                         Component core-libs
Bylaws · Census
                          Discussion panama dash dev at openjdk dot java dot net
Legal
                               Effort L
Workshop
                            Duration M
JEP Process
                          Relates to JEP 426: Vector API (Fourth Incubator)
Source code
Mercurial
                                      JEP 414: Vector API (Second Incubator)
GitHub
                        Reviewed by John Rose, Vladimir Ivanov, Vladimir Kozlov
Tools
                        Endorsed by John Rose, Vladimir Kozlov
jtreg harness
                             Created 2021/06/24 16:44
Groups
                            Updated 2023/05/12 15:35
(overview)
Adoption
                               Issue 8269306
Build
Client Libraries
                    Summary
Compatibility &
 Specification
 Review
                    Introduce an API to express vector computations that reliably compile at runtime to
Compiler
                    optimal vector instructions on supported CPU architectures, thus achieving
Conformance
Core Libraries
                    performance superior to equivalent scalar computations.
Governing Board
HotSpot
IDE Tooling & Support
                    History
Internationalization
JMX
                   The Vector API was first proposed by JEP 338 and integrated into Java 16 as an
Members
                    incubating API. A second round of incubation was proposed by JEP 414 and
Networking
Porters
                    integrated into Java 17.
Quality
Security
                    We propose here to incorporate enhancements in response to feedback as well as
Serviceability
                    performance improvements and other significant implementation enhancements.
Vulnerability
Web
                    We include the following notable changes:
Projects
(overview, archive)

    Support the ARM Scalar Vector Extension (SVE) platform.

Amber
Babylon

    Improve the performance of vector operations that accept masks on

CRaC
                        architectures that support masking in hardware.
Caciocavallo
Closures
Code Tools
                    Goals
Coin
Common VM

    Clear and concise API — The API should be capable of clearly and concisely

 Interface
Compiler Grammar
                        expressing a wide range of vector computations consisting of sequences of
Detroit
                        vector operations composed within loops and possibly with control flow. It
Developers' Guide
Device I/O
                        should be possible to express a computation that is generic with respect to
Duke
                        vector size, or the number of lanes per vector, thus enabling such
Galahad
Graal
                        computations to be portable across hardware supporting different vector
IcedTea
                        sizes.
JDK 7
JDK 8

    Platform agnostic — The API should be CPU architecture agnostic, enabling

JDK 8 Updates
IDK 9
                        implementations on multiple architectures supporting vector instructions.
JDK (..., 21, 22, 23)
                        As is usual in Java APIs, where platform optimization and portability conflict
JDK Updates
JavaDoc.Next
                        then the bias will be toward making the API portable, even if that results in
Jigsaw
                        some platform-specific idioms not being expressible in portable code.
Kona
Kulla

    Reliable runtime compilation and performance on x64 and AArch64

Lambda
Lanai
                        architectures — On capable x64 architectures the Java runtime, specifically
Leyden
Lilliput
                        the HotSpot C2 compiler, should compile vector operations to
Locale Enhancement
                        corresponding efficient and performant vector instructions, such as those
Loom
                        supported by Streaming SIMD Extensions (SSE) and Advanced Vector
Memory Model
 Update
                        Extensions (AVX). Developers should have confidence that the vector
Metropolis
                        operations they express will reliably map closely to relevant vector
Mission Control
Multi-Language VM
                        instructions. On capable ARM AArch64 architectures C2 will, similarly,
Nashorn
                        compile vector operations to the vector instructions supported by NEON
New I/O
OpenJFX
                        and SVE.
Panama
Penrose

    Graceful degradation — Sometimes a vector computation cannot be fully

Port: AArch32
Port: AArch64
                        expressed at runtime as a sequence of vector instructions, perhaps
Port: BSD
                        because the architecture does not support some of the required
Port: Haiku
Port: Mac OS X
                        instructions. In such cases the Vector API implementation should degrade
Port: MIPS
                        gracefully and still function. This may involve issuing warnings if a vector
Port: Mobile
Port: PowerPC/AIX
                        computation cannot be efficiently compiled to vector instructions. On
Port: RISC-V
                        platforms without vectors, graceful degradation will yield code competitive
Port: s390x
Portola
                        with manually-unrolled loops, where the unroll factor is the number of
SCTP
                        lanes in the selected vector.
Shenandoah
Skara
Sumatra
                    Non-Goals
Tiered Attribution
                     It is not a goal to enhance the existing auto-vectorization algorithm in
Type Annotations
Valhalla
                        HotSpot.
Verona
VisualVM

    It is not a goal to support vector instructions on CPU architectures other

Wakefield
                        than x64 and AArch64. However it is important to state, as expressed in
Zero
ZGC
                        the goals, that the API must not rule out such implementations.
ORACLE
                     It is not a goal to support the C1 compiler.
                     It is not a goal to guarantee support for strict floating point calculations as
                        is required by the Java platform for scalar operations. The results of floating
                        point operations performed on floating point scalars may differ from
                        equivalent floating point operations performed on vectors of floating point
                        scalars. Any deviations will be clearly documented. This non-goal does not
                        rule out options to express or control the desired precision or
                        reproducibility of floating point vector computations.
                    Motivation
                    A vector computation consists 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 containing a sequence of eight
                    integers (i.e., eight lanes), the two vectors can be added together using a single
                    hardware instruction. The vector addition 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 already supports auto-vectorization, which transforms scalar operations
                    into superword operations which are then mapped to vector instructions. The set of
                    transformable scalar operations is limited, and also fragile with respect to changes
                    in code shape. Furthermore, only a subset of the available vector instructions
                    might be utilized, limiting the performance of generated code.
                    Today, a developer who wishes to write scalar operations that are reliably
                    transformed into superword operations needs to understand HotSpot's auto-
                    vectorization algorithm and its limitations in order to achieve reliable and
                    sustainable performance. In some cases it may not be possible 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 (thus the
                    Arrays::hashCode methods), nor can it auto-vectorize code to lexicographically
                    compare two arrays (thus we added an intrinsic for lexicographic comparison).
                   The Vector API aims to improve the situation by providing a way to write complex
                    vector algorithms in Java, using the existing HotSpot auto-vectorizer 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 comparisons, which an auto-vectorizer may never
                    optimize. Numerous domains can benefit from this explicit vector API including
                    machine learning, linear algebra, cryptography, finance, and code within the JDK
                    itself.
                    Description
                   A vector is represented by the abstract class Vector<E>. The type variable E is
                    instantiated as the boxed type of the 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 a vector governs how an instance of
                    Vector<E> is mapped to a hardware vector register when vector computations are
                    compiled by the HotSpot C2 compiler. The length of a vector, i.e., the number of
                   lanes or elements, is the vector size divided by the element size.
                   The set of element types (E) supported is 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 correspond to vector sizes of 64, 128, 256, and 512
                    bits, as well as max bits. A 512-bit shape 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. A max-bits shape supports the maximum vector size of the
                    current architecture. This enables support for the ARM SVE platform, where
                    platform implementations can support any fixed size ranging from 128 to 2048
                    bits, in increments of 128 bits.
                    We believe that these simple shapes are generic enough to be useful on all
                    relevant platforms. However, as we experiment with future platforms during the
                    incubation of this API we may further modify the design of the shape parameter.
                    Such work is not in the early scope of this project, but these possibilities partly
                    inform the present role of shapes in the Vector API. (For further discussion see the
                   future work section, below.)
                   The combination of element type and shape determines a vector's species,
                    represented by VectorSpecies<E>.
                    Operations on vectors are classified as either lane-wise or cross-lane.

    A lane-wise operation applies a scalar operator, such as addition, to each

                        lane of one or more vectors in parallel. A lane-wise operation usually, but
                        not always, produces a vector of the same length and shape. Lane-wise
                        operations are further classified as unary, binary, ternary, test, or
                        conversion operations.

    A cross-lane operation applies an operation across an entire vector. A

                        cross-lane operation produces either a scalar or a vector of possibly a
                        different shape. Cross-lane operations are further classified as permutation
                        or reduction operations.
                   To reduce the surface of the API, we define collective methods for each class of
                    operation. These methods take operator constants as input; these constants are
                    instances of the VectorOperator.Operator class and are defined in static final
                   fields in the VectorOperators class. For convenience we define dedicated
                    methods, which can be used in place of the generic methods, for some common
                   full-service operations such as addition and multiplication.
                    Certain operations on vectors, such conversion and reinterpretation, are inherently
                   shape-changing; i.e., they produce vectors whose shapes are different from the
                    shapes of their inputs. Shape-changing operations in a vector computation can
                    negatively impact portability and performance. For this reason the API defines a
                    shape-invariant flavor of each shape-changing operation when applicable. For best
                    performance, developers should write shape-invariant code using shape-invariant
                    operations insofar as possible. Shape-changing operations are identified as such in
                   the API specification.
                   The Vector<E> class declares a set of methods for common vector operations
                   supported by all element types. For operations specific to an element type there
                    are six abstract subclasses of Vector<E>, one for each supported element type:
                    ByteVector, ShortVector, IntVector, LongVector, FloatVector, and
                   DoubleVector. These type-specific subclasses define additional operations that are
                    bound to the element type since the method signature refers either to the element
                    type or to the related array type. Examples of such operations include reduction
                    (e.g., summing all lanes to a scalar value), and copying a vector's elements into an
                    array. These subclasses also define additional full-service operations specific to the
                   integral subtypes (e.g., bitwise operations such as logical or), as well as operations
                   specific to the floating point types (e.g., transcendental mathematical functions
                   such as exponentiation).
                   As an implementation matter, these type-specific subclasses of Vector<E> are
                   further extended by concrete subclasses for different vector shapes. These
                    concrete subclasses are not public since there is no need to provide operations
                    specific to types and shapes. This reduces the API surface to a sum of concerns
                    rather than a product. Instances of concrete Vector classes are obtained via
                   factory methods defined in the base Vector<E> class and its type-specific
                    subclasses. These factories take as input the species of the desired vector instance
                    and produce various kinds of instances, for example the vector instance whose
                    elements are default values (i.e., the zero vector), or a vector instance initialized
                   from a given array.
                   To support control flow, some vector operations optionally accept masks
                    represented by the public abstract class VectorMask<E>. Each element in a mask
                    is a boolean value corresponding to a vector lane. A mask selects the lanes to
                    which an operation is applied: It is applied if the mask element for the lane is true,
                    and some alternative action is taken if the mask is false.
                    Similar to vectors, instances of VectorMask<E> are instances of non-public
                    concrete subclasses 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 vector instances involved in the operation. Vector comparison
                    operations produce masks, which can then be used as input to other operations to
                    selectively operate on certain lanes and thereby emulate flow control. Masks can
                    also be created using static factory methods in the VectorMask<E> class.
                    We anticipate that masks will play an important role in the development of vector
                    computations that are generic with respect to shape. This expectation is based on
                    the central importance of predicate registers, the equivalent of masks, in the ARM
                    Scalable Vector Extensions and in Intel's AVX-512.
                    On such platforms an instance of VectorMask<E> is mapped to a predicate
                    register, and a mask-accepting operation is compiled to a predicate-register-
                    accepting vector instruction. On platforms that don't support predicate registers, a
                    less efficient approach is applied: An instance of VectorMask<E>is mapped, where
                    possible, to a compatible vector register, and in general a mask-accepting
                    operation is composed of the equivalent unmasked operation and a blend
                    operation.
                   To support cross-lane permutation operations, some vector operations accept
                    shuffles represented by the public abstract class VectorShuffle<E>. Each element
                   in a shuffle is an int value corresponding to a lane index. A shuffle is a mapping of
                    lane indexes, describing the movement of lane elements from a given vector to a
                    result vector.
                    Similar to vectors and masks, instances of VectorShuffle<E> are instances of non-
                    public concrete subclasses defined for each element type and length combination.
                   The instance of VectorShuffle<E> used in an operation should have the same
                   type and length as the vector instances involved in the operation.
                   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 are of the same length.)
                    Here is an equivalent vector computation, using the Vector API:
                        static final VectorSpecies<Float> SPECIES = FloatVector.SPECIES PREFERRED;
                        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;
                             }
                        }
                    To start, we obtain a preferred species whose shape is optimal for the current
                    architecture from FloatVector. We store it in a static final field so that the
                    runtime compiler treats the value as constant and can therefore better optimize
                    the vector computation. The main loop then iterates over the input arrays in
                    strides of the vector length, i.e., the species length. It loads float vectors of the
                    given species from arrays a and b at the corresponding index, fluently performs the
                    arithmetic operations, and then stores the result into array c. If any array elements
                    are left over after the last iteration then the results for those tail elements are
                    computed with an ordinary scalar loop.
                   This implementation achieves optimal performance on large arrays. The HotSpot
                    C2 compiler generates machine code similar to the following on an Intel x64
                    processor supporting AVX:
                                      0x000000113d43890: vmovdqu 0x10(%r8,%rbx,4),%ymm0
                        0.43% /
                          7.38%
                                        0x000000113d43897: vmovdqu 0x10(%r10,%rbx,4),%ymm1
                          8.70%
                                        0x000000113d4389e: vmulps %ymm0,%ymm0,%ymm0
                          5.60%
                                        0x000000113d438a2: vmulps %ymm1,%ymm1,%ymm1
                         13.16%
                                        0x000000113d438a6: vaddps %ymm0,%ymm1,%ymm0
                         21.86%
                                        0x000000113d438aa: vxorps -0x7ad76b2(%rip),%ymm0,%ymm0
                          7.66%
                                        0x000000113d438b2: vmovdqu %ymm0,0x10(%r9,%rbx,4)
                         26.20%
                                        0x0000000113d438b9: add
                                                                         $0x8,%ebx
                          6.44%
                                                                        %r11d,%ebx
                                        0x000000113d438bc: cmp
                                        0x0000000113d438bf: jl
                                                                         0x0000000113d43890
                   This is the output of a JMH micro-benchmark for the above code using the
```

```
branch of Project Panama's development repository. These hot areas of generated
machine code show a clear translation to vector registers and vector instructions.
We disabled loop unrolling in order to make the translation clearer; otherwise,
HotSpot would unroll this code using existing C2 loop optimizations. All Java object
allocations are elided.
On platforms supporting predicate registers, the example above could be written
more simply, without the scalar loop to process the tail elements, while still
achieving optimal performance:
   void vectorComputation(float[] a, float[] b, float[] c) {
        for (int i = 0; i < a.length; i += SPECIES.length()) {
            // VectorMask<Float> m;
            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 the loop body we obtain a loop dependent mask for input to the load and store operations. When i < SPECIES.loopBound(a.length) the mask, m, declares all

SPECIES.loopBound(a.length) <= i < a.length and (a.length - i) <=SPECIES.length(), the mask may declare a suffix of unset lanes. The load and store operations will not throw out-of-bounds exceptions since the mask prevents

We would prefer that developers write in the above style for all supported

could be enhanced to transform the loop, peeling off the last iteration and removing the mask from the loop body. This remains an area for further

appropriate hardware registers and vector instructions when available.

and so on, which take a parameter describing the specific operation to be

To avoid an explosion of C2 intrinsics we define generalized intrinsics

platforms and achieve optimal performance, but today on platforms without

predicate registers the above approach is not optimal. In theory the C2 compiler

The Vector API has two implementations. The first implements operations in Java, thus it is functional but not optimal. The second defines intrinsic vector operations for the HotSpot C2 run-time compiler so that it can compile vector computations to

corresponding to the various kinds of operations such as unary, binary, conversion,

performed. Approximately twenty new intrinsics support the intrinsification of the

lanes are set. For the last iteration of the loop, when

access to the array beyond its length.

}

investigation.

entire API.

Run-time compilation

prototype of the Vector API and implementation found on the vectorIntrinsics

```
We expect ultimately to declare vector classes as primitive classes, as proposed by
Project Valhalla in JEP 401 (Primitive Objects). In the meantime Vector<E> and its
subclasses are considered value-based classes, so identity-sensitive operations on
their instances should be avoided. Although vector instances are abstractly
composed of elements in lanes, those elements are not scalarized by C2 - a
vector's value is treated as a whole unit, like an int or a long, that maps to a
vector register of the appropriate size. Vector instances are treated specially by C2
in order to overcome limitations in escape analysis and avoid boxing.
Intel SVML intrinsics for transcendental operations
The Vector API supports transcendental and trigonometric lanewise operations on
floating point vectors. On x64 we leverage the Intel Short Vector Math Library
(SVML) to provide optimized intrinsic implementations for such operations. The
```

intrinsic operations have the same numerical properties as the corresponding

The assembly source files for SVML operations are in the source code of the jdk.incubator.vector module, under OS-specific directories. The JDK build

specific shared library. This library is fairly large, weighing in at just under a megabyte. If a JDK image, built via jlink, omits the jdk.incubator.vector

module then the SVML library will not be copied into the image.

process compiles these source files for the target operating system into an SVML-

The implementation only supports Linux and Windows at this time. We will consider macOS support later, since it is a non-trivial amount of work to provide assembly

scalar operations defined in java.lang.Math.

source files with the required directives.

given the potential performance gains.

Future work

elements.

**Alternatives** 

incrementally over time.

The HotSpot runtime will attempt to load the SVML library and, if present, bind the operations in the SVML library to named stub routines. The C2 compiler generates code that calls the appropriate stub routine based on the operation and vector species (i.e., element type and shape). In the future, if Project Panama expands its support of native calling conventions to support vector values then it may be possible for the Vector API implementation to load the SVML library from an external source. If there is no performance impact with this approach then it would no longer be necessary to include SVML in source

form and build it into the JDK. Until then we deem the above approach acceptable,

primitive classes. We expect, further, to leverage Project Valhalla's generic specialization of primitive classes so that instances of Vector<E> can be

primitive values whose concrete types are primitive types. This will make it

incubation. Memory layouts that describe vector species may prove useful,

easier to optimize and express vector computations. Subtypes of Vector<E> for specific types, such as IntVector, might not be required once we have generic specialization over primitive classes. We intend to incubate the API over multiple releases and adapt it as primitive classes and related facilities become available. We intend to enhance the API to load and store vectors using JEP 412 (Foreign Function & Memory API) when that API transitions out of

for example to stride over a memory segment comprised of vector

 We anticipate enhancing the implementation to improve the optimization of loops containing vectorized code, and generally improve performance

As mentioned above, we expect ultimately to declare vector classes as

 We also anticipate enhancing the combinatorial units tests to assert that C2 generates vector hardware instructions. The unit tests currently assume, without verification, that vector hardware instructions will be generated via sufficient repeated execution. We will explore the use of C2's IR Test Framework to assert, cross-platform, that vector nodes are present in the IR graph (for example, using regex matching). If this approach is problematic we may explore a rudimentary approach and use of the nonproduct -XX:+TraceNewVectors flag to print vector nodes.

HotSpot's auto-vectorization is an alternative approach, but it would require

Vector API, since auto-vectorization with complex control flow is very hard to

significant work. It would, moreover, still be fragile and limited compared to the

- perform. In general, 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 is too easy to write a loop that fails to auto-vectorize, for a reason that 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**

all supported types and shapes, over various data sets.

proportion of tests and the manner in which they are generated.

There is a risk that the API will be biased to the SIMD functionality

We will develop combinatorial unit tests to ensure coverage for all operations, for

## We will also develop performance tests to ensure that performance goals are met and vector computations map efficiently to vector 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

Risks and Assumptions

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.

supported on x64 architectures, but this is mitigated with support for

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

© 2024 Oracle Corporation and/or its affiliates Terms of Use · License: GPLv2 · Privacy · Trademarks