Module jdk.incubator.vector

Package jdk.incubator.vector

`package jdk.incubator.vector`

Incubating Feature. Will be removed in a future release.

This package provides classes to express vector computations that, given suitable hardware and runtime ability, are accelerated using vector hardware instructions.

A *vector* is a sequence of a fixed number of *lanes*, all of some fixed *element type* such as `byte`, `long`, or `float`. Each lane contains an independent value of the element type. Operations on vectors are typically *lane-wise*, distributing some scalar operator (such as addition) across the lanes of the participating vectors, usually generating a vector result whose lanes contain the various scalar results. When run on a supporting platform, lane-wise operations can be executed in parallel by the hardware. This style of parallelism is called *Single Instruction Multiple Data* (SIMD) parallelism.

In the SIMD style of programming, most of the operations within a vector lane are unconditional, but the effect of conditional execution may be achieved using *masked operations* such as `blend()`, under the control of an associated `VectorMask`. Data motion other than strictly lane-wise flow is achieved using *cross-lane* operations, often under the control of an associated `VectorShuffle`. Lane data and/or whole vectors can be reformatted using various kinds of lane-wise conversions, and byte-wise reformatting reinterpretations, often under the control of a reflective `VectorSpecies` object which selects an alternative vector format different from that of the input vector.

`Vector<E>` declares a set of vector operations (methods) that are common to all element types. These common operations include generic access to lane values, data selection and movement, reformatting, and certain arithmetic and logical operations (such as addition or comparison) that are common to all primitive types.

Public subtypes of `Vector` correspond to specific element types. These declare further operations that are specific to that element type, including unboxed access to lane values, bitwise operations on values of integral element types, or transcendental operations on values of floating point element types.

Some lane-wise operations, such as the `add` operator, are defined as a full-service named operation, where a corresponding method on `Vector` comes in masked and unmasked overloadings, and (in subclasses) also comes in covariant overrides (returning the subclass) and additional scalar-broadcast overloadings (both masked and unmasked). Other lane-wise operations, such as the `min` operator, are defined as a partially serviced (not a full-service) named operation, where a corresponding method on `Vector` and/or a subclass provide some but all possible overloadings and overrides (commonly the unmasked varient with scalar-broadcast overloadings). Finally, all lane-wise operations (those named as previously described, or otherwise unnamed method-wise) have a corresponding `operator token` declared as a static constant on `VectorOperators`. Each operator token defines a symbolic Java expression for the operation, such as `a + b` for the `ADD` operator token. General lane-wise operation-token accepting methods, such as for a unary lane-wise operation, are provided on `Vector` and come in the same variants as a full-service named operation.

This package contains a public subtype of `Vector` corresponding to each supported element type: `ByteVector`, `ShortVector`, `IntVector`, `LongVector`, `FloatVector`, and `DoubleVector`.

Here is an example of multiplying elements of two float arrays `a` and `b` using vector computation and storing result in array `c`.

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

void vectorMultiply(float[] a, float[] b, float[] c) {
  // It is assumed array arguments are of the same size
  for (int i = 0; i < a.length; i += SPECIES.length()) {
        VectorMask<Float> m = SPECIES.indexInRange(i, a.length);
        FloatVector va = FloatVector.fromArray(SPECIES, a, i, m);
        FloatVector vb = FloatVector.fromArray(SPECIES, b, i, m);
        FloatVector vc = va.mul(vb)
        vc.intoArray(c, i, m);
  }
}
```

In the above example, we use masks, generated by `indexInRange()`, to prevent reading/writing past the array length. The first `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 the 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:

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

void vectorMultiply(float[] a, float[] b, float[] c) {
  int i = 0;
  // It is assumed array arguments are of the same size
  for (; i < SPECIES.loopBound(a.length); i += SPECIES.length()) {
        FloatVector va = FloatVector.fromArray(SPECIES, a, i);
        FloatVector vb = FloatVector.fromArray(SPECIES, b, i);
        FloatVector vc = va.mul(vb)
        vc.intoArray(c, i);
  }

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

The scalar computation after the vector computation is required to process a *tail* of `TLENGTH` array elements, where `TLENGTH < SPECIES.length()` for the vector species. The examples above use the preferred species (`FloatVector.SPECIES_PREFERRED`), ensuring code dynamically adapts to optimal shape for the platform on which it runs.

The helper method `loopBound()` is used in the above code to find the end of the vector loop. A primitive masking expression such as `(a.length & ~(SPECIES.length() - 1))` might also be used here, since `SPECIES.length()` is known to be 8, which is a power of two. But this is not always a correct assumption. For example, if the `FloatVector.SPECIES_PREFERRED` turns out to have the platform-dependent shape `S_Max_BIT`, and that shape has some odd hypothetical size such as 384 (which is a valid vector size according to some architectures), then the hand-tweaked primitive masking expression may produce surprising results.

Performance notes

This package depends on the runtime's ability to dynamically compile vector operations into optimal vector hardware instructions. There is a default scalar implementation for each operation which is used if the operation cannot be compiled to vector instructions.

There are certain things users need to pay attention to for generating optimal vector machine code:

- The shape of vectors used should be supported by the underlying platform. For example, code written using `IntVector` of `VectorShape` `S_512_BIT` will not be compiled to vector instructions on a platform which supports only 256 bit vectors. Instead, the default scalar implementation will be used. For this reason, it is recommended to use the preferred species as shown above to write generically sized vector computations.
- Most classes defined in this package should be treated as value-based classes. This classification applies to `Vector` and its subtypes, `VectorMask`, `VectorShuffle`, and `VectorSpecies`. With these types, identity-sensitive operations such as == may yield unpredictable results, or reduced performance. Oddly enough, `v.equals(w)` is likely to be faster than `v==w`, since `equals` is *not* an identity sensitive method. Also, these objects can be stored in locals and parameters and as `static final` constants, but storing them in other Java fields or in array elements, while semantically valid, will may incur performance risks.

For every class in this package, unless specified otherwise, any method arguments of reference type must not be null, and any null argument will elicit a `NullPointerException`. This fact is not individually documented for methods of this API.

Interface Summary

Interface	Description
VectorOperators.Associative	Type for all reassociating lane-wise binary operators, usable in expressions like `e = v0.reduceLanes(ADD)`.
VectorOperators.Binary	Type for all lane-wise binary (two-argument) operators, usable in expressions like `w = v0.lanewise(ADD, v1)`.
VectorOperators.Comparison	Type for all binary lane-wise boolean comparisons on lane values, usable in expressions like `m = v0.compare(LT, v1)`.
VectorOperators.Conversion<E,F>	Type for all lane-wise conversions on lane values, usable in expressions like `w1 = v0.convert(I2D, 1)`.
VectorOperators.Operator	Root type for all operator tokens, providing queries for common properties such as arity, argument and return types, symbolic name, and operator name.
VectorOperators.Ternary	Type for all lane-wise ternary (three-argument) operators, usable in expressions like `w = v0.lanewise(FMA, v1, v2)`.
VectorOperators.Test	Type for all unary lane-wise boolean tests on lane values, usable in expressions like `m = v0.test(IS_FINITE)`.
VectorOperators.Unary	Type for all lane-wise unary (one-argument) operators, usable in expressions like `w = v0.lanewise(NEG)`.

VectorSpecies<E>	Interface for managing all vectors of the same combination of element type (ETYPE) and shape.

Class Summary

Class	Description
ByteVector	A specialized `Vector` representing an ordered immutable sequence of `byte` values.
DoubleVector	A specialized `Vector` representing an ordered immutable sequence of `double` values.
FloatVector	A specialized `Vector` representing an ordered immutable sequence of `float` values.
IntVector	A specialized `Vector` representing an ordered immutable sequence of `int` values.
LongVector	A specialized `Vector` representing an ordered immutable sequence of `long` values.
ShortVector	A specialized `Vector` representing an ordered immutable sequence of `short` values.
Vector<E>	A sequence of a fixed number of *lanes*, all of some fixed *element type* such as `byte`, `long`, or `float`.
VectorMask<E>	A `VectorMask` represents an ordered immutable sequence of `boolean` values.
VectorOperators	This class consists solely of static constants that describe lane-wise vector operations, plus nested interfaces which classify them.
VectorShuffle<E>	A `VectorShuffle` represents an ordered immutable sequence of `int` values called *source indexes*, where each source index numerically selects a source lane from a compatible `Vector`.

Enum Class Summary

Enum Class	Description
VectorShape	A `VectorShape` selects a particular implementation of `Vector`s.

Report a bug or suggest an enhancement
For further API reference and developer documentation see the Java SE Documentation, which contains more detailed, developer-targeted descriptions with conceptual overviews, definitions of terms, workarounds, and working code examples.
Java is a trademark or registered trademark of Oracle and/or its affiliates in the US and other countries.
Copyright © 1993, 2021, Oracle and/or its affiliates, 500 Oracle Parkway, Redwood Shores, CA 94065 USA.
All rights reserved. Use is subject to license terms and the documentation redistribution policy. Modify Cookie Preferences. Modify Ad Choices.