Vectors and Numerics on the JVM

Part I: Performance Model

Vladimir Ivanov
HotSpot JVM Compiler team
Java Platform Group
Oracle Corp.

Java
Your
Next
(Cloud)

JVMLS 2019

Safe Harbor Statement

Vector API

Perspective

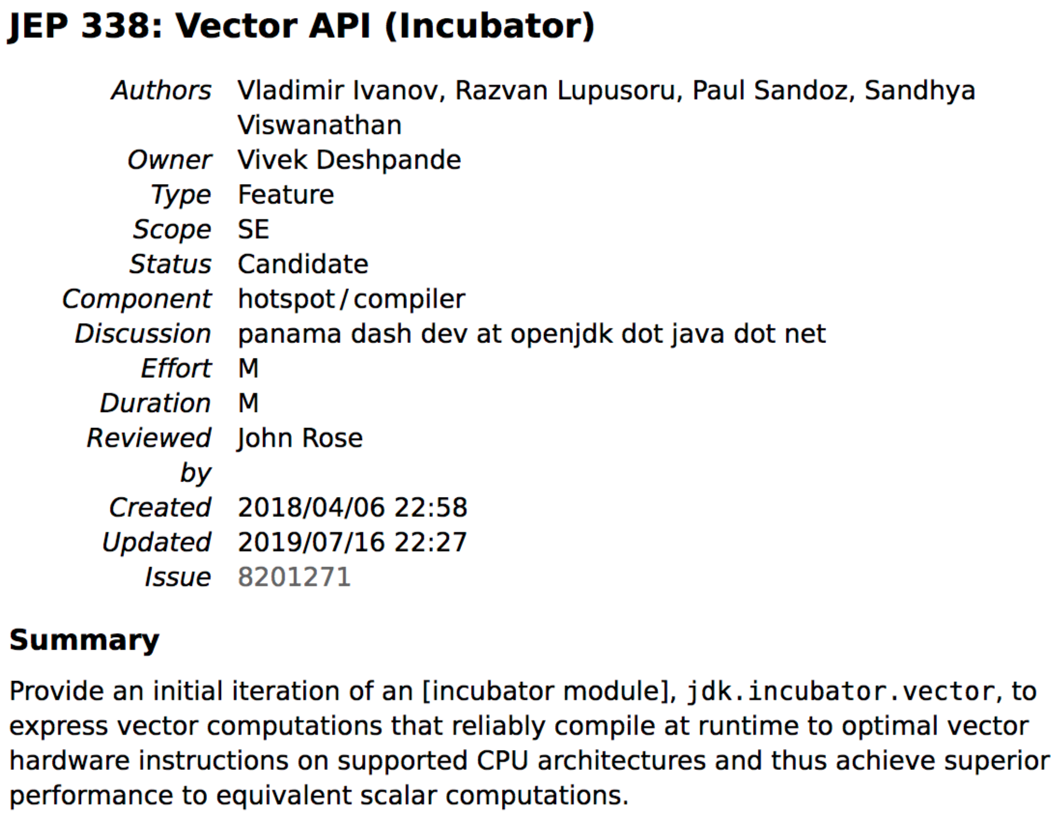
- JVMLS
  - 2016: “Vector API for Java”
  - 2017: “Vectors and Values”
  - 2018: “Java Vector API”
  - **2019: “Vectors and Numerics”**

1. Machine Code Snippets + Super-longs (2016-2017)
2. MVT-based Vectors (2017)
3. **Intrinsic-backed Typed Vectors (2017-now)**

Current Status (August, 2019)

Vector API in Panama

- JEP is still in Candidate state, but…
- First version of API is in CSR!
  - https://bugs.openjdk.java.net/browse/JDK-8223348
  - To be delivered in an upcoming OpenJDK release
  - Will be an incubator project, pending integration with Valhalla
  - Ongoing basic experimentation, including machine learning kernels
  - Who uses it? What's built on top of it? … is TBD. Ideas solicited.
- Lots of work on productizing the implementation went in



JEP 338: Vector API (Incubator)

Authors	Vladimir Ivanov, Razvan Lupusoru, Paul Sandoz, Sandhya Viswanathan
Owner	Vivek Deshpande
Type	Feature
Scope	SE
Status	Candidate
Component	hotspot / compiler
Discussion	panama dash dev at openjdk dot java dot net
Effort	M
Duration	M
Reviewed by	John Rose
Created	2018/04/06 22:58
Updated	2019/07/16 22:27
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.

Case Study: Vectors as Numerics

Challenges

- Performance is the primary goal
  - close to hardware capabilities
- But the only practical representation is boxed
  - no suitable carrier types available
  - unfeasible to add new basic types
- The only option is to rely on JVM to optimize abstractions away
  - don’t make JVM job harder
    - choose proper abstractions
    - JVM-aware implementation

Vector API

Design Goals

1. **Expressive** and **portable** API
   - “principle of least astonishment”
   - uniform coverage operations and data types
   - type-safe
2. **Performant**
   - predictable performance
   - high quality of generated code
   - competitive with existing facilities for auto-vectorization
3. **Graceful** performance **degradation**
   - fallback for "holes" in native architectures

Vector API Design

Roads not taken

Mutable containers == “registers”

- shared boxes, updated in place
- hopefully less boxing to care about

Vector API Design

Roads not taken

~~Mutable containers == “registers”~~

- JIT has to reason about their state
- hard to avoid memory operations for updates

Immutable vectors == vector values

- more boxes to care about
- easier for JIT to reason

Vector API Design

Roads not taken

~~Mutable containers == “registers”~~

Fixed-length vectors

- user codes against vector

Immutable vectors == vector values

Vector API Design

Roads not taken

~~Mutable containers == “registers”~~

~~Fixed-length vectors~~

- no way to adapt to hardware

Immutable vectors == vector values

Length-agnostic vector views

- particular vector shapes are chosen at runtime

Vector API Design

Roads not taken

~~Mutable containers == “registers”~~

~~Fixed-length vectors~~

”Shape-less" vectors

- raw bits
- mimics hardware registers

Immutable vectors == vector values

Length-agnostic vector views

Vector API Design

Roads not taken

~~Mutable containers == “registers”~~

~~Fixed-length vectors~~

~~”Shape-less" vectors~~

Immutable vectors == vector values

Length-agnostic vector views

Strongly typed vectors

- both in size/width and element type
  - enforced by runtime checks
- no implicit conversions performed

Vector API Design

Roads not taken

~~Mutable containers == “registers”~~

~~Fixed-length vectors~~

~~”Shape-less" vectors~~

Carrier type as element type

Immutable vectors == vector values

Length-agnostic vector views

Strongly typed vectors

Vector API Design

Roads not taken

~~Mutable containers == “registers”~~

~~Fixed-length vectors~~

~~”Shape-less" vectors~~

~~Carrier type as element type~~

Immutable vectors == vector values

Length-agnostic vector views

Strongly typed vectors

Element type != carrier type

- carries semantic info, not just “raw bits”
- enables vectors of exotic types
  - unsigned types, exact/saturated operations, minifloats

Vector API Design

Roads not taken

~~Mutable containers == “registers”~~

~~Fixed-length vectors~~

~~”Shape-less" vectors~~

~~Carrier type as element type~~

immintrin.h ported to Java

- operation == hardware instruction

Immutable vectors == vector values

Length-agnostic vector views

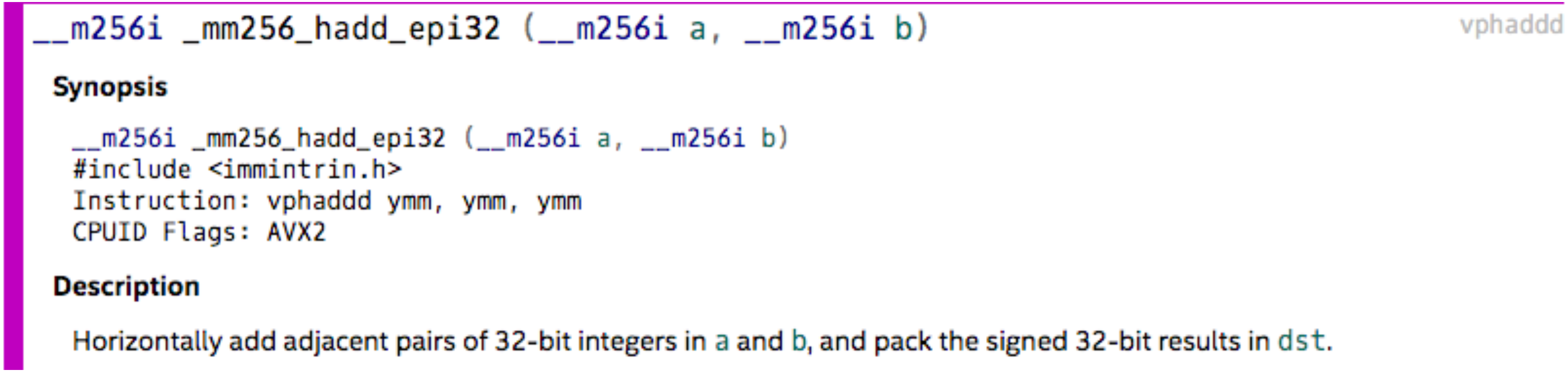
Strongly typed vectors

Element type != carrier type

Vector API Design

Roads not taken

- immintrin.h ported to Java
  - operation == single instruction



`__m256i _mm256_hadd_epi32 (__m256i a, __m256i b)` vphaddd

Synopsis

```
__m256i _mm256_hadd_epi32 (__m256i a, __m256i b)
#include <immintrin.h>
Instruction: vphaddd ymm, ymm, ymm
CPUID Flags: AVX2
```

Description

Horizontally add adjacent pairs of 32-bit integers in a and b, and pack the signed 32-bit results in dst.

Vector API Design

Roads not taken

- immintrin.h ported to Java
  - operation == single instruction

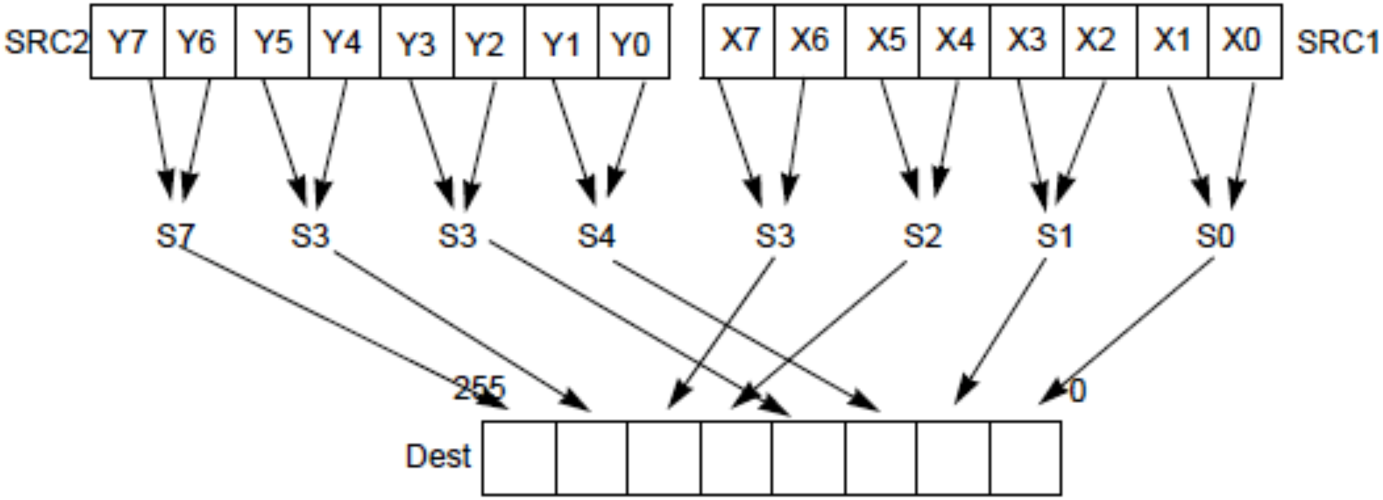


Figure 4-10. 256-bit VPHADDD Instruction Operation

Vector API Design

Roads not taken

~~Mutable containers == “registers”~~	Immutable vectors == vector values
~~Fixed-length vectors~~	Length-agnostic vector views
~~”Shape-less" vectors~~	Strongly typed vectors
~~Carrier type as element type~~	Element type != carrier type
~~immintrin.h ported to Java~~	Portable across wide range of HW

Vector API

```
interface Vector<E> {
    Vector<E> add(Vector<E> v2);
}

interface IntVector extends Vector<Integer> {
    IntVector add(Vector<Integer> v2);
}
```

```
IntVector x = ..., y = ...; // vectors of 8 ints
IntVector z = x.add(y);     // element-wise addition
```

↓

```
vpaddd %ymm1,%ymm0,%ymm0
```

Implementation

Vector API Design

Roads not taken

~~Mutable containers == “registers”~~

~~Fixed-length vectors~~

~~”Shape-less" vectors~~

~~Carrier type as element type~~

~~immintrin.h ported to Java~~

Immutable vectors

Length-agnostic vector views

Strongly typed vectors

Element type != carrier type

Portable across wide range of HW

Implementation Challenges

1. **How to represent vector operations on JVM level?**
   - typed vectors + parameterized intrinsics

2. **Optimize away vector boxes**
   - required for mapping Vector instances to vector registers in generated code
   - Int256Vector => ymm register on x86/AVX

CUT HERE

3. Vectorize higher-order operations
   - higher-order operations are **not** part of the API for now

Key implementation aspects

JVM support

1. Strongly-Typed Vectors
   - class per vector shape
2. Parameterized JVM intrinsics
   - small number of entry points expose large number of behaviors
3. Custom vector box elimination in C2
   - powered by implicit aggressive reboxing
   - stop-the-gap solution until inline classes arrive

Vector Box Elimination

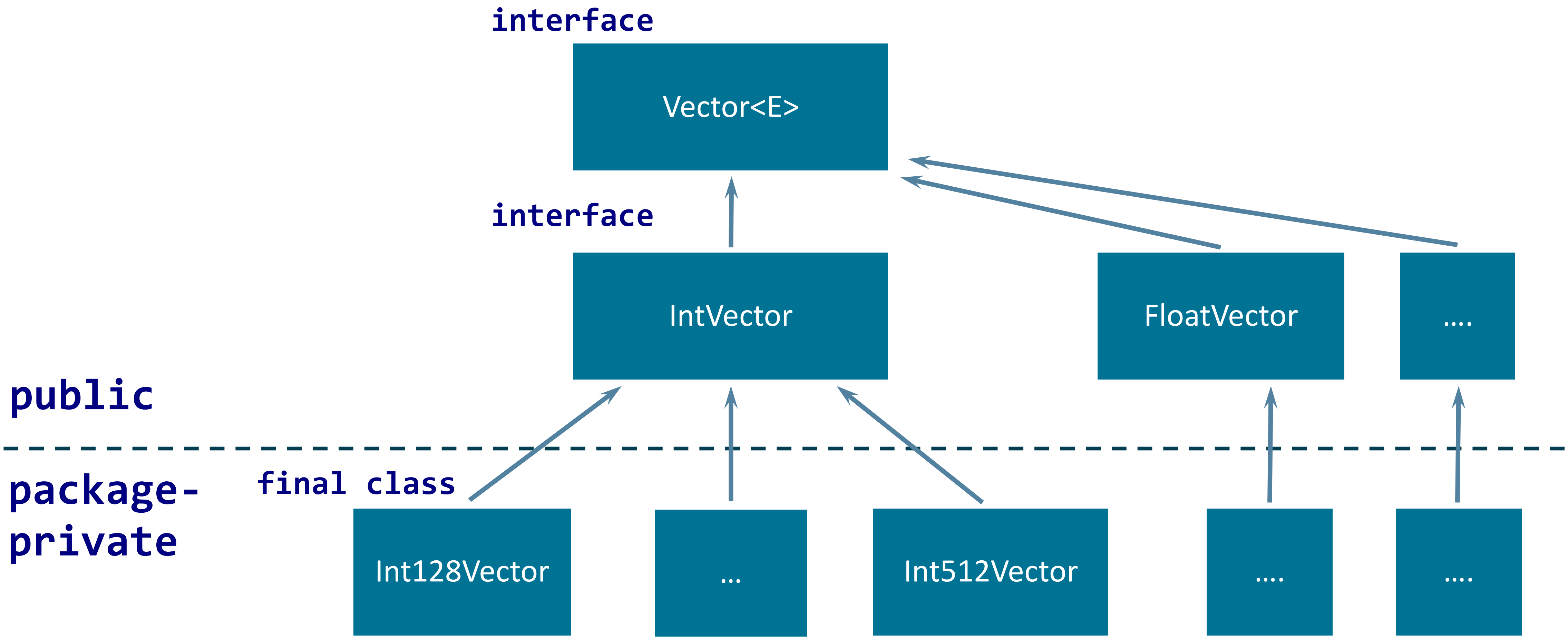
- Crucial for decent performance
- Escape Analysis in C2
  - doesn’t cover all the cases (e.g., non-trivial control flow)
  - conservative, hence brittle
    - depends on inlining decisions
    - easy for a user to break it
- Inline classes should solve the issue
  - Easier to optimize on JVM side
- Stop-the-gap solution: custom vector box elimination analysis
  - Heavily relies on aggressive reboxing

Strongly-Typed Vectors

- “Well-known“ to the JVM
  - special treatment in the JVM
  - C2 knows how to map the values to appropriate vector registers
  - custom vector box elimination pass in C2
    - implicit reboxing (very aggressively!)

size	**8**	**16**	**32**	**64**	**128**	**256**	**512**	**…**	**MAX**
x86	**EAX**			**RAX**	**XMM0**	**YMM0**	**ZMM0**	**-**	**[XYZ]MM0**
JVM	B	S	I	J	**Int128Vector** **Long128Vector** **Float128Vector** …	**Int256Vector** **Long256Vector** **Float256Vector** …	**Int512Vector** **Long512Vector** **Float512Vector** …	…	**IntMaxVector** **LongMaxVector** **FloatMaxVector** …

`package jdk.incubator.vector;`



interface

Vector<E>

interface

IntVector

FloatVector

....

public

package-private

final class

Int128Vector

...

Int512Vector

....

....

```
package jdk.incubator.vector;

/*non-public*/ class VectorIntrinsics {

    @HotSpotIntrinsicCandidate
    static
    <V extends Vector<?>>
    V binaryOp(int      operatorId,
               Class<V> vectorClass,
               Class<?> elementType,
               int      vlen,
               V v1,
               V v2,
               BiFunction<V,V,V> defaultImpl) {…}
```

```
package jdk.incubator.vector;

/*non-public*/ class VectorIntrinsics {

    @HotSpotIntrinsicCandidate
    static
    <V extends Vector<?>>
    V binaryOp(int        operatorId,   // vector operation
               Class<V> vectorClass,  // vector class
               Class<?> elementType,  // vector element
               int        vlen,         // vector length

               V v1, V v2,
               BiFunction<V,V,V> defaultImpl) {…}
```

```
package jdk.incubator.vector;

/*non-public*/ class VectorIntrinsics {

    @HotSpotIntrinsicCandidate
    static
    <V extends Vector<?>>
    V binaryOp(int      operatorId,
               Class<V> vectorClass,
               Class<?> elementType,
               int      vlen,

               V v1, V v2, // operation arguments

               BiFunction<V,V,V> defaultImpl) {…}
```

```
package jdk.incubator.vector;

/*non-public*/ class VectorIntrinsics {

    @HotSpotIntrinsicCandidate
    static
    <V extends Vector<?>>
    V binaryOp(int      operatorId,
               Class<V> vectorClass,
               Class<?> elementType,
               int      vlen,
               V v1, V v2,

               BiFunction<V,V,V> defaultImpl) {…}

               // implementation in Java
```

```
Int256Vector v1 = …
Int256Vector v2 = …

BiFunction<…> int256addImpl = (v1,v2) ->
     v1.bOp(v2, (i, a, b) -> (int)(a + b));

Int256Vector vr =
      binaryOp(OP_ADD,
               Int256Vector.class,
               int.class,
               8,
               v1, v2,
               int256addImpl);
```

```
Int256Vector v1 = …
Int256Vector v2 = …

BiFunction<…> int256addImpl = (v1,v2) ->
     v1.bOp(v2, (i, a, b) -> (int)(a + b));

Int256Vector vr =
      binaryOp(OP_ADD,
               Int256Vector.class,
               int.class,
               8,
               v1, v2,
               int256addImpl);
```

`int256addImpl.apply(v1, v2)`

`vpaddd %v1,%v2,%vr`

Performance

Existing Benchmarks

- Mandelbrot
- SepiaFilter
- Large set of microbenchmarks
  - http://hg.openjdk.java.net/panama/dev/file/a059f2c353cf/test/jdk/jdk/incubator/vector/benchmark/src/main/java/benchmark/jdk/incubator/vector/
- Externally developed benchmark suites
  - https://github.com/richardstartin/vectorbenchmarks/ by Richard Startin
    - DotProduct, MatrixMultiplication, …
  - https://github.com/blacklion/panama-benchmarks/tree/master/vector

    by Lev Serebryakov

Performance Pitfalls

Main Causes

1. Box elimination failures
   - boxing in tight vector code has severe impact

2. Intrinsification failures
   - causes box elimination failures
     - implementation detail
     - Java implementations work on boxed representation
   - mixes intrinsified and non-intrinsified operations in the IR
     - complicates box elimination analysis

Performance Pitfalls

Box elimination failures

1. Identity-sensitive operations
    - aggressive reboxing, but box elimination is still conservative
        - may still break identity invariants
            - controlled by -XX:+/-AggressiveReboxing
    - treated as user mistake for now

Performance Pitfalls

Box elimination failures

1. Identity-sensitive operations
   - aggressive reboxing, but box elimination is still conservative
     - may still break identity invariants
       - controlled by -XX:+/-AggressiveReboxing
   - treated as user mistake for now
2. Inlining failures
   - box elimination analysis is inherently local
   - may be caused by profile pollution
     - multiple vector shapes seen in shape-agnostic code
   - triggers boxing/unboxing around the call

Performance Pitfalls

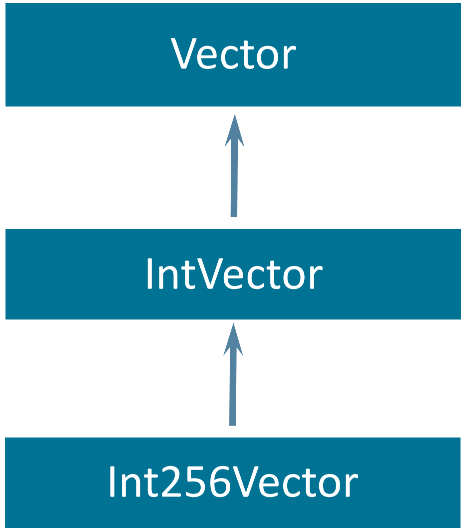
Intrinsification failure

- Missing hardware support
  - treated as a bug when used with preferred vector species
    - VectorSpecies.ofPreferred(Class<E> elementType)
  - can be encountered when working with concrete vector species
    - XxxVector.SPECIES_PREFERRED vs XxxVector.SPECIES_512

Performance Pitfalls

Intrinsification failure

- Not enough information about the operation
  - all operation defining arguments should be seen by JIT as constants
  - otherwise, non-intrinsified implementation is used
  - trade-offs on implementation side
    - code customization vs code sharing
    - “call + intrinsified version” vs “default implementation”



```
binaryOp(operatorId,
         vectorClass,
         elementType,
         vlen,
         v1, v2, impl)
```

→ `impl.apply(v1, v2)`

Performance Pitfalls

Recommendations

For now:

1. Use preferred vector species when working with shape-agnostic vector code
   - XxxVector.SPECIES_PREFERRED / VectorSpecies.ofPreferred(Class elementType)

2. Keep vector code in a single method to avoid inlining issues
   - inlining heuristics are hard to reason about
   - calls in cold code may pose some challenges to vector box elimination
     - aggressive reboxing sometimes improves the situtation

Better JVM support

~~Value~~ Inline Classes

- Reliable solution to boxing issues
  - completely obsoletes custom box elimination logic
  - concrete typed vector classes (XxxNnnVector) migrate to inline classes
    - hidden from users, exposed through Vector interface or primitive specializations (XxxVector)
  - Identity-sensitive operations don’t block optimizations
    - Either forbidden or have consistent behaviour irrespective of buffer identity
  - Flattening enables better design
    - Super-longs as raw carrier types (Int128/Int256/Int512) + XxxNnnVector as typed wrappers
- What about inlining issues and intrinsification?

~~Value~~ Inline Classes

Inlining

- Doesn’t completely eliminate inlining issues
  - … and profile pollution is still there
  - buffering around calls is needed without additional JVM support
    - depending on JVM implementation, buffering may be cheaper than boxing

- Possible answer - vector calling conventions
  - Inline classes enable custom calling conventions in the JVM
    - Pass arguments/receive results in scalarized form
    - … but that works only for inline classes in the signature

~~Value~~ Inline Classes

Vector calling convention

- Map concrete vector classes to vector registers?
  - but XxxNnnVector are implementation detail and not part of the API!
- Cover XxxVector instead?
  - but it’s not an inline class, but an interface!
  - … and XxxVector may represent “super-vectors”
- Begs for a different representation
  - A single inline class which encapsulates whole hardware vector register + vector shape (size + element type) information
    - MaxVector – like XxxMaxVector, but with element type omitted
  - Custom entry point based on profile info

~~Value~~ Inline Classes

Vector calling convention

- Begs for a different representation
  - A single inline class which encapsulates whole hardware vector register + vector shape (size + element type) information
    - MaxVector – like XxxMaxVector, but with element type omitted
  - Custom entry points based on profile?
- Requires additional work for intrinsification
  - Type info is not statically known anymore
  - Less of an issue for newer hardware
    - predication in AVX512 and SVE (ARM) enables variable size instruction encodings

Summary

Summary

- Vector<E>
  - + new carrier types
  - + intrinsics
    - – AVX* on x86, NEON/SVE on ARM
  - + inline classes

- Complex
  - +/- new carrier types (64-/128-bit)
  - +/- intrinsics
  - + inline classes

- Half precision (binary16), bfloat16
  - \- new carrier type (16-bit)
  - +/- intrinsics
    - – F16C, AVX512_BF16 on x86
  - + inline classes

Summary

- Vector<E>
  - **+** new carrier types
  - **+** intrinsics
    - AVX* on x86, NEON/SVE on ARM
  - **+** inline classes
  - **+** shape-agnostic

- Vector<Complex>
- Vector<binary16>
- Vector<bfloat16>

…

- Complex
  - **+/-** new carrier types (64-/128-bit)
  - **+/-** intrinsics
  - **+** inline classes
  - **-** shape-agnostic

- Half precision (binary16), bfloat16
  - **-** new carrier type (16-bit)
  - **+/-** intrinsics
    - F16C, AVX512_BF16 on x86
  - **+** inline classes
  - **-** shape-agnostic
- Minifloats, binary128/256, …

Java™

ORACLE®