Vectors and other Primitives

John Rose, JVM Architect

JVM Language Summit, Santa Clara, July 2019

http://cr.openjdk.java.net/~jrose/pres/201907-Vectors.pdf

ORACLE

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract.
It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.

What's cool about Panama vectors?

http://hg.openjdk.java.net/panama/dev/branches >> vectorIntrinsics

- nice demos (we hope)
- vectors = right-sized data processing (multi-word SIMD)
- good old Java is doing some new tricks on today's vector CPUs
  - (we just heard about some really creative JIT & JDK work)
  - assembly level performance, with all the comforts of `$JAVA_HOME`
- Valhalla mojo = object APIs without pointers/headers/heaps
- insight and experience navigating towards Java futures:
  - templated data and algorithms (like C++ but native to Java)
  - shaping new primitives: complex, unsigned-int, int128, etc.
  - higher-level vectors, with higher-level operations (`FORALL` in Java)

ORACLE

DEMO

http://cr.openjdk.java.net/~jrose/vectors/DEMOJVMLS2019.jsh

```
// DEMOJVMLS2019.jsh : simple interactive demo of 2019 Vector API
// run jshell from a recent build of Panama vectorIntrinsics branch:
// $ cd panama; hg pull -u; hg co e45a5c05a746; make jdk
// $ build/macosx-x86_64-server-release/jdk/bin/jshell DEMOJVMLS2019.jsh
/env --add-modules jdk.incubator.vector
import jdk.incubator.vector.*;
import jdk.incubator.vector.Vector;
import static jdk.incubator.vector.VectorOperators.*;
// load successive squares into a, alternating signs into b, small k
float[] a = new float[24], b = new float[a.length], r = new float[a.length];
for (int i = 0; i < a.length; i++) { a[i] = i*i; b[i] = (i&1)==0?1:-1; }
var k = .002f;
var VSP = FloatVector.SPECIES_PREFERRED;
// compute forall<i>  r[i] = fma(sqrt(a[i]), b[i], k)
for (int i = 0; i < a.length; i += VSP.length()) {
 var av = FloatVector.fromArray(VSP, a, i);
 var bv = FloatVector.fromArray(VSP, b, i);
 var rv = av.lanewise(SQRT).lanewise(FMA, bv, k);
 rv.intoArray(r, i); }
var rv = VSP.fromArray(r, 0); rv; rv.species() // stuff to print
rv.lanewise(COS); rv.test(IS_NEGATIVE); rv.lanewise(COS).test(IS_NEGATIVE)
// once more with feeling!
/reload
```


ORACLE

HOW ABOUT THOSE VECTORS…

ORACLE

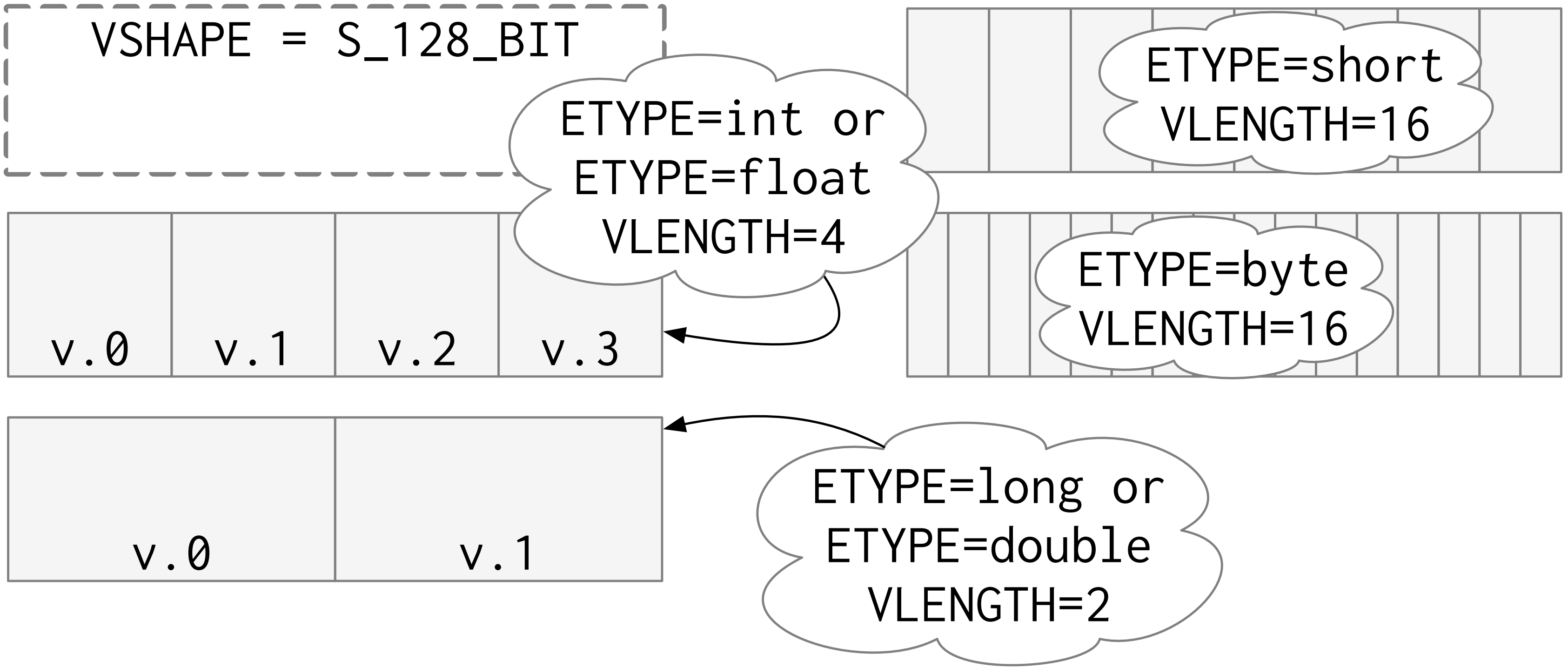
The basics: what’s in a vector?

doc/root/jdk.incubator.vector/jdk/incubator/vector/Vector.html

- A vector is a small, dense tuple of scalars of fixed length `VLENGTH`
- The scalars are all of the same primitive `ETYPE` (“element type”)
  - Each `ETYPE` has a size in bits (`ESIZE`). Also float, integral, …
- Each location in the vector is called a lane, numbered from zero
  - Thus, `v = [ v.0 | v.1 | … | v.7 ]` (`VLENGTH=8`)
- Vectors are close to the hardware, classified via total bit-size
  - `sizeof(v) = VLENGTH * ESIZE`
- Vector shape (`VSHAPE`) determines bit-size and register class.
  - `VSHAPE` and `ESIZE` together imply `VLENGTH`, maybe other things…
- Finally, `VSHAPE` plus `ETYPE` implies vector **species** (`VSPECIES`).

The basics: what's in a vector? *(dense payloads)*

doc/root/jdk.incubator.vector/jdk/incubator/vector/Vector.html



The basics: lane-wise operations are distributed

doc/root/…/Vector.html#lane-wise

- Unary distribution: `v ▷ op ≔ [ v.0 ▷ op | v.1 ▷ op | … ]`
- Scalar distribution: `v ▷ op(e) ≔ [ v.0 ▷ op(e) | v.1 ▷ op(e) | … ]`
  `broadcast(e) ≔ [ e | e | … ]` (for some particular `VSPECIES`)
- N-ary distribution: `v ▷ op(v*) ≔ [ v.0 ▷ op(v*.0) | v.1 ▷ op(v*.1) |…]`

```
//for (…i…) r[i] = fma(sqrt(a[i]), b[i], k);
for (int i…; i += VLENGTH) {
  for (int L = 0; L < VLENGTH; L++) {
    r[i+L] = fma(sqrt(a[i+L]), b[i+L], k);
    //rv = av ▷ SQRT ▷ FMA(bv, k);
    //rv = av.lanewise(SQRT).lanewise(FMA, bv, k);
} }
```

The basics: memory access is block-wise

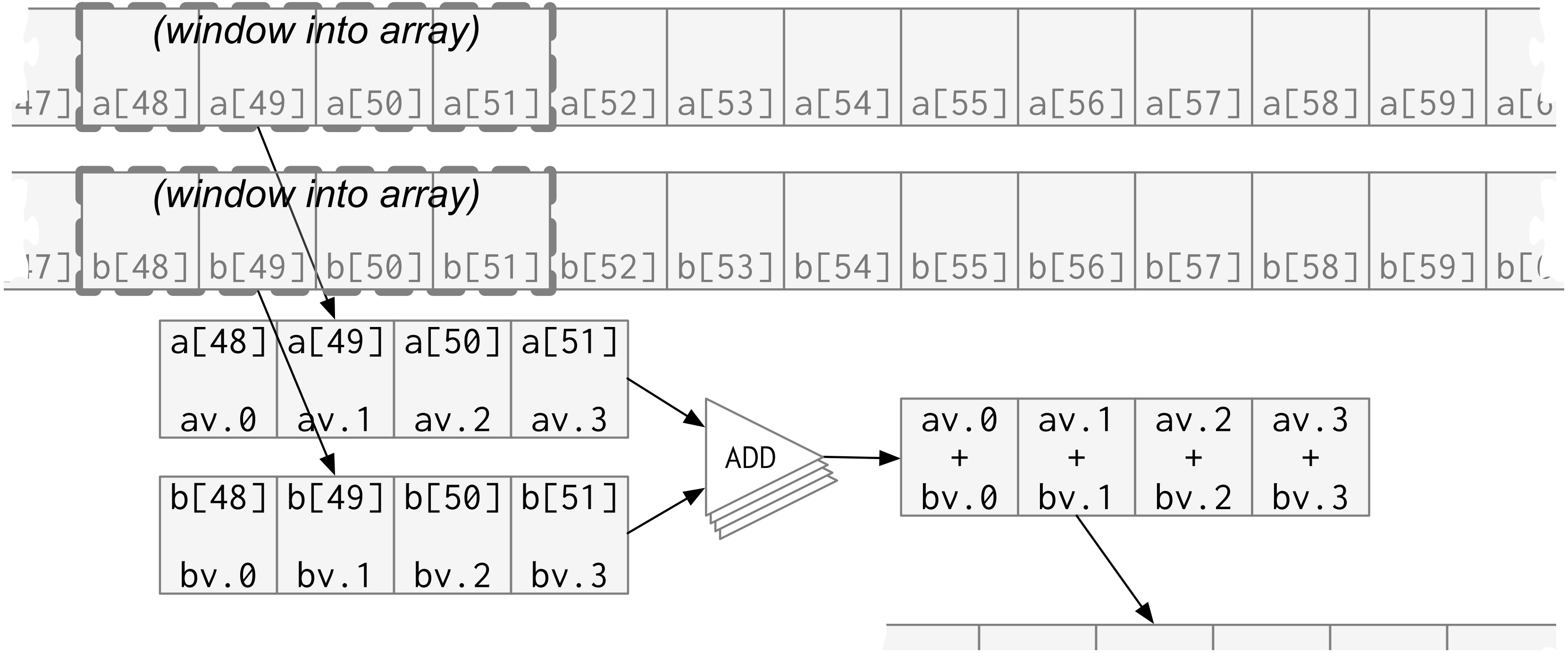
doc/root/.../Vector.html#lane-wise

- Load: `fromArray(a, i) ≔ [ a[i+0] | a[i+1] | … | a[i+VLENGTH-1] ]`
- Store to new: `v.toArray() ≔ new ETYPE[]{ v.0, v.1, … }`
- Store to old: `v.intoArray(a, i) ≔ { a[i+0] = v.0; a[i+1] = v.1; … }`

```
var VSP = FloatVector.SPECIES_PREFERRED;
for (int i…; i += VSP.length()) {
  var av = FloatVector.fromArray(VSP, a, i);
  var bv = FloatVector.fromArray(VSP, b, i);
  var rv = av.lanewise(SQRT).lanewise(FMA, bv, k);
  rv.intoArray(r, i);
}
```

Lane-wise is coherent with block-wise

SIMD programming: Single Instruction (operation) Multiple Data (lanes)



ORACLE

The basics: why use a vector?

- Lane-wise operations run in parallel (speedup factor = `VLENGTH`).
  - arithmetic units replicated across lanes (this silicon is cheap)
- Loads/stores are the same scale (cache line) as memory fabric ops.
  - whole cache line used ⇒ memory traffic contains only useful data

- Lane operations "fly in formation" through CPU; low traffic control costs
  - Equivalent scalar loop must watch for cross-lane dependencies
- Resulting user model: Unroll all your loops by `VLENGTH` and repack.
  - Sometimes JITs can do this for you, but it's hard to control.
  - If your CPU is multiple-issue, the JIT may unroll more after that.

ORACLE

Hand unrolling, really? Give us a break.

- There must be a better way.   What are workarounds for vectorization?
- Your original algorithm talks about big data (arrays) and scalars.
  - Vector code, in addition, deals with an intermediate scale (`VLENGTH`).
  - More complexity from the new entities and new edge conditions.
  - Greater performance ⇐ greater control ⇐ greater attention & skill.
- Scalar notations (C/Java for-loops) auto-vectorize ***if you are lucky***.
- Direct array processing notations work at the largest grain size.
  - Fortran `FORALL` statement, APL-like languages (Julia, MATLAB).
- Assembly code is fast, but very hard to write, with a short shelf life.
- Explicit vectors are sometimes the worst option—except all the others.

JAVA'S GOT A BRAND NEW BAG

the middle ground: High level explicit vector code

- What if you could get control close to assembly code, from C or Java?
  - After `VLENGTH` unrolling, the compiler or JIT finishes optimizing.
- C's `<immintrin.h>` : 1 intrinsic function call ≈ 1 instruction.
  - Downsides: Low-level notation. Not portable. C-level tooling.
- Enter Java's new trick: the Vector API
  - Explicit like C with intrinsics; 1 method call ≈ 1 instruction.
  - Packaged Java-style with interfaces, methods, generic types.
  - Works on the Java toolchain (IDEs, jshell, etc.)
- Extra benefit: JIT compilation can dynamically choose best `VLENGTH`
  - "Write once, unroll (differently) everywhere"

so, a Java API for explicit vector programming

- Types `FloatVector`, `IntVector` (… `Double/Long/Byte/ShortVector`)
  - Generic top-type `Vector<E>` (so `FloatVector <: Vector<Float>`)
  - `VectorSpecies<E>` to reflect over vector types; `VectorShape` enum.
- Methods to load/store to/from arrays & full NIO buffer integration.
- Lane-wise operator methods (arity 1/2/3) with many operations.
  - Also lane-wise test methods (arity 1/2) with more operations.
  - Also lane-wise conversion methods with yet more operations.
- `VectorMask<E>` to capture test results and steer subsequent ops.
  - Vector and mask operations to help control loop "edge cases".
- Local cross-lane movement represented with `VectorShuffle<E>`

Panama Vector API requirements

- Must look like Java: Objects, interfaces, generics, safety, tooling.
- Must be able to directly express a range of typical vector loop kernels
  - Dot product, hash code, string match, crypto, sort, …
- A vectorized for-loop must be maintainable (perhaps with tradeoffs)
  - Maintainable because appropriately abstract, legible, portable.
  - Vector shape must be abstractable from loop shape.
  - (Payoff: Legible, portable code has a longer shelf life!)
- Explicitly non-portable code should be possible, but not encouraged.
  - User makes final choices between performance and maintainability.
- Operator notations should be natural. ***THIS BIT ISN'T TRUE YET.***

Panama Vector API methods

```
L = .length(), ET = v.elementType(), VSP = v.species(), v.check(ET)
w = v.lanewise(OP [,v′/e [,v″/e]] [,m])  /*Unary|Binary|Ternary OP*/
w = v.add(v′/e [,m]), sub/mul/div/min/max/…  /*“full service” methods*/
w = v.addIndex(step)  /*add scaled lane index*/
m = v.compare(OP, v″/e [,m]),  m = v.test(OP [,m]), m = v.eq/lt(v′/e)
w = v.blend(v′, m)  /* lanewise(m ? v′ : v) */
w = v.convert(OP, part)  w = v.convertShape(OP, species, part)
w = v.reinterpretShape/AsBytes/AsInts/…
sh = v.toShuffle(),  w = v.viewAsIntegralLanes…
w = v.slice(origin [,v′] [,m]),  unslice…
w = v.rearrange(shuffle [,v′/m]),  w = v.selectFrom(v′)
v.intoArray(a, i), v.intoBB(bb,off,bo), a = v.toArray()
v = TVector.fromArray(a, i),  v = [v/TVector/VSP].broadcast(e)
```


ORACLE

Vector operations

doc/root/jdk.incubator.vector/jdk/incubator/vector/VectorOperators.html

Binary OP: v.lanewise(OP,v′/e [,m]), v.reduce(OP [,m])				
ADD/SUB/MUL/…	MIN/MAX	AND/OR/XOR/…	LSHL/ASHR/…	ATAN2/POW/…

Unary OP: v.lanewise(OP [,m]), v.lanewise(OP,m)				
NOT	ABS	NEG	SIN/COS/TAN/…	EXP/LOG/SQRT/…

v.lw(OP,v′,v″)
FMA/…

v.compare(OP,v′/e)
LT/GT/EQ/…

v.test(OP)	
IS_DEFAULT	IS_NAN/…

ORACLE

Vector comparison operations

doc/root/jdk.incubator.vector/jdk/incubator/vector/VectorOperators.html

Conversion OP: w=v.convert(OP,part)				
B2S/B2I/B2F/…	L2B/L2S/L2I/…	REINTERPRET_F2I	REINTERPRET_D2L	REINTERPRET_…
S2B/S2I/S2F/…	F2B/F2S/F2I/…	ZERO_EXTEND_B2I	ZERO_EXTEND_B2I	ZERO_EXTEND_I2L
I2B/I2S/I2F/…	D2B/D2S/D2I/…	INPLACE_B2I/S2I	INPLACE_D2F/D2I…	INPLACE_ZERO_ EXTEND_B2I/I2L/…

Fixed sized chunks vs. size-changing operations

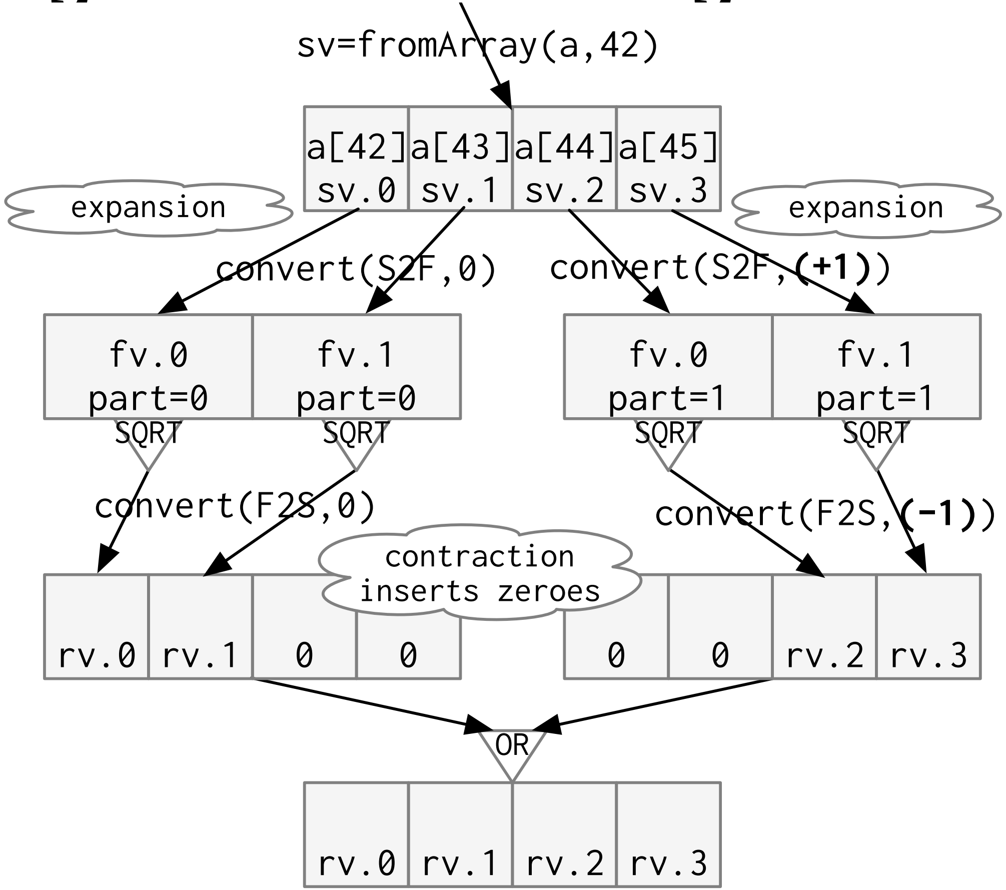
doc/root/…/Vector.html#expansion

- Key idea: Reify potential size changes as a ***part-number*** parameter.
- Example: `(w, w′, w″, w‴) = v.convert(B2I, [part=0,1,2,3])`
- Example: `(w, w′, …) = v.reinterpretShape(VSP, [part=…])`
- Example: 16-bit square root, using temporary expansion:

```
var FSP = FloatVector.SPECIES_PREFERRED, VSP = FSP.withLanes(short.class);
for (int i…; i += VSP.length()) {
  var sv = ShortVector.fromArray(VSP, a, i);
  ShortVector rv = sv.broadcast(0);
  for (int part = 0; part < 2; part++) {
    var fv = sv.convert(S2F, part).lanewise(SQRT).plus(0.5f);
    rv = rv.lanewise(OR, fv.convert(F2S, -part));
  } rv.intoArray(a, i); }
```


ORACLE

Expanding data flows through multi-part vectors



ORACLE

influences from Intel AVX

https://software.intel.com/sites/landingpage/IntrinsicsGuide

- Vector shape control and abstraction from AVX, AVX2, AVX512
- Masks look forward to AVK512 ‘k’ registers.
  - But `VectorMask<E>` hides its implementation (it might be a vector)
- Large number of snowflake ops pushed us toward `lanewise(OP)`.
- Generally, the support for lane-wise C expressions is strong.
  - (Java and C have multiple scalar sizes, many ops & conversions.)
- Reductions (“horizontal add”) are common. (Scans are not.)
- Gather/scatter ops are incomplete until AVX512.
- Cross-lane permutations: General vperm, plus many “funny butterflies”.
  - `VectorShuffle<E>` is a thinly-veiled vector (or array?), like `VectorMask`.

ARM64 SVE

https://developer.arm.com/docs/ddi0584/latest

- Vectors might be long and oddly-sized, and will be detected at runtime.
  - This is a good match for Java's portability goals.
  - We had to remove power-of-two assumptions from the API.
- Like AVX, a good set of C-expression support (ops, conversions)
  - Similar treatment of `VSIZE = VLENGTH * ESIZE`
  - This helped us settle on shape-invariance as a normal user model.
- Nice suite of cross-lane movement (zip/unzip/pack/unpack/transpose)
  - We want to optimize "well known" shuffles into such instructions.
- Data-driven (mask-based) lane compression not covered yet.
  - (Intel doesn't have this operation. But it's `Stream::filter`!)

What works well…

- Vectors are objects, Java is good at modeling them. (No surprise.)
- Simple vector loops compile (often) to simple hot assembly loops.
- A large range of AVX, AVX2, & AVX512 instructions are reachable
  - It seems likely we can do the same with NEON, SVE and others
- We have reasonable-looking portable semantics
  - Byte order, bit order, exceptions and array range safety, masks
  - Conventions for “expansion” and “contraction” (zip/unzip, etc.).
- So, the same source code can run with different vector ISAs
- Source code can also be hand-tuned for particular vector ISAs
  - The data-driven operator scheme leaves room for “snowflake” ops.

And what doesn’t work so well…

- Vectors require very aggressive inlining and unboxing
  - Valhalla will make this systematic. For now it’s ad hoc and fragile.
- Code is tricky and hard to maintain, because of specialization hacks
  - A NIO-style textual preprocessor manages template types
  - A ton of `@ForceInline` gives us an effect like template methods
- We say `Vector<Integer>` when we really mean `Vector<int>`
  - Valhalla plans to address this problem, for the sake of inline types.
- Java stops at 8 primitive types, so no `Vector<complex>`, `Vector<int128>`
  - We expect Vahalla will let us define types like `complex` and `int128`.

Old-school algebra & FORTRAN are inescapable

- Vector expressions ***LOOK NOTHING LIKE*** scalar expressions.
- Algebra expressions like `r = a*x + b` are here to stay.
  - In Java that must be `r = a.mul(x).add(b)`. (As in Vector API.)
  - This is a readability problem. Users have a right to balk at this.
- *Operator overloading?* C++ and Python versions are too wild for Java.
  - Maybe we can cook up some algebras (operator suites w/ contracts).
  - But this needs research, and specialized generics are a prerequisite.
- A better near term solution is *lambda cracking* (a la .NET).
  - No language changes required, just a new form of reflection.
  - Lambdas could be checked at compile-time via javac intrinsics.
  - Smooth upgrade from limited operators (`ADD`, `FMA`) to lambdas.

ORACLE

A gentle introduction to the cracking of lambdas

- Today: `vr = va.lanewise(SUB, vb.lanewise(MUL, 42)).lanewise(MAX, 0);`
  - **If you know how**, you can read it as: `r = max(a - b*42, 0);`
- Next we make a little AST language to extend type `Operator` types:

```
    static final Binary MYOP = MAX.of(SUB.of(A,MUL.of(B,42)), 0);
vr = va.lanewise(vb, MyOP);
```

- Mix in some static javac intrinsics, to perform some static checks: https://bugs.openjdk.java.net/browse/JDK-8205637
- Or, break out the parser: `MYOP = expression("max(a - b*42, 0)”);`
- And for dessert, sugary cracklin’ lambdas:

```
vr = va.lanewise(vb, (a,b)->max(a - b*42, 0);
```

  - It’s AST hacking under the hood. Maybe some can be at compile-time.
  - This is a long string to pull. Let’s eat this dessert for breakfast tomorrow.

Rash speculation about Primitives of the Future

- The challenge with Java primitives is they are “just data”, not methods.
  - Their behaviors are in odd places: JLS for operators, `Math.abs`.
- With Valhalla, methods on wrapper types find a natural home.
  - But operators don’t model well in single-receiver OOLs.
- An approach: Use generic interfaces to capture the rules of algebra.
  - “just the data” (`int`, `Complex`, `Unsigned`, `Vector`) is in type param(s).
  - The behavior parts are in “witness” object(s) that implement ops

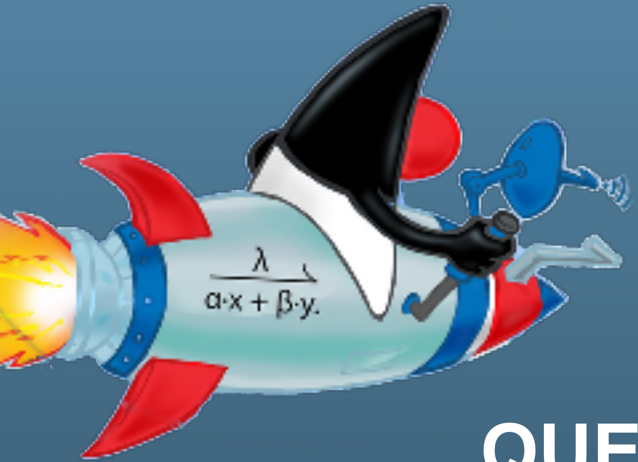
```
interface BitwisePrimitive<T> { T and(T a, T b); … }
```

- Lambdas can be cracked and retargeted from (say) `long` to `Unsigned`, given the presence of a suitable witness `BitwisePrimitive<Unsigned>`
- Conversion rules are witnessed by `ConvertiblePrimitive<T,U>` (etc.)

ORACLE

And we always want more…

- More operators: `REVERSE_BITS`, `ROUND`, `CEILING`, …
  - Macro-operators: AST first; then some sugary cracked lambdas)
  - Snowflakes: `AES_STEP`, `CLMUL`, `SATURATING_ADD`, funny butterflies, …
- More support for near-neighbor communication (shuffles, pack/unpack)
  - More flavors of zip/unzip/pack/unpack/transpose (SOA vs. AOS)
  - Data-driven lane packing (vectorized `Stream::filter`)
  - Segmented scan (reduce with partials and mask-driven reset)
- More loop shapes: Integrated pre/main/post notations.
  - Stream-based vector loops. Maybe array processing?
  - Experiment with BLAS heavy lifting (does it make sense?)
- Integration (via Panama) with vector types in system ABIs.
- More lane types (via Valhalla): complex, fixed-point, hyper-longs, single bits.



QUESTIONS?

ORACLE

The previous is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract.
It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.