ORACLE

Vector API design: Selected topics

(flat values, future shapes, cross-lane motion, snowflakes, atomics)

John Rose, Oracle JPG

http://cr.openjdk.java.net/~jrose/pres/202202-VectorTopics.pdf

Feb 16, 2022

Fitting vectors into bare values — a few ideas

Key observation: Vector lanes are indexed, not selected by name

- A vector is a homogeneous short array
  - A value class is a heterogenous tuple
- Array indexing v[i] requires an in-memory presentation
  - A value class, even if in memory, cannot index its fields
  - Class fields are often, but not always, contiguous
- First compromise: A private multi-field feature in the class loader
  - One field, secretly replicated N times CONTIGUOUSLY
  - (Probably) allocated by JVM after any/all other fields
  - Access (other than to first copy) is via var-handle only
- Additional compromise: Register allocator assigns it a vector register
  - Either automatic, or an extra ad hoc register class annotation

More details: http://cr.openjdk.java.net/~jrose/values/multi-field.html

Example “multi-field” feature

```
@RegisterAllocation(128) value class Shape128 {
    @MultiField(2) long data;
    //long data#1 ⇐ allocated contiguously
}
@RegisterAllocation(256) value class Shape256 {
    @MultiField(4) long data;
    //long data#{1,2,3} ⇐ allocated contiguously
}
@RegisterAllocation(512) value class Shape512 {
    @MultiField(8) long data;
    //long data#{1,2,3,4,5,6,7} ⇐ allocated contiguously
}
```

Example var-handle based access (HALF-BAKED)

```
value class Shape512 {
    @MultiField(8) long data;
    private static long OFF_data = U.objectFieldOffset(F_data);
    private static final VarHandle VH_data =
      new VarHandleMultiLongs(Shape512.class, OFF_data, 8);
    public long data(int i) {
      return (long) VH_data.get(this, i);
    }
    public Shape512 dataUpdate(int i, long x) {
      Shape512 pbuf = U.makePrivateBuffer(this);
      VH_data.set(pbuf, i, x);
      return U.finishPrivateBuffer(pbuf);
    } //OR return (Shape512) VH_data.‘update’(this, i, x);
}
```

After discussion, the ‘update’ operator doesn’t look so good; use arrays

```
value class Shape512 {
    @MultiField(8) long data;
    private static long OFF_data = U.objectFieldOffset(F_data);
    private static final VarHandle VH_data =
      new VarHandleMultiLongs(Shape512.class, OFF_data, 8);
    public long data(int i) {
      return (long) VH_data.get(this, i);
    }
    public static Shape512 make(long[] a) {  //optimizer makes temporary ‘a’ disappear
      Shape512 pbuf = U.makePrivateBuffer(this);
      for (int i = 0; i < 8; i++)
        VH_data.set(pbuf, i, a[i]);
      return U.finishPrivateBuffer(pbuf);
    }
}
```

Straightforward to expand to multi-vector shapes

```
@RegisterAllocation(128) value class Shape128x2 {
    @MultiField(2*2) long data;
}
@RegisterAllocation(256) value class Shape256x3 {
    @MultiField(4*3) long data;
}
@RegisterAllocation(512) value class Shape512x4 {
    @MultiField(8*4) long data;
}
// Nested representation would be more complex to implement
value class Shape256x5 {
    @MultiField(5) Shape256 nestedVector;
    // requires carefully contiguous layout of the nestedVectors
}
```

Idea from the group: Maybe have “one size fits most” for polymorphic?

```
@RegisterAllocation(128) value class Shape128 extends Shape { @MultiField(2) long data; }
@RegisterAllocation(256) value class Shape256 extends Shape { @MultiField(4) long data; }
@RegisterAllocation(512) value class Shape512 extends Shape { @MultiField(8) long data; }

// AND THEN:

@RegisterAllocation(512 /*=MAX*/)
abstract class Shape permits Shape128, Shape256, Shape512 { }

// … models the overlapping zmm/ymm/xmm register file
// with a klass token, maybe useful for polymorphic calling conventions??
```

The shapes of things to come —rectangles, tiles, matrices

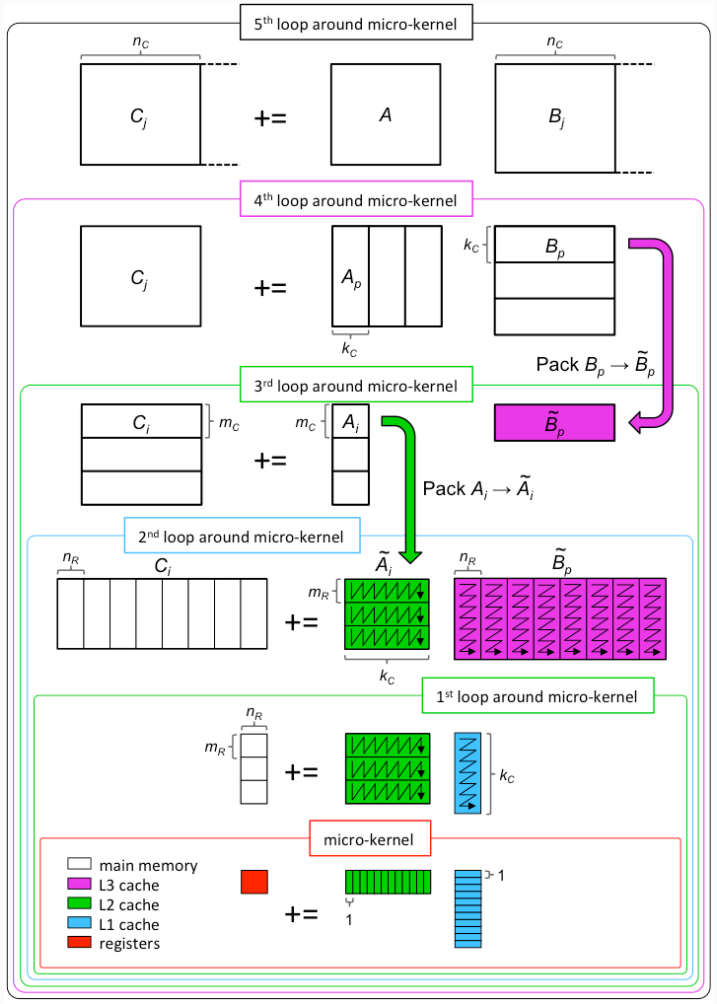
Key observation: Multi-vector algorithms are “a thing”

- Logically rectangular “tiles” (as in the new Intel AMX feature)
  - Show up in optimized “BLAS microkernels” (as in BLIS, GotoBLAS).
  - C[m,n] += Sum[k] A[m,k] * B[k,n] for m,n small, like lane counts (4/6/8)
  - Key operations: Outer product (A[*,k] against B[k,*]), then add into C[*,*] for each k
  - Motion along either logical dimension: replicate, reduce, reorder, transpose
- Loop unrolling: Use “as many registers as you can get away with”
  - Requires testing to see how many that might be (playing “The Price is Right” in vector register file)
  - Easier to configure as a multi-vector shape, not “unroll and jam” of source code
  - This technique combines with tiles: Keep adding tiles until we run out of registers
  - Common operation: Partial reduction (N-way unroll ⇒ N-to-1 reduction of accumulator)
- Vector-of-structs: A vector of three-element structs should be three vectors.
  - Common operation: Transpose (SoV ⇔ VoS), other ad hoc shuffles.

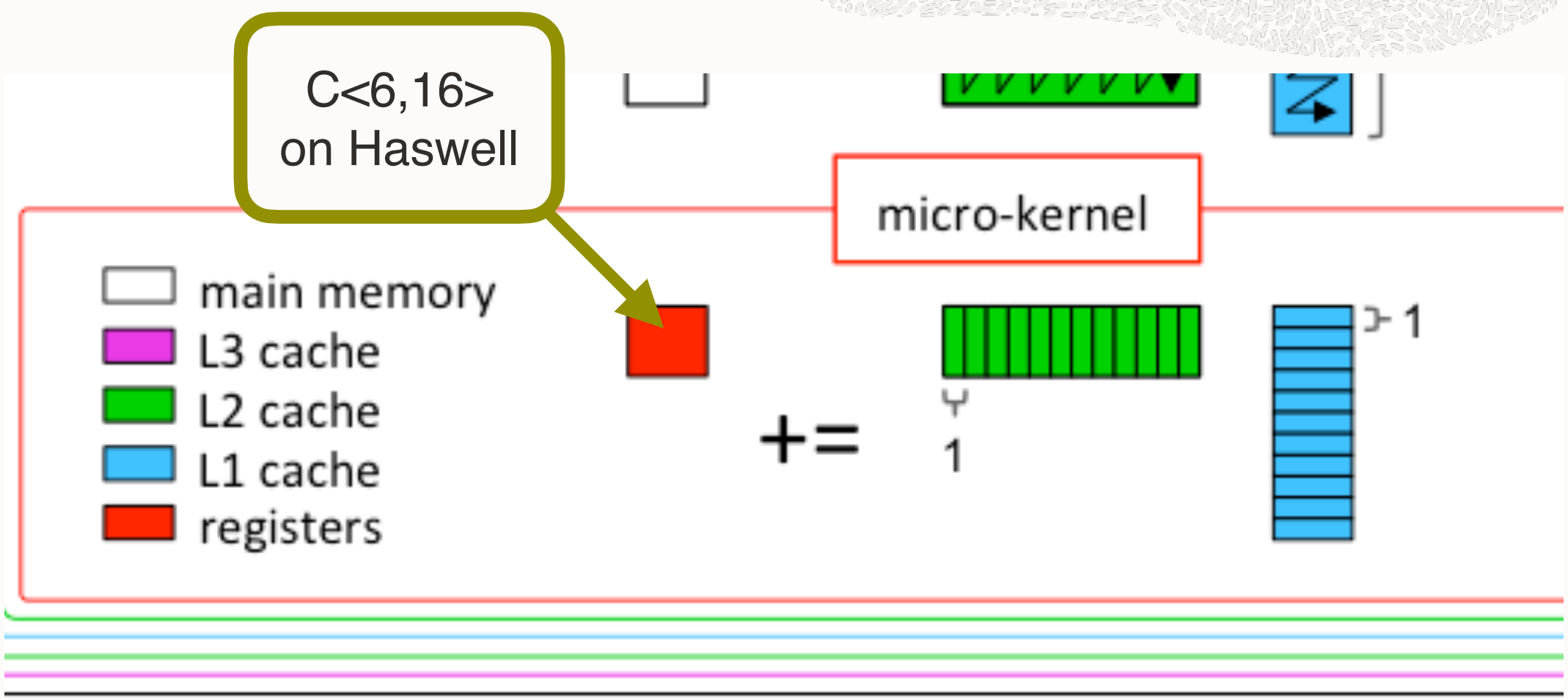
Some basic operations for multi-vectors

- Native vector species (VLEN1) replicated by some small RLEN, VLEN2=VLEN1 x RLEN
- All the usual lanewise stuff. Also slicing, compression, (full) reduction. Masking (with big masks).
- Shuffle (with a jumbo index array)
  - Some set patterns like transpose, zip/unzip, SoV / VoS
  - Support partial broadcast (a,b,c…) to (a,a,a,b,b,b,c,c,c…) or (a,b,c…,a,b,c…,a,b,c…)
  - Compiles to RLEN single-vector shuffles, possibly with RLEN merges.
  - Partial broadcast can set up an outer product (matrix tile multiply/FMA)
- Reshaping shuffle (expansion/contraction) is probably important
  - Generalizes today's pick-from-two-vectors shuffle operation
  - Use to extract or insert or broadcast VLEN1-vector against VLEN2 vector
- Striding partial reduction = (a,b,c,d,e,f) to (a+c+e, b+d+f) (reduce by RLEN)
  - Other orders of reduction are best obtained by a previous shuffle?

Challenge problem: BLIS microkernel



Challenge problem: BLIS microkernel



Challenge problem: BLIS microkernel

The Bliss SGEMM microkernel for Haswell is here:
https://github.com/flame/blis/blob/HEAD/kernels/haswell/3/bli_gemm_haswell_asm_d6x8.c#L177

```
// C<96> += A<16> * B<6>
// [ymm4..ymm15]<96> += [ymm0..ymm1]A<16> * [(ymm2..ymm3)x3]B<6>
vbroadcastss(mem(rbx, 6*4), ymm2)  vbroadcastss(mem(rbx, 7*4), ymm3)
vfmadd231ps(ymm0, ymm2, ymm4)  vfmadd231ps(ymm1, ymm2, ymm5)
vfmadd231ps(ymm0, ymm3, ymm6)  vfmadd231ps(ymm1, ymm3, ymm7)
vbroadcastss(mem(rbx, 8*4), ymm2)  vbroadcastss(mem(rbx, 9*4), ymm3)
vfmadd231ps(ymm0, ymm2, ymm8)  vfmadd231ps(ymm1, ymm2, ymm9)
vfmadd231ps(ymm0, ymm3, ymm10) vfmadd231ps(ymm1, ymm3, ymm11)
vbroadcastss(mem(rbx, 10*4), ymm2) vbroadcastss(mem(rbx, 11*4), ymm3)
vfmadd231ps(ymm0, ymm2, ymm12) vfmadd231ps(ymm1, ymm2, ymm13)
vfmadd231ps(ymm0, ymm3, ymm14) vfmadd231ps(ymm1, ymm3, ymm15)
```

Challenge problem: BLIS microkernel

```
// C<96> += A<16> * B<6>
// [ymm4..ymm15]<96> += [ymm0..ymm1]A<16> * [(ymm2..ymm3)x3]B<6>

var A_SP = FloatVector.SPECIES_256.times(2);  //<16>
var B_SP = FloatVector.SPECIES_64.times(3);   //<6>
var C_SP = FloatVector.SPECIES_256.times(12); //<96>
var C_V = FloatVector.fromArray(C_SP, C_ARR, cIndex); cIndex += 96;
var A_V = FloatVector.fromArray(A_SP, A_ARR, aIndex); aIndex += 16;
var AREPL_V = C_V.broadcastCyclic(A_V);
for (;;) { …
  var B_V = FloatVector.fromArray(B_SP, B_ARR, bIndex);  bIndex += 6;
  var BSPREAD_V = C_V.broadcastExpand(B_V);
  C_V = C_V.fma(AREPL_V, BSPREAD_V);
}
```

Challenge accepted: Paul Sandoz writes it w/o multi-vectors

https://mail.openjdk.java.net/pipermail/panama-dev/2021-March/012664.html

```
// Row 0: AB += OldC x beta
AxB_0L = vOldCL.fma(vbr_beta, AxB_0L);
AxB_0R = vOldCR.fma(vbr_beta, AxB_0R);

// Save results to free up registers:
AxB_0L.intoArray(c, c_row_iter);
AxB_0R.intoArray(c, c_row_iter + 8);
c_row_iter += rs_c;

// Row 1: AB += OldC x beta
vOldCL = FloatVector.fromArray(SPECIES_256, c, c_row_iter);
vOldCR = FloatVector.fromArray(SPECIES_256, c, c_row_iter + 8);
AxB_1L = vOldCL.fma(vbr_beta, AxB_1L);
AxB_1R = vOldCR.fma(vbr_beta, AxB_1R);
AxB_1L.intoArray(c, c_row_iter);
AxB_1R.intoArray(c, c_row_iter + 8);
c_row_iter += rs_c;
```

Cross-lane data motion —general observations

Kinds of cross-lane motion

- Random access shuffle: *Pulling* values to shuffle indexes which *attract* them.
  - Completely general; use when all else fails or to simulate
- Sidelong slide: Slipping back and forth, *maintaining stable order* of active set
  - Compress, expand, *sheep-and-goats (bidirectional compress)
  - Useful for quick-sort pivot, collecting *conditional* loop outputs
  - Compress/expand is useful for parsing (expand regularizes token sizes)
- Sorting/binning: *Pushing* values to be consistent with *key order*
  - *Sort, can be simulated by 2lg(N) compress operations (doing radix sort)
  - Complementary to random access shuffle (“reverse data flow”)
  - *Binning is sort followed by reduction of items with the same key
  - For such operations, a key vector drives reordering of a data vector
- **Research problem: Co-design of software and hardware**

Challenge problems: Parsing (using sidelong compress/expand)

- Load a block of UTF8 bytes
- Find the token boundaries (make a mask)
- Expand tokens to uniform size (24 or 32 bits)
- Perform bit-twiddling to normalize token to UTF16 values
- Compress to 16-bit lanes.
- Profit.

OR:

- Load a block of ASCII characters.
- Find natural language word boundaries.
- Expand tokens to uniform size (8 bytes, = max word size?)
- Strip out punctuation, white space; normalize case; visit the dictionary or histogram…

Appendix: What if my hardware doesn't do expand natively?

- Expand can be implemented as a by-index permutation (table lookup) plus a horizontal prefix-sum.
- First compute an index vector for the desired expansion.
- The index vector increases monotonically to the left, with a "bump" just **after** each mask position.
  - Example: Mask (0,0,1,0,0,1,1,0) derives the index vector (0,0,0,1,1,1,2,3).
  - Each index position sums up the total number of mask bits set **to the left**.
  - May require a series of lg(N) horizontal additions at various (power of two) offsets.
- Then perform a full index-based permutation, using the computed index vector to steer the input values.
  - The monotonically increasing index vector "pulls" each input value "to the right" into the output.

Appendix: OK, so what if my hardware doesn't do compress natively?

- Compress can be implemented as a masked indexed store (scatter) to a memory temporary.
- First compute an index vector for the desired compression.
- The index vector increases monotonically to the left, with a "bump" just **after** each mask position.
  - (Same as the index vector for compress, on the previous page!)
  - For unmasked positions, maybe store distinct, non-overlapping placeholder indexes.
  - Example: Mask (0,0,1,0,0,1,1,0) might derive the index vector (-8,-7,0,-6,-5,1,2,-4).
- Then perform a full index-based scatter, using the computed index vector to steer the input values.
  - The output buffer should be extra-sized, with half of the positions usable as "bit buckets" (-8 etc.).
  - If there's masked store primitive, just mask off the unused indexes, instead of using placeholders.
- After the memory finishes swirling around, pick up the assembled compressed output from the buffer.

- (Bonus trick: The placeholder indexes suggest how do masked scatters on AVX2 which lacks them.)

Challenge problems: Sorting (the Missing Primitive)

- Load a block of unsorted data
- Pick a pivot (maybe from this block, or maybe from a previous pass)
- Run bidirectional compress to separate across pivot (= Sheep-and-Goats from Hacker's Delight)
- Store the partitioned sub-blocks, or immediately re-sort (since it's in registers)
- When tired of sorting, store data for later merging

MUCH LATER…

- Load 2 or more merge blocks
- Perform an all-to-all (outer product) comparison
- Use the resulting bits to steer the merge

BOTH PHASES USE A SORTING OPERATION: **How should this be coded?**

Challenge problems: Taming the Permutation Zoo

- Most permutations are “not very random”
- Is there some way to describe them concisely so that the compiler can select better instructions?
- Common permutations: zip/unzip, transpose, locally block-wise reordering, talk-to-neighbor (shift/slice)
- How to capture the regular ones? (The optimizer should “see through” shuffle expressions…)
- Connection to multi-vectors: Localized permutations can be expressed as less than R lg(R) shuffles
  - The general case requires lots of merge-ups (lg(R) merge tree), but the special ones are simple
  - The very common “partial broadcasts” (cyclic copy, expanding copy) have good locality
- Is there a better notation (transparent to optimizer), than an index vector, for the “easy” permutations?
- Data-driven shuffles (selectFrom lookups) should favor constant lookup tables.
  - Range or bitwise analysis on indexes should routinely remove range checks.
  - Recent case study: https://mail.openjdk.java.net/pipermail/panama-dev/2022-February/016337.html

RESEARCH PROBLEMS…

Catching snowflakes — getting to odd VPU instructions

Special data types

- Half-float (V)
- Bits128 (CLMUL, AES)
- Complex (many various component types!)

- These can probably be expressed as placeholder types: value objects
- The value object can have a single replicated field

```
value class ComplexFloat { @MultiField(2) float reim; }
value class ComplexDouble { @MultiField(2) double reim; }
value class Bits128 { @MultiField(2) long bits; }
value class Float16 { short bits; }
```

Special operations

- Carry-less multiply [p]clmul (V+S)
- aesenc/aesdec steps (V+S)
- vp2intersect: all-to-all (outer product) comparison with dual-dimension mask reduction
- vnni: 8/16b precision dot product, 32b precision accumulation (saturating!)
- AMX (matrix tiles); tile multiplication, dot product (like vnni)
- FMA, sometimes with odd parameter order or alternating add+sub variation
- Bit-permutation
  - PDEP, PEXT (S only, not V)
  - Reverse bits, reverse bytes, (rotate? transpose?)
- many ad hoc permutations and special loads (too many to keep track of)
  - vpermilps (simple butterfly-like shift within 128-bit mega-lanes); vperm2f128

Suggestion: Incubate snowflake intrinsics in JVM-only package

```
package jdk.internal.vm.data;

@RegisterAllocation(128) public value class Bits128 {
  private @MultiField(2) lo_hi;
  public static Bits128 make(long hi, long lo) { … }

  public static Bits128 bitwiseAnd(Bits128 a, Bits128 b) { … }
  public static Bits128 bitwiseOr(Bits128 a, Bits128 b) { … }
  public static Bits128 bitwiseXor(Bits128 a, Bits128 b) { … }

  public static Bits128 multiplyFull(long a, long b) { … }
  public static Bits128 multiplyCarryless(long a, long b) { … }
  public static Bits128 aesEncodeStep(Bits128 state, Bits128 key) { … }
}
```

Suggestion: Incubate nascent new primitive and vector types similarly

```
package jdk.internal.vm.data.halffloat;

public class HalfFloat { …
  public static HalfFloat add(HalfFloat a, HalfFloat b) { … }
  public static HalfFloat sub(HalfFloat a, HalfFloat b) { … }
  public static HalfFloat mul(HalfFloat a, HalfFloat b) { … }
  public static HalfFloat sqrt(HalfFloat a) { … }
}

public class HalfFloatVector extends AbstractVector<HalfFloat> { … }
public class HalfFloat128Vector extends HalfFloatVector { … }
public class HalfFloat256Vector extends HalfFloatVector { … }
public class HalfFloat512Vector extends HalfFloatVector { … }
```

Half-baked suggestion: Incubate nascent vector lane operations

```
package jdk.internal.vm.data;

public class ExtraVectorOperators extends VectorOperators { …
  public static final Associative ADD_HALF_FLOAT = …;
  public static final Binary AES_ENC_STEP = …;
  public static final Binary CL_MUL_HI = …;
  public static final Binary CL_MUL_LO = …;
}

// Not clear this pans out.  The point is a proving ground for ad hoc ops.
```

Vector atomics
— what might be useful

The underlying realities of memory operations (an educated guess)

- Memory travels in cache blocks (with update masks)
- The CPU decomposes cache operations into smaller units, tracking data as scalars (words, bytes, …)
- Existing locking operations briefly pin cache blocks and suppress decomposition
  - Sometimes they pin *adjacent* blocks when a locked operation crosses a line

- Presumably there are difficulties with suppressing decomposition of vectors
  - Decomposing memory operations supports value numbering and reordering
  - Suppressing decomposition breaks those optimizations
  - Decomposing operations also allows more flexible fit to internal queue limits
  - Suppressing decomposition requires more resource allocation for internal queues

- Still, it is possible, *in principle,* to consider cache-locked versions of vector operations

What cache-locked vector operations might look like

- Locked vector load, unmasked
  - Allocate any load queue resources, lock one or two cache lines
  - Transfer indicated data lanes into value tracking registers; release cache
- Locked vector load, masked — similar to unmasked
  - Might lock only one cache line where unmasked would lock two
- Locked vector store, unmasked
  - Fetch unwritten parts of cache line(s), wait for all CPU data ready
  - Allocate any store queue resources, lock one or two cache lines
  - Drain store queue entries under lock; release cache
- Locked vector store, masked — similar to unmasked
  - Might lock only one cache line

Existing instructions that might help

- movdqa/movdqu: Maskable; transfers up to 512 bits. Not lockable?
  - *Mostly* atomic? (Therefore *slightly* non-atomic.)
  - (Very Naive Feature Request: Allow these guys to be lock-prefixed!)
- lock cmpxchg16b: Works on *aligned* 64-bit word pairs. Slow. Improvable?
  - Could possibly help with 2-word value types.
  - Might be usable as a plain atomic load if an impossible test-value is set up.
  - Might be usable as a plain atomic store if a likely test-value is set up.
- movdir64b: Direct cache-bypassing store for 512 bits. Slow; requires fence.
  - Write-atomic, aligned. Not read-atomic; no support for reads.
  - movdiri: Like movdir64b, but 32/64 bits. Cache-bypassing, requires fence.
- TSX instructions: Rocky history. Excessive power: multiple cache lines (DCAS/MCAS).

What cache-locked vector CAS might look like

- Two vectors A, B, one mask K, one address M, returns C
  - Allocate any queue resources, lock one or two cache lines
  - Load C=(M){K}, compare to C and A under mask K
  - Store (M){K}=B under mask, but only if C{K}=A{K}
  - Drain queues; release cache

- Two vectors A, B, *two* masks K, K2, one address M, returns C
  - Similar to above, but store (M){K2}=B under *second* mask
  - Possible restriction: test mask K is limited in size/shape

- Memory transaction is always one or two-adjacent cache lines
  - Not a general DCAS/CAS2; more like a “whole cache line” CAS
  - Similar also to CMPXCHG16B, probably slow like that one.

Use cases

- Java volatile variable, with flat value >64 bits.
  - Struct-tearing is a (rare) possibility to protect against
  - Alternative to SW transactional memory (side-locks)
- Java extended VarHandle operations, on flat value >64 bits.
- Reading/writing/linking inter-thread communication or coprocessor command blocks (in user space)
- Concurrent version control on the same cache line (nR/1W SeqLock, for example).

- The alternative is object versioning: Use an indirection, CAS-patch a 64-bit pointer to current version.
- For Java volatiles, each version of the variable is a freshly allocated “buffer” object.
- One cost: Indirection harms locality (extra cache-line fetches).
- Another cost: Garbage collector has to go around cleaning up unused indirection blocks.

Plan of record

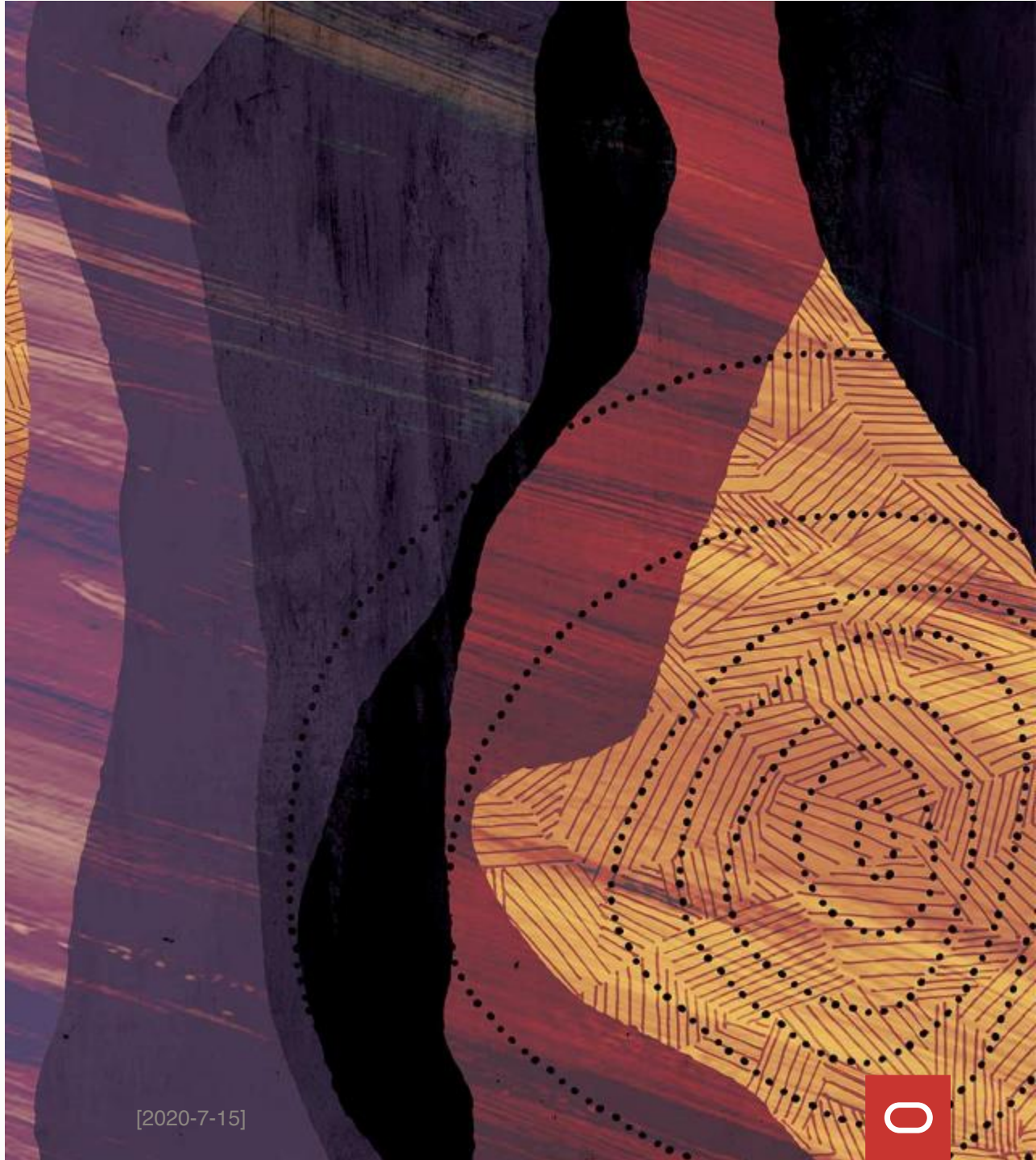
- Rely *only* on 64-bit memory atomicity.
- When in doubt, use object versioning, extra buffering, indirections.
- Invest more aggressively in one-word representations.
  - Example: value class Optional<T> { Object valOrSentinel; }
  - Instead of: value class Optional<T> { T val; boolean isPresent; }

- (Any guidance from the hardware gurus?)

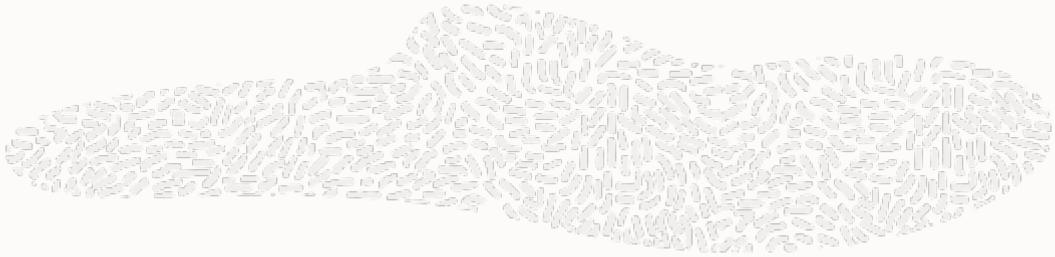
Thank You

Questions? Comments?

http://cr.openjdk.java.net/~jrose/pres/202202-VectorTopics.pdf



Safe Harbor



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, timing, and pricing of any features or functionality described for Oracle's products may change and remains at the sole discretion of Oracle Corporation.

ORACLE

Our mission is to help people
see data in new ways, discover insights,
unlock endless possibilities.