**Float16 Support in the OpenJDK Vector API**

**Summary**

Add FP16 (IEEE‑754 half‑precision) vector types to the Vector API, enabling compute and memory operations over half‑precision lanes with short carrier and Float16 box type. Provide C2 support, baseline/fallback semantics via FP32 promotion, and validation via jtreg/JMH.

**Goals**

* Introduce HalffloatVector with 64/128/256/512‑bit concrete species.
* Preserve the Vector API’s carrier model (short) while disambiguating FP16 from ShortVector via Float16 box type and an explicit *operation type*.
* Enable core ops: load/store, lanewise (unary/binary/ternary incl. FMA), compare, mask operations, broadcasts, splats.
* Provide deterministic fallback semantics using FP32 compute + FP16 round‑to‑nearest‑even (RNE) down‑conversion.

**Non‑Goals**

* No new hardware backends in this RFC (use existing C2 instruction selection patterns for respective backends and extend where necessary).
* No FP8/INT8 semantics in this change (but design leaves a path).
* No changes to Float16 scalar APIs beyond what’s required for interop.

**Motivation**

* FP16 is prevalent in ML/AI, image/video, and DSP for higher throughput and reduced bandwidth/footprint.
* Many architectures expose native FP16 SIMD; exposing this via Vector API unlocks portable performance while keeping well‑defined fallbacks.

**Design Overview**

**Java Types**

* Abstract: HalffloatVector
* Concrete: Halffloat64Vector, Halffloat128Vector, Halffloat256Vector, Halffloat512Vector
* ElementType (carrier): short.class
* BoxType: Float16 (avoids ambiguity with Vector<Short> and virtual dispatch)

**Fallback Semantics**

* Promote each short lane via Float16.floatValue() → compute in FP32 → down‑convert to FP16.

**Operation Type (disambiguation)**

* Pass an additional int operationType to VectorSupport entry points (e.g., VECTOR\_TYPE\_FP16 for now and VECTOR\_TYPE\_INT8, VECTOR\_TYPE\_FP8 in future).
* Carrier type remains T\_SHORT for TypeVect; IR opcode inference uses (operationType, opKind).

**HotSpot/C2 Integration (selected)**

* **Load/Store**  
  Entry: VectorSupport.load/store with (vectortype=Halffloat\*Vector.class, elemtype=short.class, length=N, operationType=VECTOR\_TYPE\_FP16)  
  Expander: LoadVector/StoreVector using TypeVect{element\_basic\_type=T\_SHORT, num\_elem=N}; existing short‑vector match rules apply.
* **Lanewise**  
  Entry: VectorSupport.unaryOp/binaryOp/ternaryOp (+ operationType).  
  IR: Reuse existing vector IR where backend ops exist; add FP16‑specialized nodes only where needed. (Today C2 creates specialized FP16 nodes; we continue that path.)
* **Compare/Mask**  
  Entry: VectorSupport.compare (+ operationType).  
  IR: Introduce VectorMaskCmpHFNode (semantic compare in FP16 domain).
* **Incubation Note**  
  Halffloat\* lives in the incubation module; hence in order to infer Float16 IR inline expander can take two approaches. One is to infer an Float16 IR through a name-based resolution, since VM only keep track of classed part of java.base module, this solution is acceptable in short‑term. However, much robust solution is to pass an explicit operationType parameter to intrinsic entry points as discussed above. This scheme circumvent loopholes in fragile name based resolution and can easily be extended to support other reduced precision types like INT8 or FP8 in future.

**Compatibility & Interactions**

* Distinct BoxType=Float16 prevents dispatch ambiguity with Vector<Short>.
* Interop: explicit conversion via Float.float16ToFloat(short) / Float.floatToFloat16(float).
* No behavioral changes to existing vector types.

**Testing & Validation**

* **Functional**: Extend Vector API jtreg to cover all Halffloat ops (loads/stores, lanewise incl. FMA, compares, masks, predication).
* **Performance**: Extend JMH harness; add microbenchmarks (e.g., FP16 dot‑product, semantic search kernels).
* **Correctness**: Cross‑check fallback vs hardware where available; verify RNE rounding and edge cases (NaNs, subnormals, signed zero, infinities).

**Risks & Mitigations**

* **Dispatch ambiguity** → Use Float16 box type and operationType flag.
* **Backend coverage variance** → Fall back to FP32 emulation; gated intrinsics per‑platform.
* **Precision surprises** → Document FP16 semantics, rounding, and conversions.

**Reference Implementation Plan**

1. Introduce Java types + species and wire to VectorSupport with operationType.
2. Implement C2 decode of (carrier=T\_SHORT, operationType=FP16) to FP16 IR.
3. Add VectorMaskCmpHFNode and ensure matcher patterns map to existing short/FP16 backends.
4. Land jtreg/JMH; publish perf/correctness data.

**Minimal Usage Sketch**

static final VectorSpecies<Float16> S = Halffloat.SPECIES\_PREFERRED;

short[] a = ...; short[] b = ...;

HalffloatVector acc = HalffloatVector.broadcast(S, 0);

for (int i = 0; i < S.loopBound(a.length); i += S.length()) {

var v1 = HalffloatVector.fromArray(S, a, i);

var v2 = HalffloatVector.fromArray(S, b, i);

acc = acc.lanewise(VectorOperators.FMA, v1, v2, acc);

}

float sum = 0.0f;

for (int i = 0; i < S.length(); i++) {

sum += Float.float16ToFloat(acc.lane(i));

}

return sum;

**Future Activities:-**

* Scope of this RFC is to extend array backing storage based existing vector API infrastructure to support Float16 type and enable Java users to harness the power of FP16 ISA supported by various targets and bring Float16 support at par with existing primitive Vector types.
* Our eventual goal is to make Float16 a value type and use flat array based backing storage supported by project Valhalla.

**Optimized Fallback Implementation of FP16 lanewise operations on non-AVX512-FP16 targets**

* ***hvec1.lanewise(VectorOperators.ADD, hvec2)***
* ***Fallback implementation should be auto-vectorizable, but to avoid boxing penalty it always advisable to inline the fallback implementation. Newly proposed hybrid call generator part of optimized slice implementation should facilitate this.***
* ***Alternatively, we directly write the fallback code using vector APIs. Only thing which needs to be accounted for is that HalffloatVector have 16 bit lane size, while FloatVector operate at 32 bit lane granularity.***

**Intrinsic Entry points modification**

***Current: -***

*public interface BinaryOperation<VM extends VectorPayload,*

*M extends VectorMask<?>> {*

*VM apply(VM v1, VM v2, M m);*

*}*

*@IntrinsicCandidate*

*public static*

*<VM extends VectorPayload,*

*M extends VectorMask<E>,*

*E>*

*VM binaryOp(int oprId,*

*Class<? extends VM> vmClass, Class<? extends M> mClass, Class<E> eClass,*

*int length,*

*VM v1, VM v2, M m,*

*BinaryOperation<VM, M> defaultImpl) {*

*assert isNonCapturingLambda(defaultImpl) : defaultImpl;*

*return defaultImpl.apply(v1, v2, m);*

*}*

***Desired: -***

*public interface BinaryOperation<VM extends VectorPayload,*

*M extends VectorMask<?>> {*

*VM apply(VM v1, VM v2, M m);*

*}*

*@IntrinsicCandidate*

*public static*

*<VM extends VectorPayload,*

*M extends VectorMask<E>,*

*E>*

*VM binaryOp(int oprId,*

*Class<? extends VM> vmClass, Class<? extends M> mClass,* ***Class<E> cClass,***

***Class<?> eClass,***

*int length,*

***int operType,***

*VM v1, VM v2, M m,*

*BinaryOperation<VM, M> defaultImpl) {*

*assert isNonCapturingLambda(defaultImpl) : defaultImpl;*

*return defaultImpl.apply(v1, v2, m);*

*}*

***Pass a new carrier type which truly represents the backing storage element type, additionally pass an elementType which is used to infer the boxtype of the Vector to disambiguate virtual dispatch.***

***For all primitive type vector other than Float16, carrier and element type are same. While operType will help expander in vector IR inferencing.***

***cClass must always be a primitive type, else inline expansion fails, existing inline expanders already check this.***

***bool LibraryCallKit::inline\_vector\_nary\_operation(int n) {***

***bool LibraryCallKit::inline\_vector\_call(int arity) {***

***bool LibraryCallKit::inline\_vector\_mask\_operation() {***

***bool LibraryCallKit::inline\_vector\_frombits\_coerced() {***

***bool LibraryCallKit::inline\_vector\_mem\_operation(bool is\_store) {***

***bool LibraryCallKit::inline\_vector\_mem\_masked\_operation(bool is\_store) {***

***bool LibraryCallKit::inline\_vector\_gather\_scatter(bool is\_scatter) {***

***bool LibraryCallKit::inline\_vector\_reduction() {***

***bool LibraryCallKit::inline\_vector\_test() {***

***bool LibraryCallKit::inline\_vector\_blend() {***

***bool LibraryCallKit::inline\_vector\_compare() {***

***bool LibraryCallKit::inline\_vector\_rearrange() {***

***bool LibraryCallKit::inline\_vector\_select\_from() {***

***bool LibraryCallKit::inline\_vector\_broadcast\_int() {***

***bool LibraryCallKit::inline\_vector\_convert() {***

***bool LibraryCallKit::inline\_vector\_insert() {***

***bool LibraryCallKit::inline\_vector\_extract() {***

***bool LibraryCallKit::inline\_vector\_select\_from\_two\_vectors() {***

***bool LibraryCallKit::inline\_vector\_compress\_expand() {***