Column Filter optimization using Vector.compress API

(Vector vs memory segment source)

**Microbenchmark mimicking vectorized column filter for ORC presto connector.**

import jdk.incubator.vector.\*;

import java.util.Arrays;

import java.lang.foreign.\*;

import java.nio.ByteOrder;

public class FloatColumnFilter {

public static final VectorSpecies<Float> SPECIES = FloatVector.SPECIES\_PREFERRED;

public static final int INVOC\_COUNT = 100000;

public static boolean debug = false;

public static MemorySegment ms = null;

static int ScalarFloatColumnFilter(float[] a, float[] r, float threshold) {

int j = 0;

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

if (a[i] >= threshold) {

r[j++] = a[i];

}

}

return j;

}

static int VectorFloatColumnFilter(float[] a, float[] r, float threshold) {

int j = 0;

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

FloatVector av = FloatVector.fromArray(SPECIES, a, i);

VectorMask<Float> vmask = av.compare(VectorOperators.GE, threshold);

int trueCount = vmask.trueCount();

av.compress(vmask).intoArray(r, j);

j += trueCount;

}

return j;

}

static int VectorFloatColumnFilterMS(MemorySegment ma, MemorySegment mr, float threshold) {

int j = 0;

for (int i = 0; i < ma.byteSize(); i += (SPECIES.length() << 2)) {

FloatVector av = FloatVector.fromMemorySegment(SPECIES, ma, i, ByteOrder.nativeOrder());

VectorMask<Float> vmask = av.compare(VectorOperators.GE, threshold);

int trueCount = vmask.trueCount();

av.compress(vmask).intoMemorySegment(mr, j << 2, ByteOrder.nativeOrder());

j += trueCount;

}

return j;

}

static float sumMs(MemorySegment ms, int len) {

float r = 0.0f;

for(int i = 0; i < len; i++) {

r += ms.get(ValueLayout.JAVA\_FLOAT, i << 2);

}

return r;

}

static float sumArr(float [] arr, int len) {

float r = 0.0f;

if (debug) {

System.out.println("compressed dst ===>");

System.out.println(Arrays.toString(arr));

}

for(int i = 0; i < len; i++) {

r += arr[i];

}

return r;

}

public static void main(String [] args) {

int size = Integer.parseInt(args[0]);

float selectivity = Float.parseFloat(args[1]);

int algo = Integer.parseInt(args[2]);

//debug = Boolean.parseBoolean(args[3]);

debug = false;

System.out.println("Size = " + size);

System.out.println("Selectivity = " + selectivity);

System.out.println("Algo = " + (algo == 0 ? "Scalar" : "Vector"));

System.out.println("Debug = " + debug);

float [] a = new float[size];

float [] r = new float[size];

float threshold = 10.0f;

MemorySegment ma = MemorySegment.allocateNative(size\*4, 64, SegmentScope.global());

MemorySegment mr = MemorySegment.allocateNative(size\*4, 64, SegmentScope.global());

for(int i = 0; i < size; i++) {

if (i >= selectivity\*size) {

ma.set(ValueLayout.JAVA\_FLOAT, i << 2, threshold);

}

}

Arrays.fill(a, threshold);

System.arraycopy(r, 0, a, 0, (int)(selectivity\*size));

if (debug) {

System.out.println("init src ===>");

System.out.println(Arrays.toString(a));

System.out.println("init dst ===>");

System.out.println(Arrays.toString(r));

}

if (algo == 0) {

int iters = 0;

for(int i = 0 ; i < INVOC\_COUNT ;i++) {

iters += ScalarFloatColumnFilter(a, r, threshold);

}

iters = 0;

long t1 = System.currentTimeMillis();

for(int i = 0 ; i < INVOC\_COUNT ;i++) {

iters += ScalarFloatColumnFilter(a, r, threshold);

}

long t2 = System.currentTimeMillis();

float res = sumArr(r, iters / INVOC\_COUNT);

System.out.println("[Scalar Time] " + (t2-t1) + "ms [Res] " + res);

} else if (algo == 1) {

int iters = 0;

for(int i = 0 ; i < INVOC\_COUNT ;i++) {

iters += VectorFloatColumnFilter(a, r, threshold);

}

iters = 0;

long t1 = System.currentTimeMillis();

for(int i = 0 ; i < INVOC\_COUNT ;i++) {

iters += VectorFloatColumnFilter(a, r, threshold);

}

long t2 = System.currentTimeMillis();

float res = sumArr(r, iters / INVOC\_COUNT);

System.out.println("[Vector Time] " + (t2-t1) + "ms [Res] " + res);

} else {

int iters = 0;

for(int i = 0 ; i < INVOC\_COUNT ;i++) {

iters += VectorFloatColumnFilterMS(ma, mr, threshold);

}

iters = 0;

long t1 = System.currentTimeMillis();

for(int i = 0 ; i < INVOC\_COUNT ;i++) {

iters += VectorFloatColumnFilterMS(ma, mr, threshold);

}

long t2 = System.currentTimeMillis();

float res = sumMs(mr, iters / INVOC\_COUNT);

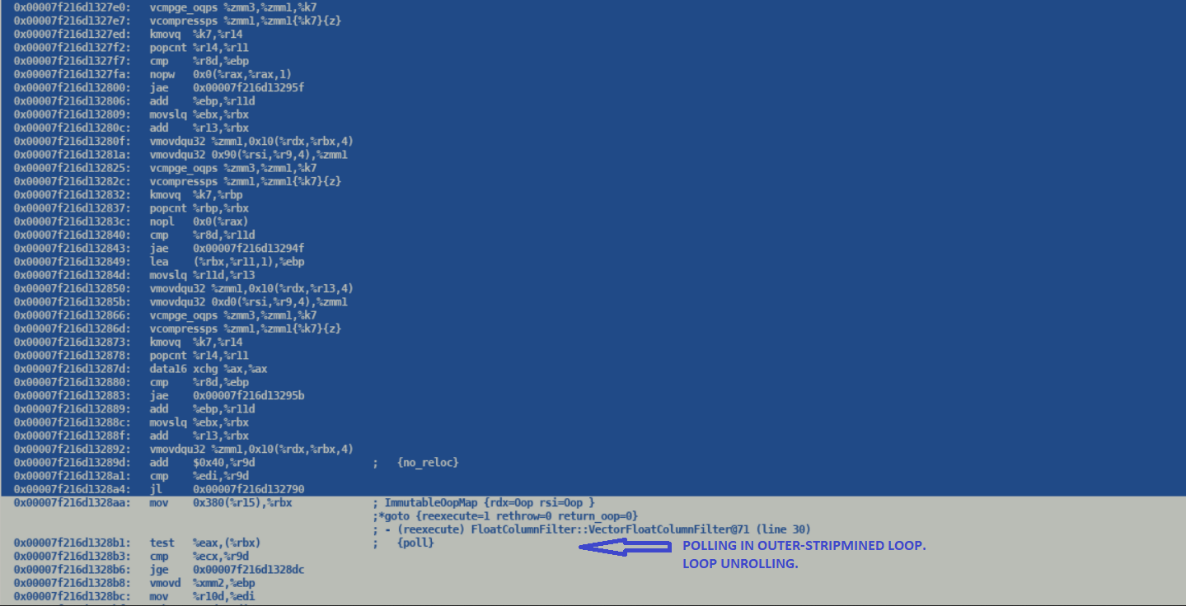
System.out.println("[VectorMS Time] " + (t2-t1) + "ms [Res] " + res);

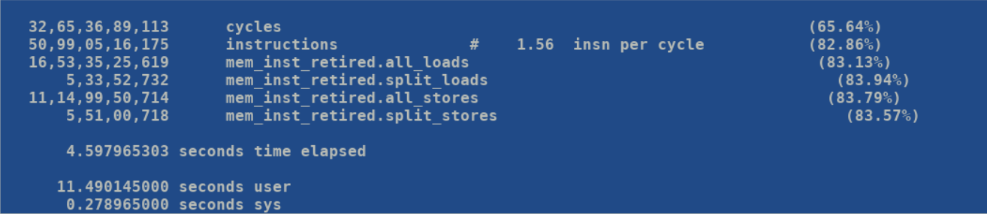
}

}

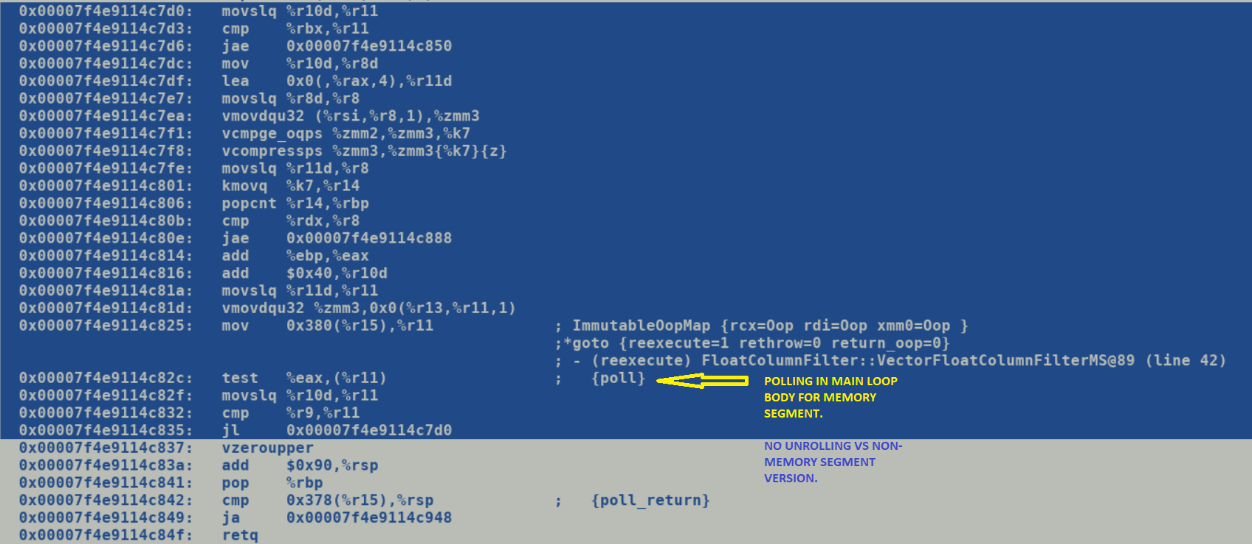
}

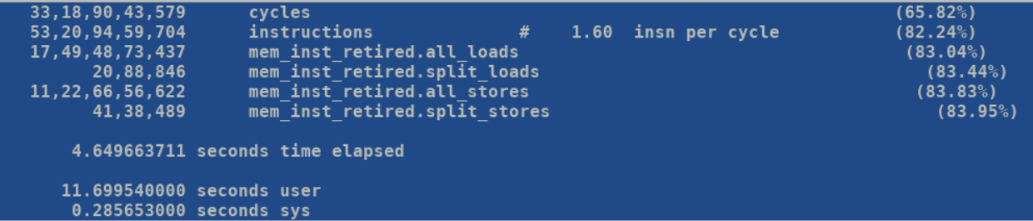
VectorFloatColumnFilter: JIT sequence shows unrolled main loop followed by safe point pool check in outer strip mined loop, this is because of successful iteration space splitting of CountedLoop and creation of pre-main-post loop.



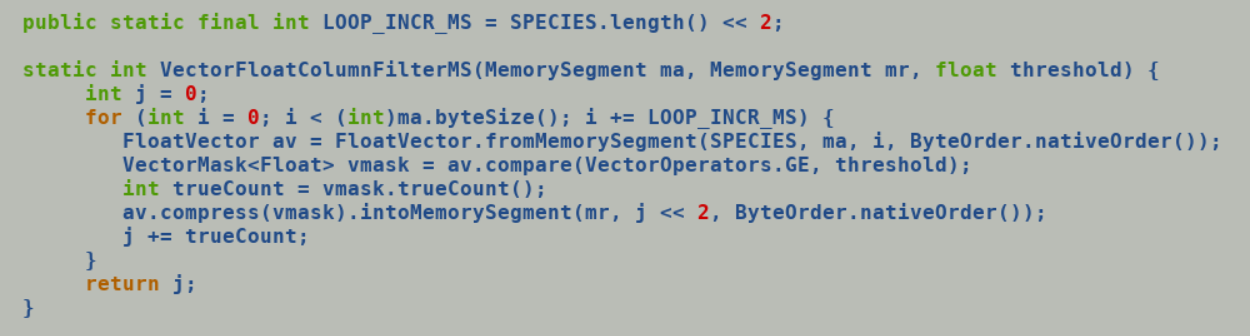


JIT excerpt of VectorFloatColumnFilterMS shows an unrolled loop with in-loop safe point polling, this happens for non-counted loops which prevents iteration space splitting.

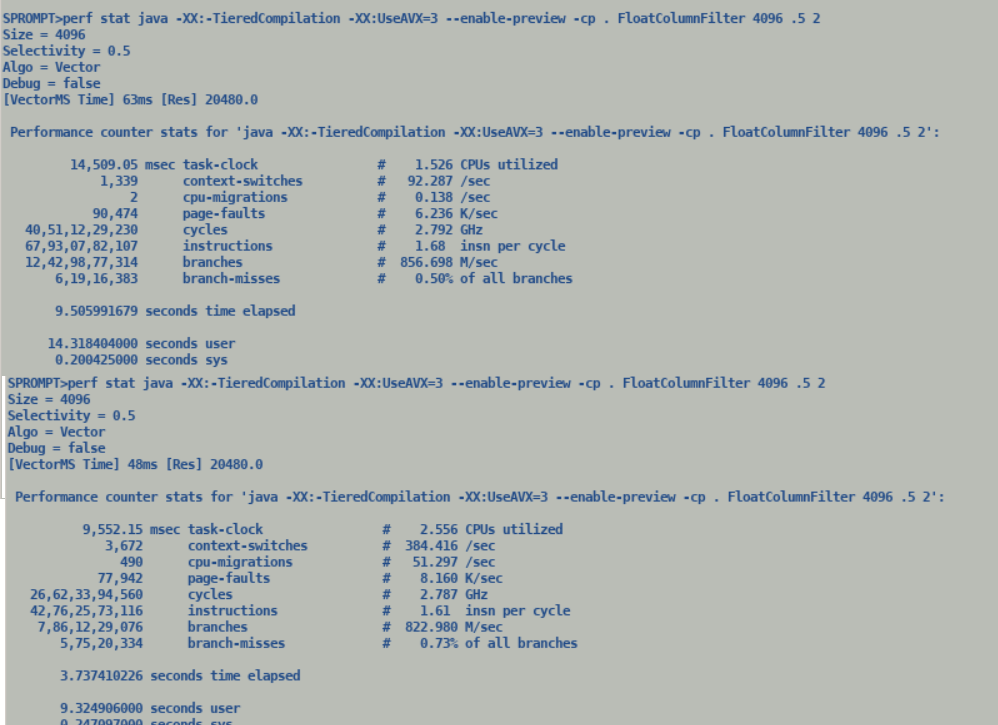




Modify the loop incrementation and limit conditions to favor C2’s counted loop detection which prevent creation of counted loops for long type trip counts.



Both iteration space splitting and unrolling is done only for counted loops.



Savings from redundant pooling show up in path length and over all latency improvement.

Recommended to always use Tiered Compilation since we do not want to have bloated compilation queues for C2 which may delay compilation of hot methods.