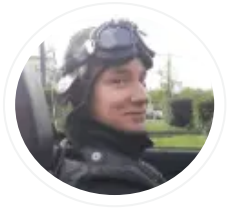


Piotr Nowojski

Software engineer at Teradata. Contributor to Prestodb.

Warsaw/Poland

Email

Twitter

GitHub

JAVA and SIMD

9 minute read

I have wanted to experiment with Java for a long time to find out whether or it can take advantage of Single Instruction, Multiple Data (SIMD) instructions speed up CPU-intensive computations. I found very little information while I was researching this, so I decided to share my own findings.

What are SIMD instructions?

SIMD instructions allow the CPU to perform the same operation on multiple values simultaneously. For example we would like to perform four multiplications on eight values:

```
z1 = x1 * y1
z2 = x2 * y2
z3 = x3 * y3
z4 = x4 * y4
```

Normally that would require eight instructions to load values from memory ir registers and four multiplication instructions. Using SIMD instructions, the C can load all four x values into the xmm0 with a single MOVUPS instruction,, another MOVUPS to load the four y values into the xmm1 register and a sin MULPS instruction to multiply them

```
+-------+-------+-------+-------+
|   x3  |   x2  |   x1  |   x0  | xmm0
+-------+-------+-------+-------+
    *       *       *       *
+-------+-------+-------+-------+
|   y3  |   y2  |   y1  |   y0  | xmm1
+-------+-------+-------+-------+
    =       =       =       =
+-------+-------+-------+-------+
| x3*y3 | x2*y2 | x1*y1 | x0*y0 | xmm0
+-------+-------+-------+-------+
```

The key feature here is that this multiplication will be performed simultaneou on all four values, which will be four times faster! Isn't that great? :) SIMD instructions are often called vectorized instructions, because you can think o them as operating on vectors of values.

The first SIMD instructions in desktop/server CPUs were introduced in 1996 Intel's MMX extension the Pentium processors. Afterwards those instruction were expanded by the SSE and AVX standards. Now it is safe to assume th almost every CPU has some level of SIMD support. Nevertheless it is impor to know whether your hardware supports SIMD operations that you want to use. For example many instructions operating on 64bit integers were added only in the latest AVX512 standard.

The problem

Let's take a step back and show this problem in a real-life engineering use case. PrestoDB, a distributed analytical SQL engine for Big Data (eg. large datasets in HDFS clusters), often has to partition the same data using the sa columns multiple times one after another. For example to perform a distribut hash JOIN algorithm, after reading the data from HDFS, Presto has to:

1. Distribute the rows among the worker nodes.
2. Within each worker, distribute the rows among CPU cores to further parallelize the execution
3. Put each row in a hash table bucket.

This creates multiple layers of distributions for which we have to ensure that rows with the same values in the key end up in the same bucket. Obviously

Presto cannot re-use same hash value at each step of the partitioning (otherwise only one bucket from 2. and 3. would be used). However calcula new hashes on each step can become a bottleneck, so Presto tries to simpl and optimize the hashing/scrambling algorithms as much as it is possible.

One trick is that in step 2., Presto computes the hash (let's call it rawHash) it does not have to re-calculate a complicated hash in the next step (3.). Inst we can re-use rawHash value by just scrambling its bits using some simple function. For this quick scramblling Presto uses the following code:

```
private static int getHashPosition(long rawHash, long mask)
{
    rawHash ^= rawHash >>> 33;
    rawHash *= 0xff51afd7ed558ccdL;
    rawHash ^= rawHash >>> 33;
    rawHash *= 0xc4ceb9fe1a85ec53L;
    rawHash ^= rawHash >>> 33;

    return (int) (rawHash & mask);
}
```

Despite being so simple it can sometimes be the most CPU-intensive operation. This makes getHashPosition function a perfect candidate for vectorization, because it could be calculated simultaneously for multiple rawHashes from consecutive rows.

Because this function uses 64 bit integers and during writing this blog I did n have an access to any CPU supporting AVX512, I have rewritten it to versio operating on 32 bit integers:

```
  private static int getHashPosition(int rawHash, int mask)
{
    rawHash ^= rawHash >>> 15;
    rawHash *= 0xed558ccd;
    rawHash ^= rawHash >>> 15;
    rawHash *= 0x1a85ec53;
    rawHash ^= rawHash >>> 15;

    return rawHash & mask;
}
```

Java and SIMD

As of Java 8, there is no way to use SIMD intrinsics in Java directly as can b done in C++ or C#, for example. In gcc we can define our data type to be a vector and perform SIMD operations directly as described in gcc documentation.

In C# there is a similar mechanism and one can use System.Numerics.

However, Java can also generate SIMD code under some conditions. If it detects that subsequent iterations of a loop perform independent calculatior Java can try to vectorize such loop. Roughly speaking, instead of doing this

```
    for (int i = 0; i < x.length; i++) {
    z[i] = x[i] * y[i];
}
```

Java can try to do this (some pseudo code):

```
 for (int i = 0; i < x.length; i += 4) {
    Load x[i, i+1, i+2, i+3] into xmm0
    Load y[i, i+1, i+2, i+3] into xmm1
    Multiply xmm0 * xmm1 and store result in xmm0
    Store xmm0 into z[i, i+1, i+2, i+3]
}
```

This optimization can be turned on/off by a JVM switch “-XX:+UseSuperWo
which is turned ON by default.

This should work fine with the getHashPosition function. For example, we c
pre-calculate those hashes in batches and store the results in a small array.
Batches should be of a reasonable size, so that our temporary array fits into
CPU caches. In the next section let’s try if this works out.

Vectorizing loop

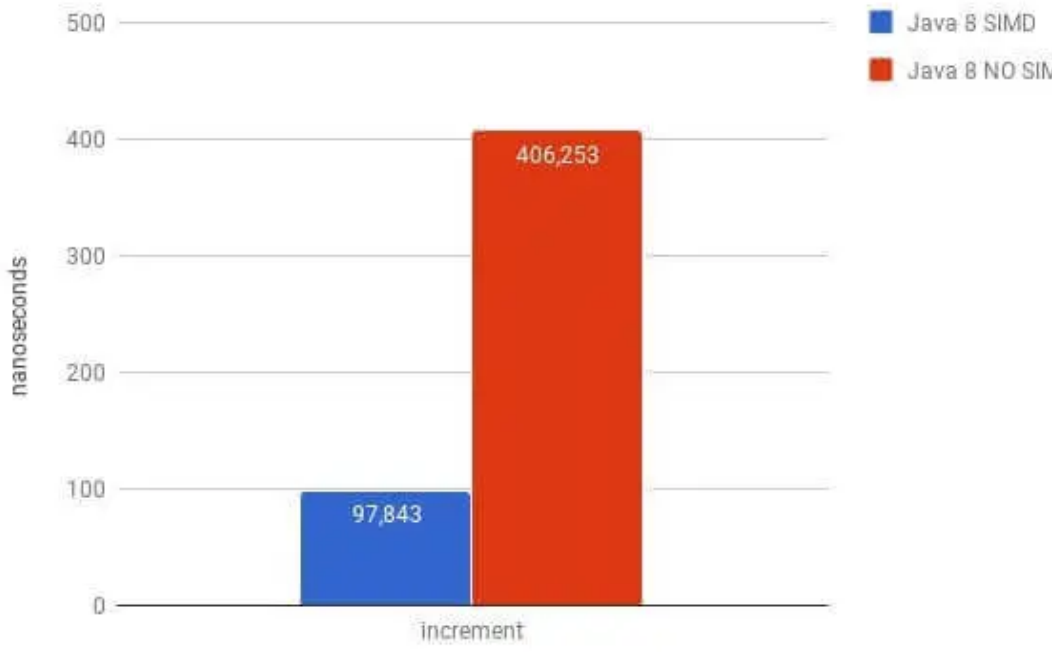
Simple incrementation

Let’s start with some simple loop over integer values. Our first benchmark is
incrementation of values in an array.

```
    @State(Thread)
@OutputTimeUnit(NANOSECONDS)
@BenchmarkMode(AverageTime)
@Fork(value = 1, jvmArgsAppend = {
        "-XX:+UseSuperWord",
        "-XX:+UnlockDiagnosticVMOptions",
        "-XX:CompileCommand=print,*BenchmarkSIMDBlog.array1"})
@Warmup(iterations = 5)
@Measurement(iterations = 10)
public class BenchmarkSIMDBlog
{
    public static final int SIZE = 1024;

    @State(Thread)
```

```
    public static class Context
    {
        public final int[] values = new int[SIZE];
        public final int[] results = new int[SIZE];

        @Setup
        public void setup()
        {
            Random random = new Random();
            for (int i = 0; i < SIZE; i++) {
                values[i] = random.nextInt(Integer.MAX_VALUE / 32);
            }
        }
    }

    @Benchmark
    public int[] increment(Context context)
    {
        for (int i = 0; i < SIZE; i++) {
            context.results[i] = context.values[i] + 1;
        }
        return context.results;
    }
}
```

JMH is used here for micro benchmarking. Results with -XX:-UseSuperWor and -XX:+UseSuperWord are the following:



That's great! Four times faster. Thanks to the -

XX:CompileCommand=print,*BenchmarkSIMDBlog.increment we can look a the code that JIT produced for this benchmark in both versions. With SuperWord we can easily find instructions from AVX2 extension that are responsible for this speedup:

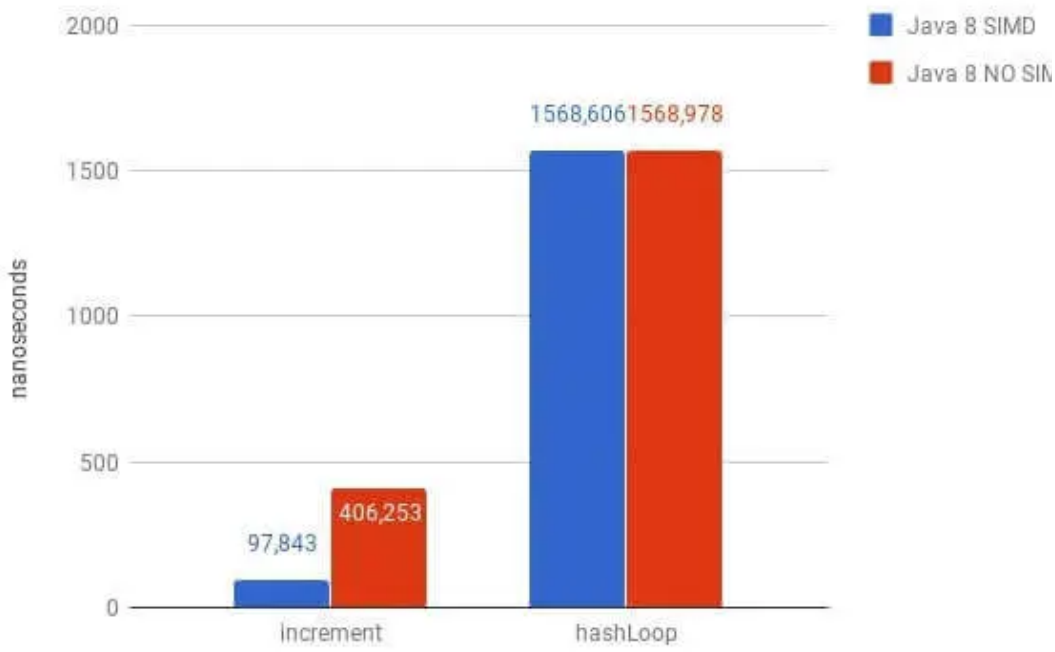
```
    0x00007f7354e59638: vmovq  -0xe0(%rip),%xmm0
    0x00007f7354e59640: vpunpcklqdq %xmm0,%xmm0,%xmm0
    0x00007f7354e59644: vinserti128 $0x1,%xmm0,%ymm0,%ymm0
    0x00007f7354e5964a: nopw   0x0(%rax,%rax,1)
    0x00007f7354e59650: vmovdqu 0x10(%r10,%r8,4),%ymm1
    0x00007f7354e59657: vpaddd %ymm0,%ymm1,%ymm1
    0x00007f7354e5965b: vmovdqu %ymm1,0x10(%r11,%r8,4)
```

Hashing integers

Now we can try vectorizing our getHashPosition method by adding another benchmark:

```
    @Benchmark
public int[] hashLoop(Context context)
{
    for (int i = 0; i < SIZE; i++) {
        context.results[i] = getHashPosition(context.values[i],
    }
    return context.results;
}

private static int getHashPosition(int rawHash, int mask)
{
    rawHash ^= rawHash >>> 15;
    rawHash *= 0xed558ccd;
    rawHash ^= rawHash >>> 15;
    rawHash *= 0x1a85ec53;
    rawHash ^= rawHash >>> 15;

    return rawHash & mask;
```

Again we are using integers rather than longs. Unfortunately the results are what one would expect.

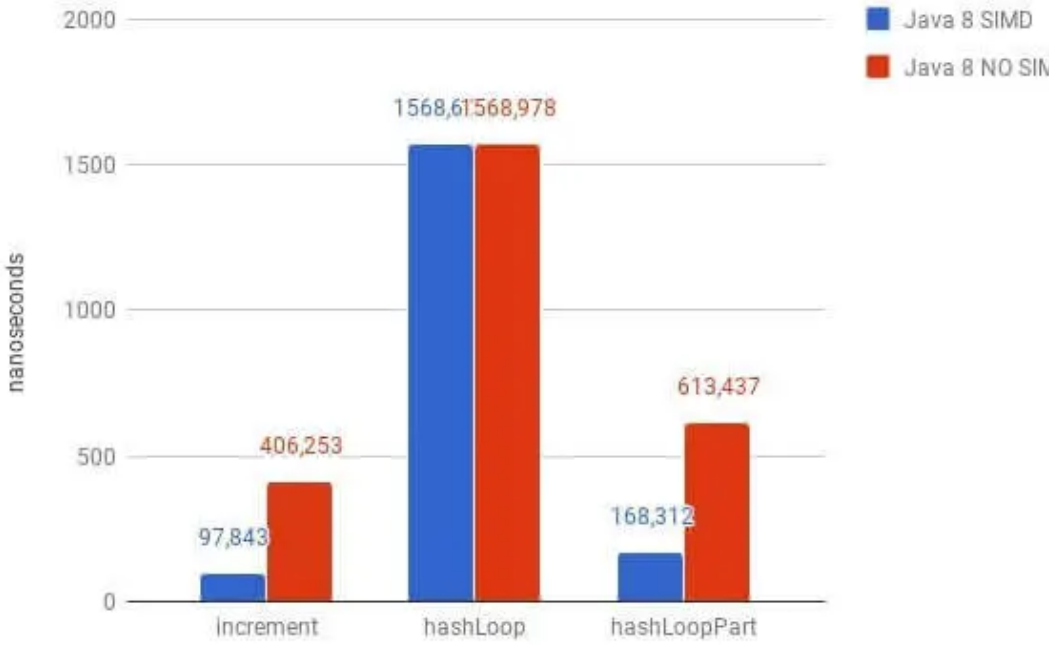


Output produced by JIT tells as that both hashLoop versions look exactly th same, so for some reason Java wasn't able to vectorize this loop. There is r fundamental reason why it shouldn't work. Arithmetic used in hashLoop is m complicated, but it still could be easily translated to a sequence of SIMD operations using only two registers. So what went wrong?

Let's check if the reason why Java did not do the optimization is that the method body is too big. Let's try splitting getHashPosition into smaller functions:

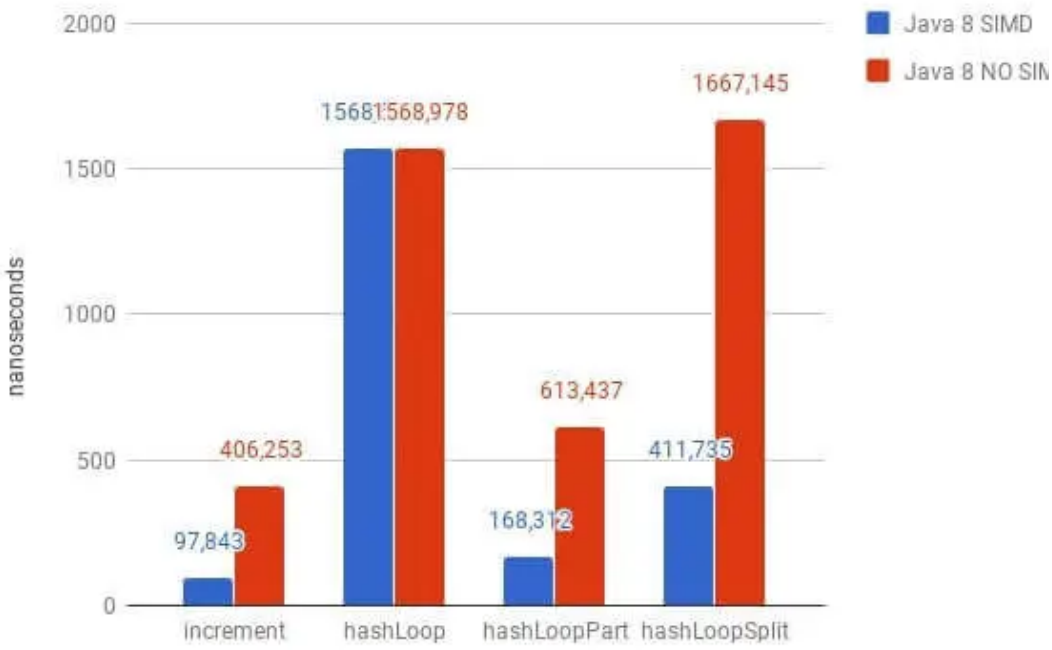
```
@Benchmark
    public void hashLoopPart(Context context)
    {
        for (int i = 0; i < SIZE; i++) {
```

```
            context.results[i] = getHashPosition1(context.values[i]
        }
    }

    private static int getHashPosition1(int rawHash)
    {
        rawHash ^= rawHash >>> 15;
        rawHash *= 0xed558ccd;

    }
```



Simplifying the getHashPosition function by dropping two thirds of its code allowed JIT to vectorize this smaller function. Let's see what happens if we implement getHashPosition as a chain of three smaller loops instead of one larger loop

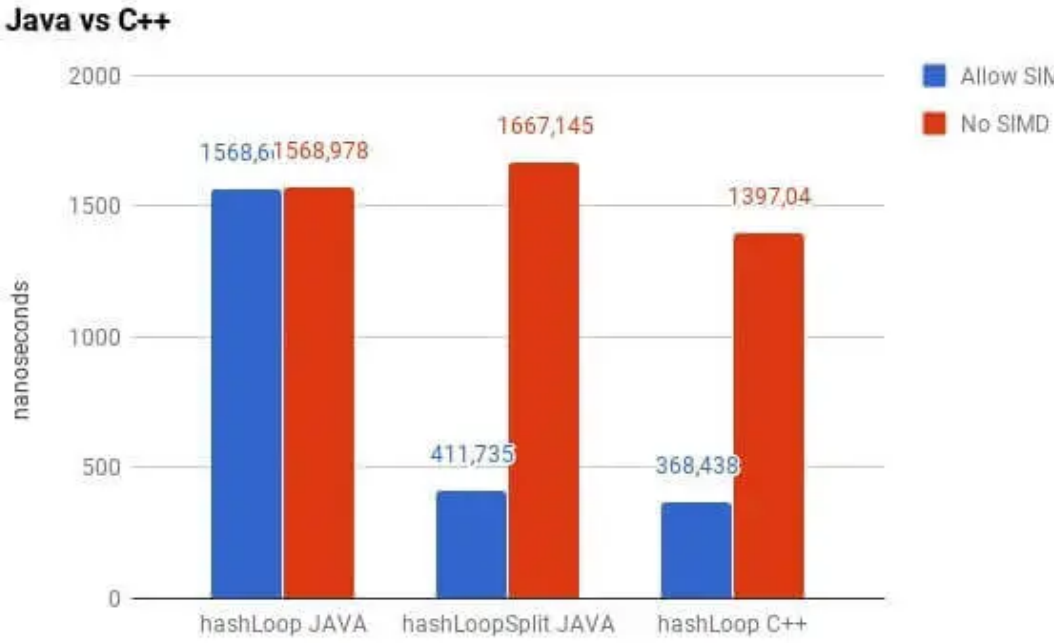
```
@Benchmark
  public int[] hashLoopSplit(Context context)
  {
      for (int i = 0; i < SIZE; i++) {
          context.results[i] = getHashPosition1(context.values[i]
      }

      for (int i = 0; i < SIZE; i++) {
          context.results[i] = getHashPosition2(context.results[i
      }

      for (int i = 0; i < SIZE; i++) {
          context.results[i] = getHashPosition3(context.results[i
      }

      return context.results;
  }

  private static int getHashPosition2(int rawHash)
  {
      rawHash ^= rawHash >>> 15;
      rawHash *= 0x1a85ec53;
      return rawHash;
  }

  private static int getHashPosition3(int rawHash, int mask)
  {
      rawHash ^= rawHash >>> 15;
      return rawHash & mask;
  }
```



Bingo! We have a factor four speed up of the vectorized version over the no vectorized. Sacrificing some performance (~6%) by splitting the loop into thr we convinced the JVM to vectorize each of the smaller loops. This gives us speed up of almost four times over the original hashLoop.

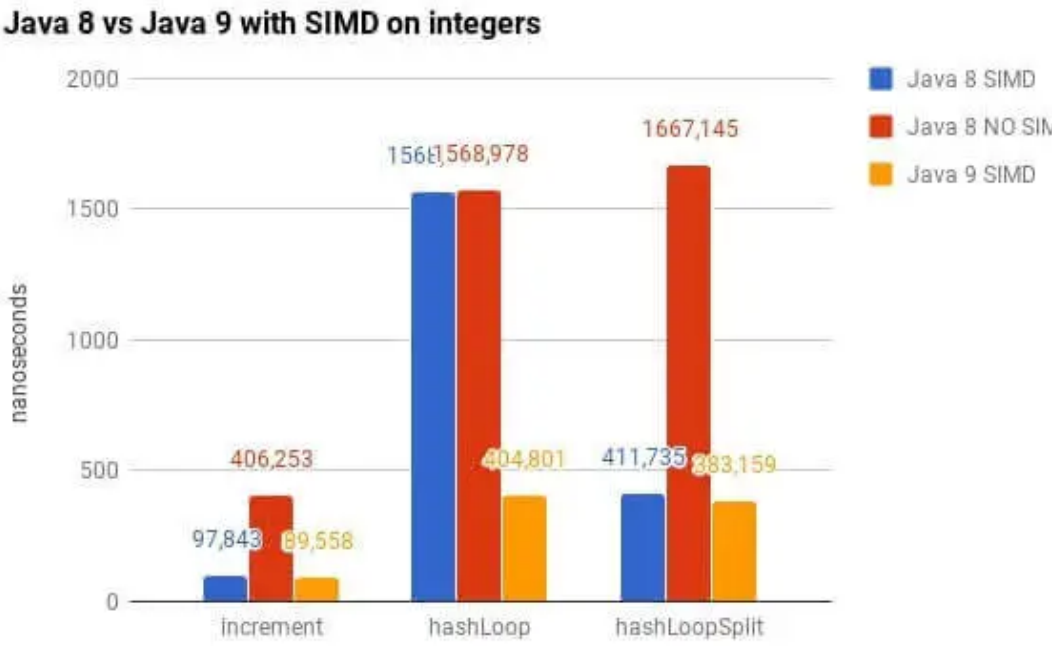
C++

After presenting those results to my colleagues, they argued that maybe the is some other underlying issue with this code that makes it impossible to vectorize. To check this hypothesis I have rewritten the hashLoop benchma into C++ code. For compilation of the C++ code I have used g++ 4.8 with -C ftree-vectorize switches (-ftree-vectorize is turned on by default with -O3).



Java 9

This made me wonder, whether there is some kind of switch that enables m aggressive loop vectorization in the JVM. I have not found anything like this However while browsing through the JVM source code that handles the UseSuperWord switch I have noticed that it has grown and changed a lot between Java 8 version that I have used (Oracle's Java 1.8.0_101) in the above benchmarks and latest master branch. I downloaded OpenJDK's sou code and compiled the latest Java 9 JVM to check if it's more clever. Here a the results:



Nice! With arithmetic done on integers the latest Java version was able to fu vectorize the getHashPosition loop without the need for the hacky splitting c the method body.

Conclusions

First of all, one must be aware when and how SIMD instructions may help improve performance. If the code is bottlenecked on memory access, using SIMD instructions won't help a bit. When arithmetic is a bottleneck of an algorithm, it still might not be possible to use SIMD instructions. Not all algorithms are easy to vectorize, especially if calculations are dependent on one another.

Secondly, even if we have code that could be speeded up by using SIMD instructions, Java doesn't support it very well. We cannot explicitly express t a variable is a vector of values and we cannot manually instruct the compile use SIMD instructions for operations on those vectors, as it is possible in C- or C#. We just have to rely on JIT to be able to vectorize loops. If we have a simple tight loop, that might work, but sometimes it won't. The loop could be complicated. Sometimes there might not be any loop to vectorize. Currently such cases Java programmers are stuck and are not able to unleash the ful potential computational power of modern CPUs. This is a shame, because a clearly visible in the above benchmarks, that using SIMD instructions can speed up code multiple times with only a little bit of effort.