Java auto vectorization example

Asked 12 months ago Active 7 months ago Viewed 1k times

6

2

I'm trying to find a concise example which shows auto vectorization in java on a x86-64 system.

I've implemented the below code using `y[i] = y[i] + x[i]` in a for loop. This code can benefit from auto vectorization, so I think java should compile it at runtime using SSE or AVX instructions to speed it up.
However, I couldn't find the vectorized instructions in the resulting native machine code.

`VecOpMicroBenchmark.java` should benefit from auto vectorization:

```
/**
 * Run with this command to show native assembly:<br/>
 * java -XX:+UnlockDiagnosticVMOptions
 * -XX:CompileCommand=print,VecOpMicroBenchmark.profile VecOpMicroBenchmark
 */
public class VecOpMicroBenchmark {

    private static final int LENGTH = 1024;

    private static long profile(float[] x, float[] y) {
        long t = System.nanoTime();

        for (int i = 0; i < LENGTH; i++) {
            y[i] = y[i] + x[i]; // line 14
        }

        t = System.nanoTime() - t;

        return t;
    }

    public static void main(String[] args) throws Exception {
        float[] x = new float[LENGTH];
        float[] y = new float[LENGTH];

        // to let the JIT compiler do its work, repeatedly invoke
        // the method under test and then do a little nap
        long minDuration = Long.MAX_VALUE;
        for (int i = 0; i < 1000; i++) {
            long duration = profile(x, y);
            minDuration = Math.min(minDuration, duration);
        }
        Thread.sleep(10);

        System.out.println("\n\nduration: " + minDuration + "ns");
    }
}
```

To find out if it gets vectorized, I did the following:

5. In the Arguments tab, under **VM arguments:** put in this: `-XX:+UnlockDiagnosticVMOptions -XX:CompileCommand=print,VecOpMicroBenchmark.profile`
6. **get libhsdis** and copy (possibly rename) the file `hsdis-amd64.so` (.dll for windows) to java/lib directory. In my case, this was `/usr/lib/jvm/java-11-openjdk-amd64/lib` .
7. **run VecOpMicroBenchmark** again

It should now print lots of information to the console, part of it being the disassembled native machine code, which was produced by the JIT compiler. If you see lots of messages, but no assembly instructions like `mov` , `push` , `add` , etc, then maybe you can somewhere find the following message: `Could not load hsdis-amd64.so; library not loadable; PrintAssembly is disabled` This means that java couldn't find the file `hsdis-amd64.so` - it's not in the right directory or it doesn't have the right name.

`hsdis-amd64.so` is the disassembler which is required for showing the resulting native machine code. After the JIT compiler compiles the java bytecode to native machine code, `hsdis-amd64.so` is used to disassemble the native machine code to make it human readable. You can find more infos on how to get/install it at How to see JIT-compiled code in JVM? .

After finding assembly instructions in the output, I skimmed through it (too much to post all of it here) and looked for `line 14` . I found this:

```
0x00007fac90ee9859: nopl    0x0(%rax)
0x00007fac90ee9860: cmp     0xc(%rdx),%esi    ; implicit exception: dispatches
to 0x00007fac90ee997f
0x00007fac90ee9863: jnb     0x7fac90ee9989
0x00007fac90ee9869: movsxd  %esi,%rbx
0x00007fac90ee986c: vmovss  0x10(%rdx,%rbx,4),%xmm0  ;*faload {reexecute=0
rethrow=0 return_oop=0}
                                        ; - VecOpMicroBenchmark::profile@16
(line 14)

0x00007fac90ee9872: cmp     0xc(%rdi),%esi    ; implicit exception: dispatches
to 0x00007fac90ee9997
0x00007fac90ee9875: jnb     0x7fac90ee99a1
```

```
0x00007fac90ee987b: movsxd  %esi,%rbx
0x00007fac90ee987e: vmovss  0x10(%rdi,%rbx,4),%xmm1  ;*faload {reexecute=0
rethrow=0 return_oop=0}
                                                     ; - VecOpMicroBenchmark::profile@20
(line 14)

0x00007fac90ee9884: vaddss  %xmm1,%xmm0,%xmm0
0x00007fac90ee9888: movsxd  %esi,%rbx
0x00007fac90ee988b: vmovss  %xmm0,0x10(%rdx,%rbx,4)  ;*fastore {reexecute=0
rethrow=0 return_oop=0}
                                                     ; - VecOpMicroBenchmark::profile@22
(line 14)
```

So it's using the AVX instruction `vaddss`. But, if I'm correct here, `vaddss` means *add scalar single-precision floating-point values* and this only adds one float value to another one (here, *scalar* means *just one*, whereas here *single* means 32 bit, i.e. `float` and not `double`).
What I expect here is `vaddps`, which means *add packed single-precision floating-point values* and which is a true SIMD instruction (SIMD = single instruction, multiple data = vectorized instruction). Here, *packed* means *multiple floats packed together in one register*.

About the ..ss and ..ps, see http://www.songho.ca/misc/sse/sse.html :

> SSE defines two types of operations; scalar and packed. Scalar operation only operates on the least-significant data element (bit 0~31), and packed operation computes all four elements in parallel. SSE instructions have a suffix -ss for scalar operations (Single Scalar) and -ps for packed operations (Parallel Scalar).

Queston:
Is my java example incorrect, or why is there no SIMD instruction in the output?

java assembly vectorization x86-64 simd

Share Improve this question Follow

asked Jan 13 '20 at 22:51

Daniel S.
5,671 4 28 69

1 Just guessing here, but you might need to ensure that the compiler knows both arrays have a size of `LENGTH`. Apparently each element access checks that the index is within the bounds of the respective array and it throws an exception if not. This may very well disable vectorization. – Jester Jan 13 '20 at 23:00

1 Check this out: stackoverflow.com/a/27666194/363455 – Alex_M Jan 13 '20 at 23:02

1 Auto-vectorization is expensive (in terms of compile time) to find cases where it's possible, and safe and correct. JVMs only JIT-compile, vs. ahead-of-time compilers having lots of time to search for optimizations. Don't be surprised when JIT compilers fail to vectorize, or when they make clunky bad scalar asm (e.g. Why is 2 * (i * i) faster than 2 * i * i in Java?) that fails to even use obvious peephole optimizations like `lea` for shift-and-add into a new destination. – Peter Cordes Jan 14 '20 at 1:44

1 Answer

Active | Oldest | Votes

3

In the `main()` method, put in `i < 1000000` instead of just `i < 1000`. Then the JIT also produces AVX vector instructions like below, and the code runs faster:

```
0x00007f20c83da588: vmovdqu 0x10(%rbx,%r11,4),%ymm0
0x00007f20c83da58f: vaddps  0x10(%r13,%r11,4),%ymm0,%ymm0
0x00007f20c83da596: vmovdqu %ymm0,0x10(%rbx,%r11,4)  ;*fastore {reexecute=0
rethrow=0 return_oop=0}
                                                     ; - VecOpMicroBenchmark::profile@22
(line 14)
```

The code from the question is actually optimizable by the JIT compiler using auto-vectorization. However, as Peter Cordes pointed out in a comment, the JIT needs quite some processing, thus it is rather reluctant to decide that it should fully optimize some code.
The solution is simply to execute the code more often during one execution of the program, not just 1000 times, but 100000 times or a million times.
When executing the `profile()` method this many times, the JIT compiler is convinced that the code is very important and the overall runtime will benefit from full optimization, thus it optimizes the code again and then it also uses true vector instructions like `vaddps`.

More details in Auto Vectorization in Java

Share Improve this answer Follow

answered Jun 8 '20 at 12:41

Daniel S.
5,671 4 28 69