Auto Vectorization in Java

published: 2020-01-16
started: 2020-01-14
last change: 2020-01-26

Abstract

This article briefly explains the prupose of vectorization, how it currently works in Java, and how to check if it's applied in a Java program. This knowledge can be turned into top-notch performance optimizations for arithmetic algorithms.
These techniques are low-level and suitable for special cases only. If you have a standard Java program that you want to performance-optimize, then you should first use other optimization techniques available. Only if you already optimized your Java code using other techniques, and you profiled it thereafter and you concluded that a part focusing on arithmetic calculations might run even faster with parallelization, then this article might be useful to you.

Contents

1. Introduction

1.1. SIMD

Typically, a program's code is executed serially. That means individual commands or statements are executed in sequence, one after another. Arithmetically focused programs are programs which do lots of calculations on numbers. Usually, these programs process big amounts of data, and many pieces of information get processed in the same way, one after another. E.g. for a simulation of 1000 particles there will be a step in the simulation which updates the location s of each particly with its current velocity v: `s = s + v`. This has to be done for every particle, i.e. for an array of particles `s[0]` to `s[999]`.
If you can combine a few particles to a batch and process the batch in one go, this speeds things up. For the above example and a group of 4 particles, this means to group the calculations like follows:

```
s[0] = s[0] + v[0];
s[1] = s[1] + v[1];
s[2] = s[2] + v[2];
s[3] = s[3] + v[3];
```

More generally speaking, this does one kind of operation on multiple data elements at once. On the level of assembly code there are instructions specifically for these grouped operations. Therefor this concept is called single instruction, multiple data, abbreviated SIMD.

The SIMD instruction for + is called **addps** (SSE instructon set) or **vaddps** (AVX instruction set) on x86 CPUs. It takes two groups as operands where each group has either 4 elements (SSE) or 8 elements (AVX). It adds each element of one group to the corresponding element of the other group. In the above example, `s[0..3]` is one group and `v[0..3]` is the other group. The resulting x86 assembly code is:

```
addps  %xmm0,%xmm1   ;add vector in xmm0 to vector in xmm1, store result in xmm0
```

1.2. Vectorization

SIMD is the name of the concept given from the perspective of instruction designers, namely the CPU manufacturers. But that's not the only perspective. In mathematics, ordered groups of a fixed number of elements (`s[0..3]` and `v[0..3]`) are called vectors. Therefor SIMD instructions are also called vector instructions. This is just another perspective on the same thing, this time from the user of the instructions.

Vectorization is the usage of vector instructions to speed up program execution. Vectorization can be done by a programmer or the possibilities for vectorization can be automatically realized by a compiler. In the latter case it's called auto vectorization.

> **Auto vectorization is a kind of code optimization which is done by a compiler, either by an AOT compiler at compile time, or by a JIT compiler at execution time.**

1.3. Vector Instructions in Java

After writing a Java program, the Java source code in Java-files gets compiled to bytecode and saved to class-files. And then, before or during the execution of the program, its bytecode is usually compiled again, this time from bytecode to native machine code. This latter compilation is usually done while the program is executed, hence it's a JIT compilation.

In Java, currently vectorization is not done by the programmer[1], but it's done automatically by the compiler. The compiler takes in standard Java bytecode and automatically determines which part can be transformed to vector instructions. Common Java environments like OpenJDK or Oracle's Java can produce vectorized machine code.

2. Code That Can Benefit From Vectorization

Code can be transformed to vectorized instructions if it executes the same operation on many consecutive elements of an array. Example:

```
float[] a = ...

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

Snippet 1: Vectorizable Code

In Snippet 1, the statement `a[i] = a[i] * a[i];` gets executed on many consecutive elements of the array `a`. The compiler can check this and instead of doing every `*` individually, it can use a vector instruction to calculate multiple results at once.

3. Check If Vectorization Is Applied

3.1. Prepare a Mirco Benchmark

To see the generated vector instructions as assembly code, we first have to create a compilable and runnable Java program which can benefit from vector instructions. For this, take the above for loop and put it into a Java file, into the `square(...)` method, along with a `main(...)` method. Write code so that `square(...)` is executed a million times, or at least a few hundreds of thousands of times. This convinces the compiler that `square(...)` is a method worth optimizing to the fullest. `square(...)` is then said to be *"running hot"* or to contain a *"hot loop"*. This running hot is achieved by the for loop in `main(...)`. So we have two loops, one in `main(...)` and one in `square(...)`. The hot loop is the one in `square(...)`.

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

    private static void square(float[] a) {
        for (int i = 0; i < a.length; i++) {
            a[i] = a[i] * a[i]; // line 11
        }
    }

    public static void main(String[] args) throws Exception {
        float[] a = new float[1024];

        // repeatedly invoke the method under test. this
        // causes the JIT compiler to optimize the method
        for (int i = 0; i < 1000 * 1000; i++) {
            square(a);
        }
    }
}
```

Snippet 2: `VectorizationMicroBenchmark.java`

3.2. Run the Mirco Benchmark

1. **Open Eclipse** and create a new project. Create a new class in the new project and name it `VectorizationMicroBenchmark`. Copy-paste the code of Snippet 2 into it.
2. Right-click the file and from the dropdown menu, choose **Run > Java Application** (ignore the output for now)
3. In Eclipse's menu, click **Run > Run Configurations...**
4. A window opened. In the window, **find VectorizationMicroBenchmark**, click it and choose the

Arguments tab

5. in the Arguments tab, under **VM arguments:** put in this:
   `-XX:+UnlockDiagnosticVMOptions -XX:CompileCommand=print,VectorizationMicroBenchmark.square`
6. **get libhsdis** and copy (possibly rename) the file `hsdis-amd64.so` (.dll for windows) to your Java-home/lib directory. On Ubuntu, this is something like `/usr/lib/jvm/Java-11-openjdk-amd64/lib`.
7. **run VectorizationMicroBenchmark** again

If you don't have Eclipse, you can do these steps analogously in any other IDE or with a text editor and `javac/java` on the command line.

Step 7. prints lots of information to the console, part of it being the disassembled native machine code. If you see lots of messages but no assembly instructions like `mov`, `push`, `add`, etc, then maybe you can find the following message somewhere in the output:
`Could not load hsdis-amd64.so; library not loadable; PrintAssembly is disabled`
If you see this message, it 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. On Linux, this also happens when you created a symlink. Java doesn't accept symlinks here. Instead, you have to copy the file.

`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 on How to see JIT-compiled code in JVM.

3.3. Output Interpretation

After finding assembly instructions in the output, you might be surprised that you do not only find the assembly code of the `square(...)` method, but instead you find several versions of it. This is because the JIT compiler does not optimize the method fully on the first run. After some invocations of the method, it compiles it to native code without optimizations. After more invocations, it compiles the method again with some optimizations, but not all. And only after several thousand invocations, the compiler is convinced that the method is so impotant that it needs to be compiled with all optimizations switched on, including vectorization. So the best compilation usually is the last one in the output.

Start searching for "`line 11`" at the end of the output, going backwards. You might find something like this:

```
0x...ac70: vmovss  0x10(%rbx,%rbp,4),%xmm0  ;*faload {reexecute=0 rethrow=0 return_oo
                                            ; - VectorizationMicroBenchmark::square@9 (line

0x...ac76: vmulss  %xmm0,%xmm0,%xmm1
0x...ac7a: vmovss  %xmm1,0x10(%rbx,%rbp,4)  ;*fastore {reexecute=0 rethrow=0 return_o
                                            ; - VectorizationMicroBenchmark::square@14 (line
```

Snippet 3: Not Vectorized Assembly Code

Note the instruction **`vmulss`** with the **`-ss`** at the end in Snippet 3.

`vmulss`: *multiply **s**calar **s**ingle-precision floating-point values*

`vmulss` multiplies only one float with another one. So this is not what we want. (Here, *scalar* means *just one* and *single-precision* means 32 bit, i.e. `float` and not `double`). We instead want to find an instruction which multiplies many floats with many other floats in one go. So keep looking on. You will

eventually find this:

```
0x...ac54: vmovdqu 0x10(%rbx,%rbp,4),%ymm0  ;*faload {reexecute=0 rethrow=0 return_oo
                                            ; - VectorizationMicroBenchmark::square@9 (line

0x...ac5a: vmulps  %ymm0,%ymm0,%ymm0
0x...ac5e: vmovdqu %ymm0,0x10(%rbx,%rbp,4)  ;*fastore {reexecute=0 rethrow=0 return_o
                                            ; - VectorizationMicroBenchmark::square@14 (line
```

Snippet 4: Vectorized Assembly Code

In Snippet 4, there is **`vmulps`** with the **`-ps`** ending.

`vmulps`: *multiply **p**acked **s**ingle-precision floating-point values*

`vmulps` is a true SIMD instruction (reminder: SIMD = single instruction, multiple data = vectorized instruction). Here, *packed* means *multiple elements packed together in one register*. This shows that auto vectorization was applied.

4. Footnotes

[1] Extensions for Java are planned which will allow programmers to explicitly use vector instructions in the source code. These extensions are not ready at the time of writing (Spring 2020). See JEP-338 and Project Panama: Interconnecting JVM and native code for details and status. (back)

5. References

http://www.songho.ca/misc/sse/sse.html - illustration of the results of -ps and -ss instructions:

https://www.felixcloutier.com/x86/ - x86 and amd64 instruction reference

http://jpbempel.blogspot.com/2015/12/printassembly-output-explained.html - PrintAssembly output explained