10d. SIMD: Independence of Iterations

Martin Alfaro
PhD in Economics

INTRODUCTION

Broadcasting heavily favors the application of SIMD instructions. In contrast, whether and when for-loops apply SIMD is more complex. Furthermore, the heuristics of the compiler, while powerful, aren't without flaws. Indeed, it's entirely possible that SIMD is implemented when it actually reduces performance or not applied when it would've been advantageous. To address this, Julia provides the `@simd` macro to manually implement SIMD, giving developers a more granular control over the optimization process.

An effective application of SIMD requires identifying the conditions under which this optimization can be applied. Failing to meet these criteria can render SIMD infeasible or necessitate code adaptations that end up slowing down computation. The ideal conditions for leveraging SIMD instructions are:

- **Independence of Iterations**: Except for reductions, which are specifically handled to ensure their feasibility.
- **Unit Strides**: Elements in collections must be accessed sequentially.
- **No Conditional Statements**: The loop body should consist solely of straight-line code.

In the upcoming sections, we'll elaborate on each of these items, additionally providing guidance on how to address scenarios not conforming to them. This section in particular exclusively focuses on the independence of iterations.

> **Warning!** - Determining Whether SIMD Has Been Implemented
>
> Assessing whether SIMD instructions are implemented requires inspecting the compiled code. Due to the complexity of this approach, we'll instead rely on execution times as a practical indicator.

REMARKS ABOUT @SIMD IN FOR-LOOPS

Recall from the previous section that the impact of macros on computational methods is intricate. The reason is that macros only serve as a hint to the compiler, rather than a strict directive. Consequently, they suggest techniques that the compiler may eventually discard or would have implemented regardless—the compiler has the final say on which optimizations are worth adopting. In this context, the inclusion of `@simd` in a for-loop is far from a guarantee that SIMD will actually be implemented.

Furthermore, it's notoriously difficult to predict whether SIMD instructions are beneficial in particular scenarios. This is due to several factors. Firstly, different CPU architectures provide varying levels of support for SIMD instructions. [1] This diversity in SIMD capabilities means that the benefits of SIMD tend to vary greatly by hardware.

Second, as we already mentioned, it's hard to anticipate when and how SIMD will be applied in our code. The compiler relies on sophisticated heuristics to determine when SIMD may be advantageous, but they aren't infallible. Indeed, it's entirely possible that SIMD is implemented when it actually reduces performance or not applied when it would've been advantageous.

Despite these complexities, structuring operations in certain ways can improve the likelihood of implementing SIMD beneficially. By identifying these conditions, we'll be able to write code that's more amenable to SIMD optimization. It's worth remarking, though, that **the recommendations we'll present should be interpreted as general principles, rather than absolute rules**. Given the complexity of SIMD, benchmarking remains necessary to validate the existence of any performance improvement.

> **Safety of SIMD**
>
> Strictly speaking, SIMD is a form of parallelization. We'll see in subsequent sections that parallelization may render code unsafe and lead to catastrophic errors when used improperly. `@simd` doesn't involve these types of risks, since it's been designed to apply only when it's safe to do so. Specifically, the compiler will disregard SIMD if the conditions for its safe application aren't met.

INDEPENDENCE OF ITERATIONS

To effectively apply SIMD, iterations should be independent. This means that no iteration should depend on the results of previous iterations or affect the results of subsequent ones. When this condition is met, each iteration can be executed in parallel. A typical scenario is when we need to apply some function $f\left(x_i\right)$ to each element x_i of a vector `x`.

In the following, we illustrate this case via a polynomial transformation of `x`. The transformation will be done through for-loops with and without SIMD. We'll also compare these approaches with broadcasting, which applies SIMD automatically.

Importantly, as we'll explain in a subsequent section, applying `@simd` in for-loops requires the `@inbounds` macro. We'll see that, essentially, checking index bounds introduces a condition, giving rise to execution branches that hinder or directly prevent the application of SIMD.

```
x = rand(1_000_000)

function foo(x)
    output = similar(x)

    for i in eachindex(x)
        output[i] = x[i] / 2 + x[i]^2 / 3
    end

    return output
end
```

```
julia> @btime foo($x)
  806.606 μs (2 allocations: 7.629 MiB)
```

```
x = rand(1_000_000)

function foo(x)
    output = similar(x)

    @inbounds @simd for i in eachindex(x)
        output[i] = x[i] / 2 + x[i]^2 / 3
    end

    return output
end
```

```
julia> @btime foo($x)
  464.734 μs (2 allocations: 7.629 MiB)
```

```
x = rand(1_000_000)

foo(x) = @. x / 2 + x^2 / 3
```

```
julia> @btime foo($x)
  447.074 μs (4 allocations: 7.629 MiB)
```

A SPECIAL CASE OF DEPENDENCE: REDUCTIONS

SIMD requires that iterations are independent. One exception to this rule is given by reductions, which have been carefully designed for their proper handling.

Julia leverages SIMD automatically for reductions involving integers. Instead, reductions with floating-point numbers require the explicit addition of the `@simd` macro. The following example demonstrates this fact. For the case of integers, we see that there are no differences in execution times with and without `@simd`.

```
x = rand(1:10, 10_000_000)   # random integers between 1 and 10

function foo(x)
    output = 0

    for a in x
        output += a
    end

    return output
end
```

```
julia> @btime foo($x)
  2.606 ms (0 allocations: 0 bytes)
```

```
x = rand(1:10, 10_000_000)   # random integers between 1 and 10

function foo(x)
    output = 0

    @simd for a in x
        output += a
    end

    return output
end
```

```
julia> @btime foo($x)
  2.636 ms (0 allocations: 0 bytes)
```

This behavior contrasts with a sum reduction consisting of floating-point operations, as shown below.

```
x = rand(10_000_000)

function foo(x)
    output = 0.0

    for a in x
        output += a
    end

    return output
end
```

```
julia> @btime foo($x)
  5.033 ms (0 allocations: 0 bytes)
```

```
x = rand(10_000_000)

function foo(x)
    output = 0.0

    @simd for a in x
        output += a
    end

    return output
end
```

```
julia> @btime foo($x)
  2.753 ms (0 allocations: 0 bytes)
```

Why Floating Points Are Treated Differently

Unlike integers, addition of floating-point numbers doesn't obey associativity: due to the inherent imprecision of floating-point arithmetic, `(x+y)+z` may not be exactly equal to `x+(y+z)`. This is one of several reasons why floating-point numbers are distinct from mathematical real numbers: they are finite-precision approximations that don't always follow the same mathematical properties we expect from real numbers.

The following code shows this feature of floating points.

```
x = 0.1 + (0.2 + 0.3)
```

```
julia> x
0.6
```

```
x = (0.1 + 0.2) + 0.3
```

```
julia> x
0.6000000000000001
```

By instructing the compiler to ignore the non-associativity of floating-point arithmetic, SIMD instructions can optimize performance by reordering terms. However, this approach assumes that the operations do not rely on a specific order of operations. Fortunately, this assumption rarely causes issues in scientific applications, as they typically involve mathematical models that inherently assume real number properties from the outset.

FOOTNOTES

[1.] For instance, x86 architectures (Intel and AMD processors) offer SSE (Streaming SIMD Extensions) and AVX (Advanced Vector Extensions). In turn, each comprises variants supporting different vector widths and operations (e.g., the variant AVX-512 in Intel Xeon processors).