10f. SIMD: Branchless Code

Martin Alfaro
PhD in Economics

INTRODUCTION

SIMD accelerates computations by executing the same set of instructions in parallel across multiple data elements. Yet, certain programming constructs, particularly conditional statements, can severely degrade SIMD efficiency. The issue occurs because conditional statements inherently lead to different instruction paths, thus disrupting the single instruction execution that SIMD relies on. While the compiler attempts to mitigate this issue by transforming code into SIMD-compatible forms, these adaptations often result in a performance penalty.

This section explores strategies for efficiently applying SIMD in the presence of conditional operations. We'll first examine scenarios in which the compiler introduces conditional statements as a byproduct of its internal computation techniques. By employing alternative coding strategies, we'll show how these conditional statements can be bypassed.

After this, we'll explore conditional statements that are intrinsic to program logic and therefore unavoidable. By this, we mean typical scenarios where conditions are explicitly introduced, ensuring that certain operations are only executed under specific circumstances. Here, we'll revisit the usual approaches to expressing conditions, focusing on their internal implementation. The goal is to outline the relative strengths and limitations of each approach, indicating which ones are more conducive to SIMD optimizations. Finally, we'll show that conditional statements can be equivalently recast as algebraic operations, effectively removing the branching logic that disrupts SIMD execution. In fact, the compiler may implement this strategy under the hood when it's likely beneficial.

TYPE INSTABILITY AND BOUNDS CHECKING AS CONDITIONS TO AVOID

Two patterns in Julia introduce internal branching: type-unstable functions and bounds checking during array indexing. This conditional operation arises from compiler decisions rather than explicit code, making them unnoticeable.

When a function is type-unstable, Julia generates multiple execution branches, one for each type. Those extra branches, while hidden from the user, still disrupt the uniform instruction flow required by SIMD. The remedies for this case are the same as those for fixing type instabilities, for which we devoted the whole chapter 8. Regardless of any SIMD consideration, it's worth remarking that type stability should always be a priority: any attempt to achieve high performance is nearly impossible without guaranteeing it.

A second source of hidden branching arises in for-loops, where Julia performs bounds checking by default. This operation represents a subtle form of conditional execution, where each iteration is executed only when indices remain within bounds. These checks interfere with the application of SIMD, since they prevent the compiler from treating the for-loop as a uniform sequence of operations.

The example below demonstrates two key insights regarding bounds checking. First, **adding `@inbounds` is a prerequisite for the application of SIMD**. Specifically, if bound checks remain in place, simply using `@simd` alone will rarely trigger the implementation of SIMD instructions. Second, merely adding `@inbounds` can be enough to induce the compiler to apply SIMD instructions, rendering `@simd` annotations redundant for performance improvements. This additionally explains why using `@inbounds @simd` may not speed up execution times. [1]

```
x = rand(1_000_000)

function foo(x)
    output = similar(x)

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

    return output
end
```

```
julia> @btime foo($x)
  824.276 µs (3 allocations: 7.629 MiB)
```

```
x = rand(1_000_000)

function foo(x)
    output = similar(x)

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

    return output
end
```

```
julia> @btime foo($x)
  1.109 ms (3 allocations: 7.629 MiB)
```

```
x = rand(1_000_000)

function foo(x)
    output = similar(x)

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

    return output
end
```

```
julia> @btime foo($x)
  390.190 μs (3 allocations: 7.629 MiB)
```

```
x = rand(1_000_000)

function foo(x)
    output = similar(x)

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

    return output
end
```

```
julia> @btime foo($x)
  393.380 μs (3 allocations: 7.629 MiB)
```

Overall, the main takeaway from the example is that when the goal is to exploit SIMD in a for-loop, the for-loop should be preceded by `@inbounds @simd`.

> **Broadcasting and For-Loops**
>
> Broadcasting in Julia automatically disables bounds checking and encourages the use of SIMD instructions. This often makes broadcasted expressions appear faster than a straightforward for-loop. However, broadcasting isn't fundamentally different regarding its mechanics. Essentially, broadcasting is a compact notation for implementing certain for-loops. For instance, a for-loop with `@inbounds` and `@simd` usually exhibits a similar performance to a broadcast variant. This is demonstrated below.
>
> ```
> x      = rand(1_000_000)
> foo(x) = 2 ./ x
> ```
>
> ```
> julia> @btime foo($x)
>   452.692 μs (3 allocations: 7.629 MiB)
> ```

```
x      = rand(1_000_000)

function foo(x)
    output = similar(x)

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

    return output
end
```

```
julia> @btime foo($x)
  802.428 µs (3 allocations: 7.629 MiB)
```

```
x      = rand(1_000_000)

function foo(x)
    output = similar(x)

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

    return output
end
```

```
julia> @btime foo($x)
  399.047 µs (3 allocations: 7.629 MiB)
```

APPROACHES TO CONDITIONAL STATEMENTS

When conditions are integral to a program's logical flow and can't be avoided, it becomes important to consider the most effective way to introduce them. The options in this regard must consider that conditional statements can be evaluated either eagerly or lazily.

To illustrate, let's consider the computation of `1 + 1` but only if certain condition `C` is met. A lazy approach evaluates whether `C` holds true, before proceeding with the computation of `1 + 1`. Thus, the operation is deferred until it's confirmed that `C` holds. By contrast, an eager approach performs the computation immediately, regardless of whether `C` is satisfied. If the condition is eventually `false`, the result of the computation is then discarded.

When conditional statements are applied only once, a lazy approach is more performant as it avoids needless computations. However, when a condition is embedded inside a for-loop, there are multiple operations to be potentially computed. Since SIMD can compute more than one operation

simultaneously, it may be beneficial to evaluate all conditions and branches upfront, selecting the relevant branches afterward. The possibility is especially true when branches involve inexpensive computations.

In Julia, whether a conditional statement is evaluated lazily or eagerly depends on how it's written. Next, we explore this aspect in more detail.

IFELSE VS IF

The `ifelse` function in Julia follows an eager evaluation strategy, where both the condition and possible outcomes are computed before deciding which result to return. In contrast, `if` favors lazy computations, only evaluating the necessary components based on the truth value of the condition.

The following example demonstrates this computational difference between `if` and `ifelse`. It relies on a reduction operation, whose elements to be summed are contingent on a condition. Note that `ifelse` requires specifying an operation for both the true and false cases. For a sum reduction, the false case is handled by returning zero when the condition isn't met.

```
x = rand(1_000_000)

function foo(x)
    output = 0.0

    for i in eachindex(x)
        if x[i] > 0.5
            output += x[i]/2
        end
    end

    return output
end
```

```
julia> @btime foo($x)
  409.692 μs (0 allocations: 0 bytes)
```

```
x = rand(1_000_000)

function foo(x)
    output = 0.0

    @inbounds @simd for i in eachindex(x)
        if x[i] > 0.5
            output += x[i]/2
        end
    end

    return output
end
```

```
julia> @btime foo($x)
  406.976 μs (0 allocations: 0 bytes)
```

```
x = rand(1_000_000)

function foo(x)
    output = 0.0

    for i in eachindex(x)
        output += ifelse(x[i] > 0.5, x[i]/2, 0)
    end

    return output
end
```

```
julia> @btime foo($x)
  385.429 μs (0 allocations: 0 bytes)
```

```
x = rand(1_000_000)

function foo(x)
    output = 0.0

    @inbounds @simd for i in eachindex(x)
        output += ifelse(x[i] > 0.5, x[i]/2, 0)
    end

    return output
end
```

```
julia> @btime foo($x)
  89.666 μs (0 allocations: 0 bytes)
```

As the example reveals, the fact that `ifelse` is eager doesn't automatically trigger the application of SIMD. This is precisely why `@inbounds @simd` had to be included.

It's also worth remarking that applying SIMD instructions doesn't necessarily increase performance. The example below demonstrates this point.

```
x      = rand(5_000_000)
output = similar(x)

function foo!(output,x)
    for i in eachindex(x)
        output[i] = ifelse(x[i] > 0.5, x[i]/2, 0)
    end
end
```

```
julia> foo!($output,$x)
  18.720 ms (0 allocations: 0 bytes)
```

```
x      = rand(5_000_000)
output = similar(x)

function foo!(output,x)
    @inbounds @simd for i in eachindex(x)
        output[i] = ifelse(x[i] > 0.5, x[i]/2, 0)
    end
end
```

```
julia> foo!($output,$x)
  16.518 ms (0 allocations: 0 bytes)
```

TERNARY OPERATORS

Ternary operators are an alternative approach for conditional statements. They have the form `<condition> ? <action if true> : <action if false>`. Unlike the previous methods, this form isn't committed to an eager or a lazy approach. Instead, it relies on heuristics to determine which approach should be implemented. The decision depends on the compiler's assessment about which strategy will likely be faster in the application considered.

For the illustrations, we'll consider examples where `@inbounds` and `@simd` are directly added in each approach. Based on the same example as above, the ternary operator could opt for an eager approach.

```
x = rand(1_000_000)

function foo(x)
    output = 0.0

    @inbounds @simd for i in eachindex(x)
        if x[i] > 0.5
            output += x[i]/2
        end
    end

    return output
end
```

```
julia> foo!($output,$x)
  407.000 μs (0 allocations: 0 bytes)
```

```
x = rand(1_000_000)

function foo(x)
    output = 0.0

    @inbounds @simd for i in eachindex(x)
        output += ifelse(x[i]>0.5, x[i]/2, 0)
    end

    return output
end
```

```
julia> foo!($output,$x)
  89.095 μs (0 allocations: 0 bytes)
```

```
x = rand(1_000_000)

function foo(x)
    output = 0.0

    @inbounds @simd for i in eachindex(x)
        output += x[i]>0.5 ? x[i]/2 : 0
    end

    return output
end
```

```
julia> foo!($output,$x)
  88.226 μs (0 allocations: 0 bytes)
```

Instead, the ternary operator opts for a lazy approach in the following example.

```
x = rand(1_000_000)

function foo(x)
    output = 0.0

    @inbounds @simd for i in eachindex(x)
        if x[i] > 0.99
            output += log(x[i])
        end
    end

    return output
end
```

```
julia> foo!($output,$x)
  393.501 μs (0 allocations: 0 bytes)
```

```
x = rand(1_000_000)

function foo(x)
    output = 0.0

    @inbounds @simd for i in eachindex(x)
        output += ifelse(x[i] > 0.99, log(x[i]), 0)
    end

    return output
end
```

```
julia> foo!($output,$x)
  3.609 ms (0 allocations: 0 bytes)
```

```
x = rand(1_000_000)

function foo(x)
    output = 0.0

    @inbounds @simd for i in eachindex(x)
        output += x[i]>0.99 ? log(x[i]) : 0
    end

    return output
end
```

```
julia> foo!($output,$x)
  413.706 μs (0 allocations: 0 bytes)
```

TERNARY OPERATOR COULD CHOOSE A LESS PERFORMANT APPROACH

It's worth remarking that the method chosen by the ternary operator isn't foolproof. In the following example, the ternary operator actually implements the slower approach.

```
x      = rand(5_000_000)
output = similar(x)

function foo!(output,x)
    @inbounds @simd for i in eachindex(x)
        if x[i] > 0.5
            output[i] = log(x[i])
        end
    end
end
```

```
julia> foo!($output,$x)
  26.299 ms (0 allocations: 0 bytes)
```

```
x      = rand(5_000_000)
output = similar(x)

function foo!(output,x)
    @inbounds @simd for i in eachindex(x)
        output[i] = ifelse(x[i] > 0.5, log(x[i]), 0)
    end
end
```

```
julia> foo!($output,$x)
  17.272 ms (0 allocations: 0 bytes)
```

```
x      = rand(5_000_000)
output = similar(x)

function foo!(output,x)
    @inbounds @simd for i in eachindex(x)
        output[i] = x[i]>0.5 ? log(x[i]) : 0
    end
end
```

```
julia> foo!($output,$x)
  25.943 ms (0 allocations: 0 bytes)
```

WHEN EACH APPROACH IS BETTER?

As a rule of thumb, conditional statements with **computational-demanding operations will more likely benefit from a lazy implementation**. In contrast, **an eager approach is potentially more performant when branches comprise simple algebraic computations**. In fact, these heuristics tend to drive the decision adopted by the ternary operator.

To demonstrate, the following example considers a conditional statement where there's only one branch with a computation and this is straightforward. An eager approach with SIMD is faster, and coincides with the approach chosen when a ternary operator is chosen.

```
x             = rand(1_000_000)
condition(a)   = a > 0.5
computation(a) = a * 2

function foo(x)
    output = 0.0

    @inbounds @simd for i in eachindex(x)
        if condition(x[i])
            output += computation(x[i])
        end
    end

    return output
end
```

```
julia> foo!($output,$x)
  399.915 μs (0 allocations: 0 bytes)
```

```
x             = rand(1_000_000)
condition(a)   = a > 0.5
computation(a) = a * 2

function foo(x)
    output = 0.0

    @inbounds @simd for i in eachindex(x)
        output += ifelse(condition(x[i]), computation(x[i]), 0)
    end

    return output
end
```

```
julia> foo!($output,$x)
  88.727 μs (0 allocations: 0 bytes)
```

```
x             = rand(1_000_000)
condition(a)   = a > 0.5
computation(a) = a * 2

function foo(x)
    output = 0.0

    @inbounds @simd for i in eachindex(x)
        output += condition(x[i]) ? computation(x[i]) : 0
    end

    return output
end
```

```
julia> foo!($output,$x)
  89.035 μs (0 allocations: 0 bytes)
```

Instead, below we consider a branch with computational-intensive calculations. In this case, a lazy approach is faster, which is the approach implemented by the ternary operator.

```
x              = rand(2_000_000)
condition(a)   = a > 0.5
computation(a) = exp(a)/3 - log(a)/2

function foo(x)
    output = 0.0

    @inbounds @simd for i in eachindex(x)
        if condition(x[i])
            output += computation(x[i])
        end
    end

    return output
end
```

```
julia> foo!($output,$x)
  12.655 ms (0 allocations: 0 bytes)
```

```
x              = rand(2_000_000)
condition(a)   = a > 0.5
computation(a) = exp(a)/3 - log(a)/2

function foo(x)
    output = 0.0

    @inbounds @simd for i in eachindex(x)
        output += ifelse(condition(x[i]), computation(x[i]), 0)
    end

    return output
end
```

```
julia> foo!($output,$x)
  12.023 ms (0 allocations: 0 bytes)
```

```
x             = rand(2_000_000)
condition(a)  = a > 0.5
computation(a) = exp(a)/3 - log(a)/2

function foo(x)
    output = 0.0

    @inbounds @simd for i in eachindex(x)
        output += condition(x[i]) ? computation(x[i]) : 0
    end

    return output
end
```

```
julia> foo!($output,$x)
  13.124 ms (0 allocations: 0 bytes)
```

VECTORS WITH CONDITIONS

Next, we consider scenarios where you already have defined a vector holding conditions. This could occur either because the vector is already part of your dataset, or because the conditions will be reused multiple times over your code, in which case previously storing the conditions is worthy.

Storing conditions in a vector could be done through an object with type `Vector{Bool}` or `BitVector`. The latter is the default type returned by Julia, as when you define objects like `x .> 0`. Although this type offers certain performance advantages, it can also hinder the application of SIMD. In cases like this, transforming `BitVector` to `Vector{Bool}` could speed up computations. The following example demonstrates this, where the execution time is faster even accounting for the type conversion.

```
x         = rand(1_000_000)
bitvector = x .> 0.5

function foo(x,bitvector)
    output     = similar(x)


    @inbounds @simd for i in eachindex(x)
        output[i] = ifelse(bitvector[i], x[i]/i, x[i]*i)
    end

    return output
end
```

```
julia> foo($x,$bitvector)
  3.098 ms (3 allocations: 7.629 MiB)
```

```
x         = rand(1_000_000)
bitvector = x .> 0.5

function foo(x,bitvector)
    output     = similar(x)
    boolvector = Vector(bitvector)

    @inbounds @simd for i in eachindex(x)
        output[i] = ifelse(boolvector[i], x[i]/i, x[i]*i)
    end

    return output
end
```

```
julia> foo($x,$bitvector)
  848.068 µs (5 allocations: 8.583 MiB)
```

No Vector With Conditions

The conclusions stated here assume the vector holding the conditions is already defined. If this isn't the case and you want to apply SIMD instructions, you should implement `ifelse` operating on scalars, without a vector of conditions. This allows you to avoid memory allocations, while still applying SIMD effectively. The following example illustrates this point. [2]

```
x = rand(1_000_000)

function foo(x)
    output     = similar(x)
    bitvector  = x .> 0.5


    @inbounds @simd for i in eachindex(x)
        output[i] = ifelse(bitvector[i], x[i]/i, x[i]*i)
    end

    return output
end
```

```
julia> foo($x)
  3.254 ms (7 allocations: 7.749 MiB)
```

```
x = rand(1_000_000)

function foo(x)
    output     = similar(x)
    boolvector = Vector{Bool}(undef,length(x))
        boolvector .= x .> 0.5

    @inbounds @simd for i in eachindex(x)
        output[i] = ifelse(boolvector[i], x[i]/i, x[i]*i)
    end

    return output
end
```

```
julia> foo($x)
  704.903 μs (5 allocations: 8.583 MiB)
```

```
x = rand(1_000_000)

function foo(x)
    output     = similar(x)



    @inbounds @simd for i in eachindex(x)
        output[i] = ifelse(x[i]>0.5, x[i]/i, x[i]*i)
    end

    return output
end
```

```
julia> foo($x)
  467.963 μs (3 allocations: 7.629 MiB)
```

ALGEBRAIC OPERATIONS AS COMPOUND CONDITIONS

We leverage algebraic equivalences to express conditions in ways that allow us to avoid the creation of branches. Mathematically, given a set $\{b_i\}_{i=1}^n$ where $b_i \in \{0,1\}$:

- all conditions are satisfied when

$$\prod_{i=1}^{n} c_i = 1$$

- at least one condition is satisfied when

$$1 - \prod_{i=1}^{n} (1 - c_i) = 1$$

Given two Boolean scalars `c1` and `c2`, these equivalences in Julia become:

- `c1 && c2` is `Bool(c1 * c2)`

- `c1 || c2` is `Bool(1 - !c1 * !c2)`

For instance, with for-loops:

```
x                 = rand(1_000_000)
y                 = rand(1_000_000)


function foo(x,y)
    output = 0.0

    @inbounds @simd for i in eachindex(x)
        if (x[i] > 0.3) && (y[i] < 0.6) && (x[i] > y[i])
            output += x[i]
        end
    end

    return output
end
```

```
julia> foo($x)
  2.063 ms (0 allocations: 0 bytes)
```

```
x                 = rand(1_000_000)
y                 = rand(1_000_000)


function foo(x,y)
    output = 0.0

    @inbounds @simd for i in eachindex(x)
        if (x[i] > 0.3)  * (y[i] < 0.6)  * (x[i] > y[i])
            output += x[i]
        end
    end

    return output
end
```

```
julia> foo($x)
  865.019 μs (0 allocations: 0 bytes)
```

```
x                = rand(1_000_000)
y                = rand(1_000_000)


function foo(x,y)
    output = 0.0

    @inbounds @simd for i in eachindex(x)
        if (x[i] > 0.3) || (y[i] < 0.6) || (x[i] > y[i])
            output += x[i]
        end
    end

    return output
end
```

```
julia> foo($x)
  2.917 ms (0 allocations: 0 bytes)
```

```
x                = rand(1_000_000)
y                = rand(1_000_000)


function foo(x,y)
    output = 0.0

    @inbounds @simd for i in eachindex(x)
        if Bool(1 - !(x[i] > 0.3)  * !(y[i] < 0.6)  * !(x[i] > y[i]))
            output += x[i]
        end
    end

    return output
end
```

```
julia> foo($x)
  868.655 µs (0 allocations: 0 bytes)
```

Instead, with broadcasting, these equivalences become:

```
x              = rand(1_000_000)
y              = rand(1_000_000)


foo(x,y)       = @. ifelse((x>0.3) && (y<0.6) && (x>y), x,y)
```

```
julia> foo($x)
  5.223 ms (3 allocations: 7.629 MiB)
```

```
x               = rand(1_000_000)
y               = rand(1_000_000)


foo(x,y)        = @. ifelse((x>0.3) *  (y<0.6) *  (x>y), x,y)
```

```
julia> foo($x)
  537.296 μs (3 allocations: 7.629 MiB)
```

```
x               = rand(1_000_000)
y               = rand(1_000_000)


foo(x,y)        = @. ifelse((x>0.3) || (y<0.6) || (x>y), x,y)
```

```
julia> foo($x)
  3.248 ms (3 allocations: 7.629 MiB)
```

```
x               = rand(1_000_000)
y               = rand(1_000_000)


foo(x,y)        = @. ifelse(Bool(1 - !(x>0.3) * !(y<0.6) * !(x>y)), x,y)
```

```
julia> foo($x)
  497.927 μs (3 allocations: 7.629 MiB)
```

FOOTNOTES

[1.] Recall that the compiler may automatically disable bounds checking in simple scenarios. For instance, this is the case when only `x` is indexed and `eachindex(x)` is employed as the iteration range. Instead, explicit use of `@inbounds` becomes necessary when we're indexing `x` and another vector, as in the examples below.

[2.] Note that the approach for `Vector{Bool}` is somewhat different to the examples considered above. As we don't have a vector of conditions already defined, it's optimal to create `Vector{Bool}` directly, rather than defining it as a transformation of the `BitVector`. In this way, we avoid unnecessary memory allocations too.