10f. SIMD: Branchless Code

Martin Alfaro
PhD in Economics

BRANCHES

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 arises since 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 incur a performance penalty.

This section explores strategies for efficiently applying SIMD in the presence of conditional operations. We'll first examine scenarios where the compiler introduces conditional statements as an artifact 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. This includes standard scenarios where conditions are explicitly introduced in the code. In this respect, we'll revisit the usual approaches to expressing conditions, focusing on their internal implementation. We'll outline their relative strengths and limitations, indicating which approaches are more conducive to SIMD optimizations. Finally, we'll show that conditional statements can be equivalently recast as algebraic operations, which effectively removes the branching logic that disrupts SIMD execution.

TYPE STABILITY AND BOUNDS CHECKING AS AVOIDABLE CONDITIONS

Two patterns in Julia introduce hidden branches that hurt SIMD performance: type-unstable functions and bounds checking in array indexing. These conditions arise internally from compiler decisions, rather than explicit code, making them easy to overlook.

When a function is type-unstable, Julia generates multiple execution branches, one for each type. Those extra branches, while invisible to you, still disrupt the uniform instruction flow required by SIMD. The remedies for this case are the same as those for fixing type instabilities. Regardless of any SIMD consideration, recall that you should always strive for type stability. Type instability is a major performance bottleneck, with any attempt to achieve high performance becoming nearly impossible without addressing it.

The other source of hidden conditionals arises in for-loops, which perform bounds checking by default. This operation represents a subtle form of conditional execution, where each iteration is executed only when indices remain within bounds.

In relation to this scenario, the example below demonstrates two key insights. First, merely adding `@inbounds` can be enough to induce the compiler to apply SIMD instructions, rendering `@simd` annotations redundant for performance improvements. This explains why using `@inbounds @simd` in the example has a negligible impact on execution times. [1] Second, the example highlights that **adding `@inbounds` is a necessary precondition for the application of SIMD**. Simply using `@simd` on its own won't trigger the implementation of SIMD instructions, as the compiler may still be hindered by the bounds checks. Overall, if we aim to apply SIMD in a for-loop, we should prepend it with `@inbounds @simd`.

```
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)
  775.469 μs (2 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)
  792.687 μs (2 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)
  474.154 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] = 2/x[i]
    end

    return output
end
```

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

Broadcasting and For-Loops

Broadcasting disables bounds checking and strongly favors SIMD by default, often making it appear more performant than a simple for-loop. Despite this, broadcasting essentially serves as a concise notation for implementing a for-loop. As the example below demonstrates, a for-loop that has been optimized with `@inbounds` and `@simd` will typically exhibit a similar level of performance to a broadcasted operation.

```
x      = rand(1_000_000)
foo(x) = 2 ./ x
```

```
julia> @btime foo($x)
  431.487 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] = 2/x[i]
    end

    return output
end
```

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

APPROACHES TO CONDITIONAL STATEMENTS

When conditions are part of the program's logical flow and therefore unavoidable, we need to inquire on what approach is better for the introduction.

Specifically, 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. In contrast, an eager approach computes `1 + 1`, regardless of whether `C` is satisfied. If `C` turns out to be false, the computation remains unused.

When conditional statements are applied only once, a lazy approach is almost always more performant as it avoids needless computations. However, inside a for-loop, SIMD can compute multiple operations simultaneously. Consequently, 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 nuance 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` and `&&` favor lazy computations, only evaluating the necessary components based on the truth value of the condition.

The following example demonstrates this computational difference through a reduction operation that's contingent on a condition. [2]

```
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)
  415.373 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)
  414.155 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)
  393.046 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)
  87.192 μs (0 allocations: 0 bytes)
```

As the example reveals, an eager computation doesn't automatically imply the application of SIMD. This is precisely why `@simd` is included, which provides a hint to the compiler that vectorizing the operation might be beneficial. In fact, we'll show later that adding `@simd` when conditions comprise multiple statements could prompt the compiler to vectorize conditions, while still relying on a lazy evaluation.

It's also worth remarking that applying SIMD instructions doesn't necessarily increase performance. The example below demonstrates this point, where the compiler adopts a SIMD approach through `ifelse`.

```
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)
  2.806 ms (0 allocations: 0 bytes)
```

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

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

```
julia> foo!($output,$x)
  2.888 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)
  16.026 ms (0 allocations: 0 bytes)
```

TERNARY OPERATORS

Ternary operators are an alternative approach for conditional statements, consisting of the form `<condition> ? <action if true> : <action if false>`. Unlike the previous methods, this form relies on heuristics to determine whether an eager or lazy approach should be used. The decision depends on which strategy would more likely be faster in the application considered.

For the illustrations, we'll consider examples where we directly add `@inbounds` and `@simd` in each approach.

DIFFERENT CHOICES

Starting with the same example as above, we show that the ternary operator could choose 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)
  422.480 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)
  85.895 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)
  87.881 μs (0 allocations: 0 bytes)
```

Instead, the ternary operator implements 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)
  405.304 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)
  3.470 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)
  421.493 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 scenario, it actually chooses the slowest 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)
  26.620 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)
  16.864 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)
  26.517 ms (0 allocations: 0 bytes)
```

SCENARIOS UNDER WHICH EACH APPROACH IS BETTER

As a rule of thumb, **an eager approach is potentially more performant when branches comprise simple algebraic computations**. On the contrary, conditional statements with **computational-demanding operations will more likely benefit from a lazy implementation**. In fact, this is a heuristic that ternary operators commonly follow.

To demonstrate this, the following example considers a conditional statement where only one branch has a computation, which in turn 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)
  416.681 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)
  86.242 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)
  86.370 μs (0 allocations: 0 bytes)
```

Instead, the following scenario considers a branch with more 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)
  12.346 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)
  16.552 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)
  12.492 ms (0 allocations: 0 bytes)
```

VECTOR OF CONDITIONS

Next, we consider scenarios where you already have 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 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 considering the vector transformation.

```
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)
  3.393 ms (2 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)
  862.798 µs (4 allocations: 8.583 MiB)
```

No Vector of Conditions

The conclusions stated here assumes that you already have the vector holding the conditions. If this isn't the case and you want to apply SIMD instructions, you should implement `ifelse` without a vector of conditions. This allows you to avoid memory allocations, while still applying SIMD effectively. The following example illustrates this point. [3]

```
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)
  3.628 ms (6 allocations: 7.753 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)
  774.952 μs (4 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)
  501.114 μs (2 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

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

In terms of Julia, given two Boolean scalars `c1` and `c2`, these equivalences 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)
  2.116 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)
  905.078 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)
  2.724 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)
  889.879 μs (0 allocations: 0 bytes)
```

While with broadcasting:

```
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)
  5.356 ms (2 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)
  541.621 μs (2 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)
  3.354 ms (2 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)
  536.276 μs (2 allocations: 7.629 MiB)
```

FOOTNOTES

[1.] Recall that the compiler may automatically disable bounds checking in some cases, especially in straightforward cases. For instance, this would be the case in our example if only `x` had been indexed and `eachindex(x)` were employed as the iteration range. This is in contrast to scenarios like the one below, where we're indexing both `x` and `output`.

[2.] Note that `ifelse` requires specifying an operation for when the condition is true and another when it's not. For a sum reduction, this is handled by returning zero when the condition isn't met.

[3.] Note that the approach for `Vector{Bool}` is somewhat different to the examples we 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.