10g. SIMD Packages

Martin Alfaro
PhD in Economics

INTRODUCTION

Up to this point, we've leaned on the built-in `@simd` macro to encourage vectorization in for-loops. This approach, nonetheless, exhibits several limitations.

The first limitation is control. `@simd` acts as a suggestion, rather than a strict command: it hints to the compiler that SIMD optimizations might improve performance, but the implementation decision is ultimately up to the compiler's discretion. Second, `@simd` is designed to be conservative, prioritizing code safety over speed. In practice, this means it won't implement many aggressive transformations that SIMD typically requires to unlock its full capabilities.

To overcome these shortcomings, we'll introduce the `@turbo` macro from the `LoopVectorization` package. Rather than merely suggesting vectorization, `@turbo` rewrites for-loops and broadcast expressions into a form that's explicitly structured for SIMD execution. That extra power comes with an important shift in responsibility: `@turbo` assumes that your code satisfies the conditions necessary for these transformations, placing the burden of safety and correctness on the user.

A further advantage of `LoopVectorization` is its integration with the `SLEEFPirates` package. This enables SIMD-accelerated implementations of common mathematical functions, including logarithms, exponentials, powers, and trigonometric operations.

CAVEATS ABOUT IMPROPER USE OF @TURBO

In contrast to `@simd`, applying `@turbo` demands particular care, as a misapplication can silently produce incorrect results. The risk stems from the additional assumptions that `@turbo` makes to enable more aggressive optimizations. In particular, `@turbo` operates under two key assumptions:

- **No out-of-bound access**: `@turbo` omits index bound checks, thus relying on that all indices are valid.
- **Iteration order invariance**: `@turbo` supposes that, aside from reductions, the outcome of the for-loop doesn't depend on the order in which iterations execute.

The second assumption is especially relevant. Even if your for-loop has no explicit dependency, floating-point arithmetic are still sensitive to reordering because it isn't associative. This causes results to depend on the iteration order. Integer operations, by contrast, are unaffected. The following

example illustrates the problem: because each iteration depends on the result of the previous one, applying `@turbo` violates the iteration-order invariance assumption and therefore yields incorrect results.

NO MACRO

```
x = [0.1, 0.2, 0.3]

function foo!(x)
    for i in 2:length(x)
        x[i] = x[i-1] + x[i]
    end
end
```

```
julia> foo!(x)
julia> x
3-element Vector{Float64}:
 0.1
 0.3
 0.6
```

@SIMD

```
x = [0.1, 0.2, 0.3]

function foo!(x)
    @inbounds @simd for i in 2:length(x)
        x[i] = x[i-1] + x[i]
    end
end
```

```
julia> foo!(x)
julia> x
3-element Vector{Float64}:
 0.1
 0.3
 0.6
```

@TURBO

```
x = [0.1, 0.2, 0.3]

function foo!(x)
    @turbo for i in 2:length(x)
        x[i] = x[i-1] + x[i]
    end
end
```

```
julia> foo!(x)
julia> x
3-element Vector{Float64}:
 0.1
 0.3
 0.5
```

Considering that `@turbo` isn't suitable for all operations, we next present cases where the macro can be safely applied.

SAFE APPLICATIONS OF @TURBO

Safe applications of `@turbo` fall into two broad categories: **embarrassingly parallel problems** and **reductions**.

In the first case, **iterations are completely independent**, making execution order irrelevant for the final outcome. The example below illustrates this case by performing an independent transformation on each element of a vector. We also show that the use of `@turbo` isn't restricted to for-loops, also allowing for broadcast operations.

DEFAULT

```
x           = rand(1_000_000)
calculation(a) = a * 0.1 + a^2 * 0.2 - a^3 * 0.3 - a^4 * 0.4

function foo(x)
    output     = similar(x)

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

    return output
end
```

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

@SIMD

```
x           = rand(1_000_000)
calculation(a) = a * 0.1 + a^2 * 0.2 - a^3 * 0.3 - a^4 * 0.4

function foo(x)
    output     = similar(x)

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

    return output
end
```

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

@TURBO (FOR-LOOP)

```
x              = rand(1_000_000)
calculation(a) = a * 0.1 + a^2 * 0.2 - a^3 * 0.3 - a^4 * 0.4

function foo(x)
    output     = similar(x)

    @turbo for i in eachindex(x)
        output[i] = calculation(x[i])
    end

    return output
end
```

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

@TURBO (BROADCASTING)

```
x              = rand(1_000_000)
calculation(a) = a * 0.1 + a^2 * 0.2 - a^3 * 0.3 - a^4 * 0.4

foo(x)         = @turbo calculation.(x)
```

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

The second safe application is **reductions**. While reductions introduce dependencies across iterations, they represent a special case that `@turbo` handles properly.

DEFAULT

```
x              = rand(1_000_000)
calculation(a) = a * 0.1 + a^2 * 0.2 - a^3 * 0.3 - a^4 * 0.4

function foo(x)
    output     = 0.0

    for i in eachindex(x)
        output += calculation(x[i])
    end

    return output
end
```

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

@SIMD

```
x              = rand(1_000_000)
calculation(a) = a * 0.1 + a^2 * 0.2 - a^3 * 0.3 - a^4 * 0.4

function foo(x)
    output     = 0.0

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

    return output
end
```

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

@TURBO

```
x              = rand(1_000_000)
calculation(a) = a * 0.1 + a^2 * 0.2 - a^3 * 0.3 - a^4 * 0.4

function foo(x)
    output     = 0.0

    @turbo for i in eachindex(x)
        output += calculation(x[i])
    end

    return output
end
```

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

SPECIAL FUNCTIONS

Another important application of `LoopVectorization` arises through its integration with the library *SLEEF* (an anocrym for "SIMD Library for Evaluating Elementary Functions"). This accelerates the evaluation of several mathematical functions, including the exponential, logarithmic, power, and trigonometric functions. SLEEF is exposed in `LoopVectorization` via the `SLEEFPirates` package,

Below, we illustrate the use of `@turbo` for each type of function. For a complete list of supported functions, see the SLEEFPirates documentation. All the examples rely on an element-wise transformation of `x` via the function `calculation`, which will take a different form depending on the function illustrated.

LOGARITHM

DEFAULT

```
x              = rand(1_000_000)
calculation(a) = log(a)

function foo(x)
    output     = similar(x)

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

    return output
end
```

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

@SIMD

```
x              = rand(1_000_000)
calculation(a) = log(a)

function foo(x)
    output     = similar(x)

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

    return output
end
```

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

@TURBO (FOR-LOOP)

```
x              = rand(1_000_000)
calculation(a) = log(a)

function foo(x)
    output     = similar(x)

    @turbo for i in eachindex(x)
        output[i] = calculation(x[i])
    end

    return output
end
```

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

@TURBO (BROADCASTING)

```
x              = rand(1_000_000)
calculation(a) = log(a)

foo(x) = @turbo calculation.(x)
```

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

EXPONENTIAL FUNCTION

DEFAULT

```
x              = rand(1_000_000)
calculation(a) = exp(a)

function foo(x)
    output     = similar(x)

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

    return output
end
```

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

@SIMD

```
x              = rand(1_000_000)
calculation(a) = exp(a)

function foo(x)
    output     = similar(x)

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

    return output
end
```

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

@TURBO (FOR-LOOP)

```
x              = rand(1_000_000)
calculation(a) = exp(a)

function foo(x)
    output     = similar(x)

    @turbo for i in eachindex(x)
        output[i] = calculation(x[i])
    end

    return output
end
```

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

@TURBO (BROADCASTING)

```
x              = rand(1_000_000)
calculation(a) = exp(a)

foo(x) = @turbo calculation.(x)
```

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

POWER FUNCTIONS

This includes any operation comprising terms x^y.

DEFAULT

```
x              = rand(1_000_000)
calculation(a) = a^4

function foo(x)
    output     = similar(x)

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

    return output
end
```

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

@SIMD

```
x              = rand(1_000_000)
calculation(a) = a^4

function foo(x)
    output     = similar(x)

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

    return output
end
```

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

@TURBO (FOR-LOOP)

```
x              = rand(1_000_000)
calculation(a) = a^4

function foo(x)
    output     = similar(x)

    @turbo for i in eachindex(x)
        output[i] = calculation(x[i])
    end

    return output
end
```

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

@TURBO (BROADCASTING)

```
x              = rand(1_000_000)
calculation(a) = a^4

foo(x) = @turbo calculation.(x)
```

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

The implementation for power functions includes calls to other function, such as the one for square roots.

DEFAULT

```
x              = rand(1_000_000)
calculation(a) = sqrt(a)

function foo(x)
    output     = similar(x)

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

    return output
end
```

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

@SIMD

```
x              = rand(1_000_000)
calculation(a) = sqrt(a)

function foo(x)
    output     = similar(x)

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

    return output
end
```

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

@TURBO (FOR-LOOP)

```
x              = rand(1_000_000)
calculation(a) = sqrt(a)

function foo(x)
    output     = similar(x)

    @turbo for i in eachindex(x)
        output[i] = calculation(x[i])
    end

    return output
end
```

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

@TURBO (BROADCASTING)

```
x              = rand(1_000_000)
calculation(a) = sqrt(a)

foo(x) = @turbo calculation.(x)
```

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

TRIGONOMETRIC FUNCTIONS

Among others, `@turbo` can handle the functions `sin`, `cos`, and `tan`. Below, we demonstrate its use with `sin`.

DEFAULT

```
x              = rand(1_000_000)
calculation(a) = sin(a)

function foo(x)
    output     = similar(x)

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

    return output
end
```

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

@SIMD

```
x              = rand(1_000_000)
calculation(a) = sin(a)

function foo(x)
    output     = similar(x)

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

    return output
end
```

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

@TURBO (FOR-LOOP)

```
x              = rand(1_000_000)
calculation(a) = sin(a)

function foo(x)
    output     = similar(x)

    @turbo for i in eachindex(x)
        output[i] = calculation(x[i])
    end

    return output
end
```

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

@TURBO (BROADCASTING)

```
x              = rand(1_000_000)
calculation(a) = sin(a)

foo(x)         = @turbo calculation.(x)
```

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