10g. Packages For SIMD

Martin Alfaro
PhD in Economics

INTRODUCTION

So far, we've relied on the built-in `@simd` macro to apply SIMD instructions. This approach, nonetheless, exhibits several limitations. First, `@simd` acts as a suggestion rather than a strict command: it hints to the compiler that SIMD optimizations might improve performance, but ultimately leaves the implementation decision up to the compiler's discretion. Second, `@simd` prioritizes code safety over speed, restricting access to advanced SIMD features in favor of avoiding unintended bugs. Third, its application is limited to for-loops.

To overcome these flaws, we introduce the `@turbo` macro from the `LoopVectorization` package. Unlike `@simd`, `@turbo` enforces SIMD optimizations when called, eliminating ambiguity and ensuring vectorized instructions are applied. It also employs more aggressive optimizations than `@simd`, shifting the responsibility for safe usage onto users. Finally, `@turbo` supports both for-loops and broadcasting operations.

CAVEATS ABOUT IMPROPER USE OF @TURBO

In contrast to `@simd`, applying `@turbo` requires extra caution, as its misapplication can yield incorrect results. This risk stems from `@turbo`'s additional assumptions about the operations processed, which are incorporated to enable more aggressive optimization. In particular:

- `@turbo` doesn't perform index bound checking, potentially leading to out-of-bounds memory access.
- `@turbo` assumes the outcome is invariant to iteration order. The only exception is reduction operations, which are handled properly.

The latter is an issue when it comes to floating-point operations, but not for integers. It essentially determines that dependent iterations can yield incorrect results. The following example illustrates this problem, where each iteration depends on a previous iteration's outcome.

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 applied safely.

SAFE APPLICATIONS OF @TURBO

There are two safe applications of `@turbo` that cover a wide range of cases. The first one is **when iterations are completely independent**, making execution order irrelevant for the outcome obtained.

For instance, the following code snippet applies an independent transformation to each element of a vector.

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

The second application is **reductions**. Although reductions comprise dependent 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> @ctime 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> @ctime 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> @ctime foo($x)
  179.364 μs (0 allocations: 0 bytes)
```

SPECIAL FUNCTIONS

An indirect application of the package `LoopVectorization` occurs because it leverages the library *SLEEF*. This is an acronym for "SIMD Library for Evaluating Elementary Functions". SLEEF is available in Julia through the package `SLEEFPirates` and is designed to boost the computations of some mathematical functions via SIMD instructions. In particular, it speeds up the computations of the exponential, logarithmic, power, and trigonometric functions.

Below, we illustrate the use of `@turbo` for each type of function. See **here** for a list of all the functions supported.

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

POWER FUNCTIONS

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

The implementation of power functions includes 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> @ctime 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> @ctime 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> @ctime 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> @ctime 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> @ctime 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> @ctime 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> @ctime 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> @ctime foo($x)
  1.315 ms (3 allocations: 7.629 MiB)
```