10g. **SIMD Packages**

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 the implementation decision is up to the compiler's discretion. Second, `@simd` prioritizes code safety over speed, restricting access to advanced SIMD features to avoid bugs. Third, its application is limited to for-loops.

To overcome these shortcomings, we introduce the `@turbo` macro from the `LoopVectorization` package. Unlike `@simd`, the `@turbo` macro enforces SIMD optimizations, guaranteeing the application of vectorized instructions. It also performs more aggressive optimizations than `@simd`, shifting the responsibility for correctness and safety onto the user. Additionally, `@turbo` extends beyond for-loops, also supporting broadcasting operations. A final advangate of `@turbo` is that it integrates with the `SLEEFPirates` package. This provides SIMD-accelerated implementations of common mathematical functions such as logarithms, exponentials, powers, and trigonometric operations.

CAVEATS ABOUT IMPROPER USE OF @TURBO

In contrast to `@simd`, applying `@turbo` requires extra caution, as its misapplication can lead to incorrect results. The risk stems from the additional assumptions that `@turbo` makes to enable more aggressive optimization. In particular:

- **No bound checking**: `@turbo` omits index bound checks, potentially leading to out-of-bounds memory access.

- **Iteration order invariance**: `@turbo` assumes the outcome doesn't depend on the order of iteration, with the sole exception of reduction operations.

The second assumption is particularly relevant for floating-point arithmetic, where non-associativity can cause **results to vary with iteration order**. Integer operations, by contrast, are unaffected. The following example demonstrates this issue: because each iteration depends on the outcome of the previous one, applying `@turbo` produces incorrect results.

NO MACRO

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

@SIMD

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

@TURBO

```julia
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
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

There are two safe applications of `@turbo` that cover a wide range of cases. The first applies **when 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. Importantly, `@turbo` isn't restricted to for-loops, also allowing for broadcast operations. We show both applications.

DEFAULT

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

@SIMD

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

@TURBO (FOR-LOOP)

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

@TURBO (BROADCASTING)

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

The second safe application is **reductions**. While reductions consists of dependent iterations, they're a special case that `@turbo` handles properly.

DEFAULT

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

@SIMD

```julia
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

```julia
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"). SLEEF is exposed in `LoopVectorization` via the `SLEEFPirates` package, which accelerates the evaluation of several mathematical functions. They include the exponential, logarithmic, power, and trigonometric functions.

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

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

@SIMD

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

@TURBO (FOR-LOOP)

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

@TURBO (BROADCASTING)

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

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

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

EXPONENTIAL FUNCTION

DEFAULT

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

@SIMD

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

@TURBO (FOR-LOOP)

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

@TURBO (BROADCASTING)

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

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

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

POWER FUNCTIONS

This includes any operation comprising terms x^y.

DEFAULT

```julia
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
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

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

@SIMD

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

@TURBO (FOR-LOOP)

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

@TURBO (BROADCASTING)

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

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

```julia
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

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

@SIMD

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

@TURBO (FOR-LOOP)

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

@TURBO (BROADCASTING)

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

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

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