10g. Packages For SIMD

Martin Alfaro
PhD in Economics

INTRODUCTION

So far, we've been using the built-in macro `@simd` to apply SIMD instructions. This macro is relatively limited in certain respects. For one, it only hints at the potential advantages of applying SIMD, leaving the final decision implementation to the compiler's discretion. Moreover, it only provides basic features of SIMD, prioritizing code safety over performance.

Next, we introduce the macro `@turbo` from the package `LoopVectorization`, which offers several distinct advantages. First, it enforces SIMD optimizations when invoked, rather than merely suggesting them. Furthermore, it applies more aggressive optimizations compared to `@simd`. Finally, `@turbo` supports both for-loops and broadcasting operations, contrasting with `@simd`'s exclusive applicability to for-loops.

CAVEATS ABOUT IMPROPER USE OF @TURBO

In contrast to `@simd`, applying `@turbo` requires some caution, as it may lead to incorrect results if misapplied. This issue arises because the macro makes additional assumptions about the operations being performed, with the goal of applying optimizations more aggressively. In particular:

- `@turbo` never checks index bounds, potentially leading to out-of-bounds memory access.
- `@turbo` assumes the outcome is independent of the iteration order (except for reduction operations).

An example of the latter is when computing a vector holding cumulative sums of another vector. This can be observed below, where we verify the final result by summing all values in the output vector.

NO MACRO

```julia
x = rand(1_000_000)

function foo(x)
    output = copy(x)

    for i in 2:length(x)
        output[i] = output[i-1] + x[i]
    end

    return output
end
```

```
julia> sum(foo(x))
2.50038e11
```

@SIMD

```julia
x = rand(1_000_000)

function foo(x)
    output = copy(x)

    @inbounds @simd for i in 2:length(x)
        output[i] = output[i-1] + x[i]
    end

    return output
end
```

```
julia> sum(foo(x))
2.50038e11
```

@TURBO

```julia
x = rand(1_000_000)

function foo(x)
    output = copy(x)

    @turbo for i in 2:length(x)
        output[i] = output[i-1] + x[i]
    end

    return output
end
```

```
julia> sum(foo(x))
1.03169e6
```

CASES COVERED

Considering that `@turbo` isn't suitable for all operations, let's present two of its primary applications. The first one arises **when iterations are completely independent**, making their execution order irrelevant.

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

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> @ctime foo($x)
  3.098 ms (2 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> @ctime foo($x)
  5.070 ms (2 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> @ctime foo($x)
  492.031 μs (2 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> @ctime foo($x)
  406.089 μs (2 allocations: 7.629 MiB)
```

The second application is **reductions**. Although reductions inherently involve dependent iterations, they represent 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> @ctime foo($x)
  3.083 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> @ctime foo($x)
  3.299 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> @ctime foo($x)
  179.722 μs (0 allocations: 0 bytes)
```

SPECIAL FUNCTIONS

The package `LoopVectorization` leverages the library *SLEEF*, which is an acronym for "SIMD Library for Evaluating Elementary Functions". SLEEF is available in Julia through the package `SLEEFPirates` and it's designed to boost the mathematical computations of some functions by utilizing 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

```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> @ctime foo($x)
  3.395 ms (2 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> @ctime foo($x)
  3.414 ms (2 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> @ctime foo($x)
  1.229 ms (2 allocations: 7.629 MiB)
```

@TURBO (BROADCASTING)

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

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

```
julia> @ctime foo($x)
  1.237 ms (2 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> @ctime foo($x)
  1.962 ms (2 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> @ctime foo($x)
  1.950 ms (2 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> @ctime foo($x)
  600.413 μs (2 allocations: 7.629 MiB)
```

@TURBO (BROADCASTING)

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

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

```
julia> @ctime foo($x)
  596.388 μs (2 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)
  3.100 ms (2 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)
  3.305 ms (2 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)
  498.762 μs (2 allocations: 7.629 MiB)
```

@TURBO (BROADCASTING)

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

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

```
julia> @ctime foo($x)
  434.407 μs (2 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)
  1.232 ms (2 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> @ctime foo($x)
  1.231 ms (2 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> @ctime foo($x)
  614.203 μs (2 allocations: 7.629 MiB)
```

@TURBO (BROADCASTING)

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

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

```
julia> @ctime foo($x)
  614.241 μs (2 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> @ctime foo($x)
  3.841 ms (2 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> @ctime foo($x)
  3.767 ms (2 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> @ctime foo($x)
  1.386 ms (2 allocations: 7.629 MiB)
```

@TURBO (BROADCASTING)

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

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

```
julia> @ctime foo($x)
  1.384 ms (2 allocations: 7.629 MiB)
```