10c. Introduction to SIMD

[Martin Alfaro](#)
PhD in Economics

INTRODUCTION

Single Instruction, Multiple Data (SIMD) is an optimization technique widely embraced in modern CPU architectures. At its core, SIMD allows a single CPU instruction to process multiple data points concurrently instead of sequentially processing each one individually. This approach can yield substantial performance gains, particularly for workloads involving simple identical calculations repeated across multiple data elements. [1]

To illustrate the power of SIMD, consider a computation consisting of four separate addition operations. Without SIMD, you'd have to execute four distinct instructions, one for each addition. Instead, SIMD makes it possible to bundle the four additions into a single instruction, with the CPU processing them all at once. In an ideal scenario, the time required to complete four additions using SIMD or one addition without it could be the same.

The efficiency of SIMD comes from its ability to leverage parallelism within a single CPU core. By operating on vectors rather than individual elements, SIMD instructions can execute the same operation on multiple data simultaneously. This is why SIMD is usually referred to as **vectorization**.

Throughout the sections, we'll cover two approaches for implementing SIMD instructions.

- Julia's native capabilities.
- The package `LoopVectorization`.

In this section, we'll concentrate on the built-in tools for applying SIMD. In particular, we'll present conditions that trigger its automatic application. Furthermore, we'll introduce the `@simd` macro, which allows for manual implementation in for-loops. Instead, the study of `LoopVectorization` is relegated to subsequent sections. As we'll see, this package implements more aggressive optimizations, relative to Julia's base.

WHAT IS SIMD?

SIMD is a type of instruction-level parallelism that occurs within a single processor core. It's particularly effective for basic arithmetic operations, such as addition and multiplication, when the same operation must be applied to multiple data. Given the nature of these operations, it's no surprise that one of the primary applications of SIMD is in linear algebra.

At the heart of SIMD lies the process of vectorization, where data is split into sub-vectors that can be processed as single units. To facilitate this, modern processors include specialized SIMD registers designed for this purpose. Desktop and laptop processors these days typically feature 256-bit wide

registers for vectorized operations, which can hold four 64-bit floating-point numbers.

To illustrate the workings of SIMD, consider the task of adding two vectors, each comprising four elements. Specifically, let `x = [1, 2, 3, 4]` and `y = [10, 20, 30, 40]`. In traditional scalar processing, performing the operation `x + y` would require four separate addition operations, one for each pair of numbers. In contrast, all four additions can be performed with a single instruction under SIMD, producing the result `[11, 22, 33, 44]` in one step.

For larger vectors, the process remains fundamentally the same. The only difference is that the processor first partitions the vectors into sub-vectors that fit the register's capacity. After this, the processor computes all the operations within each segment simultaneously, repeating the procedure for each sub-vector.

BROADCASTING VS FOR-LOOPS

The previous analysis shows that SIMD applies to computations involving collections. Based on this, we can identify **two types of operations that can potentially benefit from SIMD instructions: for-loops and broadcasting**. The latter is automatically implemented by the compiler, without requiring any special consideration from the user.

Instead, the upcoming sections will focus on the application of SIMD in for-loops. This will require exploring the conditions under which SIMD instructions can be applied. If these conditions aren't met, SIMD will become infeasible or reduce its effectiveness substantially. In addition to elaborating on these conditions, we'll provide guidance on how to address scenarios that don't conform to them.

To pave the way and shift our attention to for-loops, we conclude this section by illustrating the automatic application of SIMD in broadcasting.

SIMD IN BROADCASTING

The decision whether to apply SIMD instructions is always handled by the compiler, which relies on heuristics to determine when it's beneficial to do so. One situation where Julia strongly favors the application of SIMD is with broadcasting, as can be noticed in the following example.

```
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)
  789.564 μs (2 allocations: 7.629 MiB)
```

```
x      = rand(1_000_000)

foo(x) = 2 ./ x
```

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

The example compares the same operation implemented using a for-loop and broadcasting. While broadcasting automatically leverages SIMD, this isn't necessarily the case with for-loops. Indeed, in this particular example, we'd need to explicitly instruct the compiler to enable SIMD, which accounts for the observed time differences.

FOOTNOTES

1. SIMD isn't exclusive to CPUs. In fact, GPUs also take advantage of it. They're a natural fit for SIMD, as their architecture was conceived for parallel processing of simple identical operations.