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, rather than sequentially processing them one by one. This parallel approach can yield substantial performance gains, especially for workloads involving simple identical calculations repeated across multiple data elements. [1]

The efficiency of SIMD lies in 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 points simultaneously. This is why the process of applying SIMD is often referred to as **vectorization**.

To illustrate, consider a computation involving four separate addition operations. Without SIMD, the computer would need to execute four distinct instructions, one for each addition. Instead, SIMD makes it possible to bundle the four additions into a single instruction, allowing the CPU to process them all at once. In an ideal scenario, the time required to complete four additions with SIMD would be the same as completing one addition without it.

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

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

This section will exclusively concentrate on the built-in tools for applying SIMD. In particular, we'll explore the conditions that trigger automatic vectorization. The next section we'll introduce the `@simd` macro, which lets you manually apply it in for-loops. The discussion of `LoopVectorization` will be saved for later sections. Relative to Julia's built-in tools, this package often implements more aggressive optimizations, but can also introduce bugs if misused.

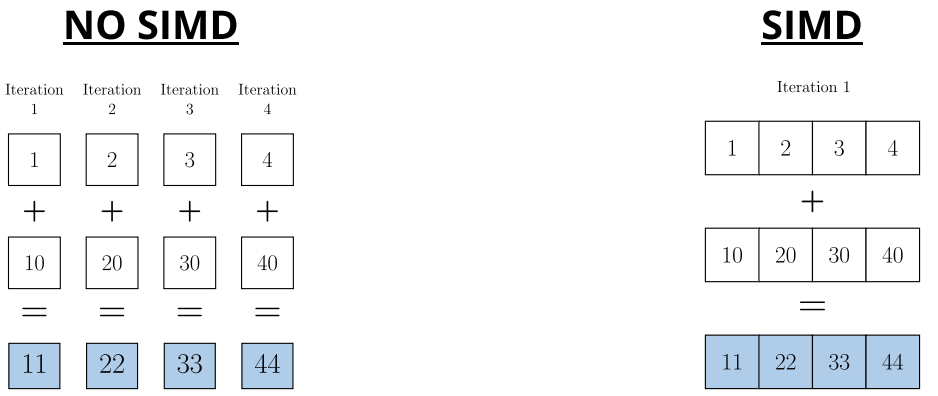
WHAT IS SIMD?

SIMD is a type of data-level parallelism that occurs within a single processor core, applying the same operation to multiple data elements at once. It's particularly effective for basic arithmetic operations, such as addition and multiplication, when the same operation must be applied to multiple data

elements. Given the nature of these operations, it's unsurprising that one of SIMD's primary applications is in linear algebra, where operations like matrix multiplication involve applying identical arithmetic steps to multiple elements.

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 operation, modern processors include specialized SIMD registers designed for this purpose. Today's processors typically feature 256-bit registers for vectorized operations, which are wide enough to hold four values of either `Float64` or `Int64`.

To illustrate the workings of SIMD, consider the task of adding two vectors, `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: `1+10`, `2+20`, `3+30`, `4+40` executed sequentially, thereby producing the result `[11, 22, 33, 44]` in four steps. In contrast, all four additions can be performed with a single instruction under SIMD in one step. The processor loads all four elements of `x` and `y` into the 256-bit register, and then runs a single "sum" instruction to compute all four additions simultaneously.



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 within the register's capacity. After this, the processor computes all the operations within each sub-vector simultaneously, repeatedly applying the same procedure for every sub-vector.

BROADCASTING AND FOR-LOOPS

Based on the previous discussion, we can identify **two types of operations that can potentially benefit from SIMD instructions: for-loops and broadcasting**. The latter operation is included since broadcasting is essentially syntactic sugar for for-loops.

The decision to apply SIMD instructions in its built-in form, nonetheless, is made entirely by the compiler. This relies on a set of heuristics to determine when their use will pay off. In the case of broadcasting, the compiler implements SIMD automatically, without any user intervention. Instead, the automatic application of SIMD in for-loops is only restricted to a few simple cases, delegating to the user the suggestion of whether SIMD should be implemented. This is why the upcoming sections

will identify conditions under which SIMD instructions should be suggested to the compiler. If these conditions aren't met, SIMD will substantially reduce its effectiveness or directly become infeasible. Given this, we'll also provide guidance on how to handle scenarios that aren't well-suited for SIMD.

SIMD IN BROADCASTING

To entirely shift our attention to for-loops in subsequent sections, we conclude by illustrating the automatic application of SIMD in broadcasting.

The following example demonstrates this. It compares the same computation implemented via a for loop and via broadcasting. While broadcasting automatically takes advantage of SIMD, this isn't necessarily true for for-loops (in fact it's not in this particular case).

```
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)
  788.201 μs (3 allocations: 7.629 MiB)
```

```
x      = rand(1_000_000)

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

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

FOOTNOTES

[1] SIMD isn't exclusive to CPUs, with GPUs also taking advantage of it. The architecture of GPUs is a natural fit for SIMD, as they were designed for parallel processing of simple identical operations.