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. The parallel approach can yield substantial performance gains, especially for workloads involving simple identical calculations repeated across multiple data elements. [1]

To illustrate the power of SIMD, 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.

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**.

Throughout the 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 and also introduce the `@simd` macro, which lets you manually apply it in for-loops. We'll save our discussion of `LoopVectorization` 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 instruction-level parallelism that occurs within a single processor core, enabling the performance of the same operation on 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. This means that `1+10`, `2+20`, `3+30`, `4+40` must be 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 "add" instruction to compute all four sums 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 the 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**.

In the case of broadcasting, the compiler implements SIMD automatically, without any user intervention. Instead, the application of SIMD in for-loops isn't guaranteed. This is why the upcoming sections will identify conditions under which SIMD instructions can be applied effectively. If these conditions aren't met, SIMD will become infeasible or substantially reduce its effectiveness. Given this, we'll also provide guidance on how to handle scenarios that aren't well-suited for SIMD.

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 to apply SIMD instructions is made entirely by the compiler, which relies on a set of heuristics to determine when their use will pay off. One case where Julia strongly favors SIMD is in broadcasting operations.

The following example demonstrates this. It compares the same computation implemented using a for loop and using broadcasting. While broadcasting automatically takes advantage of SIMD, this is not necessarily true for for-loops, and in fact it's not in this 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)
  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)
```

FOOTNOTES

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