10e. SIMD: Unit Strides

Martin Alfaro
PhD in Economics

INTRODUCTION

SIMD improves computational performance by simultaneously executing the same operation on multiple data elements. Technically, this is achieved through the use of specialized vector registers, which can hold several values (e.g., 4 floating-point numbers or 4 integers). They allow operations such as multiple additions or multiplications to be completed with a single instruction.

For SIMD to fully exploit this vector-based processing, **data should be organized and accessed in memory** in specific ways. Two fundamental concepts for understanding this are contiguous memory layout and unit stride access.

Contiguous memory layout means that data elements reside in adjacent memory addresses, therefore not exhibiting gaps. Newly allocated arrays satisfy this condition, enabling the load of entire segments directly into vector registers. In contrast, array views reference the original data structure, and thus don't guarantee contiguity. This can potentially result in highly irregular memory access patterns.

In addition, **strides** refer to the step size between consecutive memory accesses, with **unit strides** in particular entailing that elements are accessed sequentially. For example, consider a freshly allocated vector `x`. Then, accessing its elements through `eachindex(x)` (or equivalently `1:length(x)`) ensures a unit stride, as each access moves sequentially to the next element in memory. This contrasts with ranges having a non-unit stride such as `1:2:length(x)` or indices with no predictable pattern (e.g., `[1, 5, 3, 4]` as an index vector).

When it comes to performance, **SIMD is most effectively applied when data is stored in a contiguous memory block and accessed in a unit stride pattern**. This has significant practical implications for creating slices, which can take the form of copies or views. Essentially, this choice gives rise to a trade-off, as views reduce memory allocations but at the expense of hindering computational efficiency. This section explores such a decision.

REVIEW OF INDEXING

For the explanations, we'll utilize the various methods for creating slices, with each differing in how indices are defined: vector indexing, Boolean indexing, and ranges. We present the examples used for each, and then proceed to their explanation.

```
x         = [10, 20, 30]

indices   = sortperm(x)
elements  = x[indices]    # equivalent to `sort(x)`
```

```
julia> sorted_indices
3-element Vector{Int64}:
 1
 2
 3
julia> sorted_elements
3-element Vector{Int64}:
 10
 20
 30
```

```
x         = [2, 3, 4, 5, 6]

indices_1 = 1:length(x)         # unit strides, equivalent to 1:1:length(x)
indices_2 = 1:2:length(x)       # strides two
```

```
julia> collect(indices_1)
5-element Vector{Int64}:
 1
 2
 3
 4
 5
julia> collect(indices_2)
3-element Vector{Int64}:
 1
 3
 5
```

```
x         = [20, 10, 30]

indices   = x .> 15
elements  = x[indices]
```

```
julia> sorted_indices
3-element BitVector:
 1
 0
 1
julia> sorted_elements
2-element Vector{Int64}:
 20
 30
```

As for **vector indexing**, the demonstrations will be based on the function `sortperm`. Given some vector `x`, this function is such that, while `sort(x)` returns a vector with `x`'s elements sorted in ascending order, `sortperm(x)` returns the corresponding indices of these elements.

Similarly, **ranges** can be understood as a special case of vector indexing. They differ in that ranges lazily reference consecutive elements for some given strides. Recall that strides represent the gap between successive elements, and they're included in between the first and last index, i.e. `<first index>:<stride>:<last index>`. The absence of a stride implicitly assumes a step equal to 1.

While vector indexing and ranges reference the indices of `x`, **Boolean indexing** returns a Boolean vector where `1` indicates the element must be kept. This approach will be used for the creation of slices through broadcasted conditions, as in the example provided.

CONTIGUOUS BLOCKS IN MEMORY

In the section discussing decreases in memory allocations, we highlighted the benefits of using views over copies when handling slices. Specifically, views maintain references to the original data, thereby avoiding the cost of additional memory allocation. However, views can lead to irregular memory access patterns if data are too scattered. This is why this section also remarked that copies could outperform views in some scenarios. We're now in a position to explain in more depth why this occurs.

Creating copies of some data structure involves allocating the information in a new contiguous block of memory. This ensures that all elements are stored sequentially, thus offering two key advantages: a quicker fetching of elements and a more effective use of SIMD instructions.

To illustrate this, consider retrieving books from a library. If every book you need resides on a single shelf, collecting them is straightforward—you move once, grab the entire stack, and proceed. This mirrors contiguous memory access. Conversely, if the books are dispersed across different floors and sections, each retrieval demands additional time and effort, akin to non-contiguous access.

In addition to this, our recollection of books would be even more efficient if we had a cart capable of carrying multiple books at once—SIMD operations act like this cart. Note that even without such a cart, the act of gathering contiguous books remains inherently faster, as the physical effort (or computational cycles) is minimized. In other words, the benefits of SIMD are on top of the faster memory access.

VECTOR AND BOOLEAN INDEXING

The following examples highlight the two advantages from contiguous memory access. To isolate their performance effect, we create the slices outside the function to be benchmarked. In this way, the functions we'll present don't entail memory allocations.

For the case of vector indexing:

```
x       = rand(5_000_000)

indices = sortperm(x)
y       = @view x[indices]

function foo(y)
    output = 0.0

    for a in y
        output += a
    end

    return output
end
```

```
julia> @btime foo($y)
  21.481 ms (0 allocations: 0 bytes)
```

```
x       = rand(5_000_000)

indices = sortperm(x)
y       = @view x[indices]

function foo(y)
    output = 0.0

    @simd for a in y
        output += a
    end

    return output
end
```

```
julia> @btime foo($y)
  20.754 ms (0 allocations: 0 bytes)
```

```
x       = rand(5_000_000)

indices = sortperm(x)
y       = x[indices]

function foo(y)
    output = 0.0

    for a in y
        output += a
    end

    return output
end
```

```
julia> @btime foo($y)
  2.281 ms (0 allocations: 0 bytes)
```

```
x       = rand(5_000_000)

indices = sortperm(x)
y       = x[indices]

function foo(y)
    output = 0.0

    @simd for a in y
        output += a
    end

    return output
end
```

```
julia> @btime foo($y)
  902.847 μs (0 allocations: 0 bytes)
```

Instead, for Boolean indexing:

```
x       = rand(5_000_000)

indices = x .> 0.5
y       = @view x[indices]

function foo(y)
    output = 0.0

    for a in y
        output += a
    end

    return output
end
```

```
julia> @btime foo($y)
  2.206 ms (0 allocations: 0 bytes)
```

```
x       = rand(5_000_000)

indices = x .> 0.5
y       = @view x[indices]

function foo(y)
    output = 0.0

    @simd for a in y
        output += a
    end

    return output
end
```

```
julia> @btime foo($y)
  1.878 ms (0 allocations: 0 bytes)
```

```
x       = rand(5_000_000)

indices = x .> 0.5
y       = x[indices]

function foo(y)
    output = 0.0

    for a in y
        output += a
    end

    return output
end
```

```
julia> @btime foo($y)
  1.015 ms (0 allocations: 0 bytes)
```

```
x       = rand(5_000_000)

indices = x .> 0.5
y       = x[indices]

function foo(y)
    output = 0.0

    @simd for a in y
        output += a
    end

    return output
end
```

```
julia> @btime foo($y)
  246.526 μs (0 allocations: 0 bytes)
```

Finally, comparing unit strides with non-unit strides:

```
x = rand(1_000_000)
y = @view x[1:2:length(x)]

function foo(y)
    output = 0.0

    for a in y
        output += a
    end

    return output
end
```

```
julia> @btime foo($y)
  902.479 μs (0 allocations: 0 bytes)
```

```
x = rand(1_000_000)
y = @view x[1:2:length(x)]

function foo(y)
    output = 0.0

    @simd for a in y
        output += a
    end

    return output
end
```

```
julia> @btime foo($y)
  889.059 μs (0 allocations: 0 bytes)
```

```
x = rand(1_000_000)
y = x[1:2:length(x)]

function foo(y)
    output = 0.0

    for a in y
        output += a
    end

    return output
end
```

```
julia> @btime foo($y)
  196.497 μs (0 allocations: 0 bytes)
```

```
x = rand(1_000_000)
y = x[1:2:length(x)]

function foo(y)
    output = 0.0

    @simd for a in y
        output += a
    end

    return output
end
```

```
julia> @btime foo($y)
  38.274 μs (0 allocations: 0 bytes)
```

SOME REMARKS

Note that views don't necessarily impede a sequential memory access. If the view consists of sequential elements, then we'd obtain the same performance relative to a copy created outside the function.

```
x       = rand(1_000_000)

indices = 1:length(x)
y       = @view x[indices]

function foo(y)
    output = 0.0

    @simd for a in y
        output += a
    end

    return output
end
```

```
julia> @btime foo($y)
  76.950 μs (0 allocations: 0 bytes)
```

```
x       = rand(1_000_000)

indices = 1:length(x)
y       = x[indices]

function foo(y)
    output = 0.0

    @simd for a in y
        output += a
    end

    return output
end
```

```
julia> @btime foo($y)
  76.777 μs (0 allocations: 0 bytes)
```

A corollary of this example is that views should be employed in cases like this, as they additionally avoid memory allocations.

Another remark is that *storing* elements contiguously is only a necessary condition for contiguous memory *access*. To demonstrate this, let's consider a scenario where we access vectors through the ranges.

Specifically, consider a vector `x`. Moreover, there's another vector `y` that comprises the same elements as `x`, but with zeros placed in between. Formally:

```
x_size = 1_000_000

x = rand(x_size)

y = zeros(eltype(x),x_size * 2)
    temp  = view(y, 2:2:length(y))
    temp .= x
```

```
julia> x[1:3]
3-element Vector{Float64}:
 0.906299638797481
 0.44349373245960455
 0.7456733811393941
julia> y[1:6]
6-element Vector{Float64}:
 0.0
 0.906299638797481
 0.0
 0.44349373245960455
 0.0
 0.7456733811393941
```

Given this, we'd get the same value whether we add all elements from `x` or add all elements in `y` skipping zeros. However, the results below reveal that, despite both `x` and `y` being contiguous blocks in memory, unit-stride access only holds for `x`. This explains the differences in the performance observed.

```
function foo(x)
    output = 0.0

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

    return output
end
```

```
julia> @btime foo($y)
  78.349 μs (0 allocations: 0 bytes)
```

```
function foo(y)
    output = 0.0

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

    return output
end
```

```
julia> @btime foo($y)
  188.483 μs (0 allocations: 0 bytes)
```

COPIES VS VIEWS: OVERALL EFFECTS

When slices are created, the choice between copies and views requires weighing in the overhead of additional memory allocations against the performance benefits of sequential memory accesses (including a more performant application of SIMD).

One scenario where views always outperform copies was given above, where the view elements are accessed sequentially. Instead, a common scenario where copies tend to outperform views is when we need to perform multiple operations over the same slice. In this case, the cost of an additional memory allocation is usually outweighed by the performance benefits of contiguous memory access. This is illustrated below.

```
x       = rand(5_000_000)
indices = sortperm(x)

function foo(x, indices)
    y      = @view x[indices]
    output1, output2, output3 = (0.0 for _ in 1:3)

    @simd for a in y
        output1 += a^(3/2)
        output2 += a / 3
        output3 += a * 2.5
    end

    return output1, output2, output3
end
```

```
julia> @btime foo($y)
  248.861 ms (0 allocations: 0 bytes)
```

```
x       = rand(5_000_000)
indices = sortperm(x)

function foo(x, indices)
    y      = x[indices]
    output1, output2, output3 = (0.0 for _ in 1:3)

    @simd for a in y
        output1 += a^(3/2)
        output2 += a / 3
        output3 += a * 2.5
    end

    return output1, output2, output3
end
```

```
julia> @btime foo($y)
  125.033 ms (2 allocations: 38.147 MiB)
```

In general, though, benchmarking is the only way to decide whether copies or views are faster. For instance, views are faster in the following example:

```
x       = rand(5_000_000)
indices = sortperm(x)

function foo(x, indices)
    y      = @view x[indices]
    output = 0.0

    @simd for a in y
        output += a
    end

    return output
end
```

```
julia> @btime foo($x, $indices)
  22.741 ms (0 allocations: 0 bytes)
```

```
x       = rand(5_000_000)
indices = sortperm(x)

function foo(x, indices)
    y      = x[indices]
    output = 0.0

    @simd for a in y
        output += a
    end

    return output
end
```

```
julia> @btime foo($x, $indices)
  36.151 ms (2 allocations: 38.147 MiB)
```

Instead, the following scenario establishes that an approach with copies outperforms views.

```
x       = rand(5_000_000)
indices = sortperm(x)

function foo(x, indices)
    y      = @view x[indices]
    output = 0.0

    @simd for a in y
        output += a^(3/2)
    end

    return output
end
```

```
julia> @btime foo($x, $indices)
  268.653 ms (0 allocations: 0 bytes)
```

```
x       = rand(5_000_000)
indices = sortperm(x)

function foo(x, indices)
    y      = x[indices]
    output = 0.0

    @simd for a in y
        output += a^(3/2)
    end

    return output
end
```

```
julia> @btime foo($x, $indices)
  135.816 ms (2 allocations: 38.147 MiB)
```