10e. SIMD: Contiguous Access and Unit Strides

Martin Alfaro
PhD in Economics

INTRODUCTION

This section contrasts the performance of copies and views, emphasizing that copies guarantee conditions favorable to SIMD execution.

To understand this, recall that 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 must adhere to specific organizational and access rules in memory**. The two core requirements are contiguous memory layout and unit-stride access.

- **Contiguous Memory Layout**: this means that data elements reside in adjacent memory addresses with no gaps. Freshly allocated arrays meet this requirement, enabling entire segments to load directly into vector registers. In contrast, array views don't guarantee contiguity. By referencing the original data structure, views can result in highly irregular memory access patterns that jump between non-adjacent addresses.
- **Unit Strides**: strides refer to the step size between consecutive memory accesses. Unit strides means in particular that elements are accessed sequentially. For example, iterating through a freshly allocated vector `x` using `eachindex(x)` ensures unit stride: each access moves to the next adjacent address 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).

The fact that SIMD performs best when these two conditions hold creates a key trade-off for developers. On the one hand, using views reduces memory allocations, but makes SIMD less effective as it often violates contiguity or unit stride. Instead, copies create new contiguous arrays that ensure the effectiveness of SIMD, but at the expense of increase memory usage. The current section explores this trade-off.

REVIEW OF INDEXING APPROACHES

The performance trade-offs between copies and views depend on the slicing method employed. For this reason, we begin with a brief overview of the main slicing techniques.

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

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

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

Vector Indexing works by referencing elements directly through their indices. **Ranges** are simply as a special form of vector indexing that lazily references consecutive elements according to a specified stride. The general syntax is `<first index>:<stride>:<last index>`, where omitting `<stride>`

defaults to a step size of 1.

The function `sortperm` is incorporated as it'll starkly illustrate the difference between copies and views. Given some collection `x`, `sortperm(x)` returns the corresponding indices of `sort(x)`. Assuming that `x` hasn't been ordered previously, views that access elements via the indices of `sortperm(x)` will result in an irregular memory access pattern. For instance, given `x = [5, 4, 6]`, we get that `sort(x)` returns `[4, 5, 6]` and `sortperm(x)` provides `[2, 1, 3]`. Therefore, accessing elements of `x` via `sortperm(x)` will involve jumping around within the underlying data.

For its part, **Boolean indexing** returns a Boolean vector, with `1` indicating the element must be kept. This approach is primarily employed for the creation of slices based on broadcasted conditions.

BENEFITS OF SEQUENTIAL ACCESS

In the section on reducing memory allocations, we highlighted the benefits of using views over copies when working with slices: by maintaining references to the original data, views avoid memory allocation. Yet, we briefly remarked that copies could outperform views in some scenarios. We're now in a position to explain in more depth why this occurs.

Creating a copy of some data structure involves allocating the information in a new contiguous block of memory. This layout ensures that elements are stored sequentially, thus offering two key advantages: faster element retrieval and a more effective use of SIMD instructions. Views can't guarantee this property. If the referenced elements are scattered in memory, views will necessarily introduce irregular access patterns.

An analogy may help clarify this. Imagine retrieving books from a library. If every book you need are neatly arranged 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. The analogy extends further: if you also have a cart that allows you to carry multiple books at once, the process becomes even more efficient. SIMD operations play the role of this cart.

ILLUSTRATING EACH BENEFIT IN ISOLATION

The following examples illustrate the potential advantages of copies compared to views. To ensure that the memory allocations of copies aren't distorting the results, slices will be created outside the benchmarked function,

We start with a scenario where views access elements following a pattern, but without this being characterized by unit stride.

```
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)
  884.378 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)
  905.283 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)
  196.518 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)
  41.445 µs (0 allocations: 0 bytes)
```

In some cases, the access of elements doesn't follow any predictable pattern. Such a behavior can be illustrated by accessing elements via `sortperm`.

```
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)
  21.584 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)
  21.333 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)
  2.390 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)
  1.028 ms (0 allocations: 0 bytes)
```

A similar issue occurs with 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)
  2.196 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)
  1.895 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)
  1.017 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)
  279.995 µs (0 allocations: 0 bytes)
```

COPIES VS VIEWS: TOTAL EFFECTS

The previous examples defined slices outside the benchmarked functions, thus avoiding the memory allocations of copies. The goal was to emphasize the benefits of contiguous access and unit strides in isolation. However, the choice between copies and views requires incorporating the overhead of additional memory allocations. Overall, we must weigh these costs against the performance benefits of sequential memory accesses.

In general, benchmarking is the only way to decide whether copies or views deliver better performance. For instance, below we present a case where views are preferred, since operation executed is straightforward enough to benefit from SIMD.

```
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)
  23.905 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)
  35.922 ms (2 allocations: 38.148 MiB)
```

Instead, the following scenario features a more complex but SIMD-friendly operation, illustrating how copying can actually outperform using 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)
  269.828 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)
  131.374 ms (2 allocations: 38.148 MiB)
```

SPECIAL CASES

Certain patterns allow us to predict the faster approach without exhaustive benchmarking.

One scenario where views always outperform copies is when the view is referencing sequential elements. These scenarios call for the use of view, as they provide the same benefits as copies but additionally avoiding memory allocations.

```
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)
  83.773 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)
  82.349 μs (0 allocations: 0 bytes)
```

In contrast, a common scenario where copies tend to outpace views is when we need to perform multiple operations on the same slice. In this case, the cost of an additional memory allocation is usually offset by the speed gains from 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)
  269.302 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)
  142.894 ms (2 allocations: 38.148 MiB)
```