9i. Lazy Broadcasting and Loop Fusion

[Martin Alfaro](#)
PhD in Economics

INTRODUCTION

This section continues the analysis of lazy and eager operations as a means of reducing memory allocations. The focus now shifts to broadcasting operations, which strike a balance between code readability and performance.

Central to the upcoming discussion is the eager default behavior of broadcasting in Julia. This means that broadcasted operations compute their outputs immediately upon execution, thereby inevitably leading to memory allocation when applied to allocating-objects like vectors. This characteristic is particularly relevant in scenarios involving multiple intermediate broadcasted operations, resulting in multiple temporary, and therefore avoidable, memory allocations.

Next, we present various approaches to reduce allocations with intermediate broadcasted operations. We'll start highlighting the notion of **loop fusion**, which allows multiple broadcasting operations to be combined into a more efficient single operation. After this, we'll explore the `LazyArrays` package, which evaluates broadcasting operations in a lazy manner. The technique is particularly useful for reductions requiring intermediate transformations, entirely circumventing memory allocations for these intermediates.

HOW DOES BROADCASTING WORK?

To gain a deeper understanding of the optimizations we'll be discussing, let's first examine the internal mechanics of broadcasting. Under the hood, broadcasting operations are converted into optimized for-loops during compilation, rendering the two approaches computationally equivalent. In this context, broadcasting serves as syntactic sugar, eliminating the need for explicit for-loops. This allows developers to write more concise and expressive code, without compromising performance.

Despite the equivalence between broadcasting and for-loops, you'll often notice performance differences in practice. These discrepancies are largely driven by compiler optimizations, rather than inherent differences between the two approaches. Essentially, the fact that an operation supports a broadcasted form reveals further information about its underlying structure, allowing the compiler to automatically apply certain optimizations. In contrast, the generality of for-loops precludes this possibility, as the same assumptions can't be taken for granted. Nevertheless, with careful manual optimization, for-loops can always match or surpass the performance of broadcasting, thanks to their greater flexibility and potential for fine-tuning.

The following code snippets demonstrate the equivalence between the two approaches. The first tab describes the operation being performed, while the second tab provides a rough translation of broadcasting's internal implementation. The third tab further highlights this equivalence, by providing the exact code used. This requires including the `@inbounds` macro in the for-loop, which is automatically applied with broadcasting. The specific effect of this macro on computations can be disregarded, as it'll be discussed in a later section. Its sole purpose here is to illustrate the equivalence between the two approaches.

```
x      = rand(100)

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

```
julia> @btime foo($x)
  34.506 ns (1 allocation: 896 bytes)
```

```
x      = rand(100)

function foo(x)
    output = similar(x)

    for i in eachindex(x)
        output[i] = 2 * x[i]
    end

    return output
end
```

```
julia> @btime foo($x)
  48.188 ns (1 allocation: 896 bytes)
```

```
x      = rand(100)

function foo(x)
    output = similar(x)

    @inbounds for i in eachindex(x)
        output[i] = 2 * x[i]
    end

    return output
end
```

```
julia> @btime foo($x)
  32.832 ns (1 allocation: 896 bytes)
```

> **Warning!** - About `@inbounds`
>
> In the example provided, `@inbounds` **was added to illustrate the internal implementation of broadcasting, rather than a recommended practice for general use**. In fact, `@inbounds` used incorrectly **can lead to severe issues**.

> To understand what this macro does, Julia by default enforces bounds checking on array indices to prevent out-of-range access (e.g., it checks that a vector `x` with 3 elements isn't accessed at index `x[4]`). Adding `@inbounds` to a for-loop instructs Julia to bypass this check, thus speeding up computations. However, it simultaneously makes our code unsafe, including the possibility of returning incorrect results and other more pronounced issues.

A key implication of the example is that Julia's broadcasting is eager by default, meaning that the result is immediately computed and stored. In the example, this is reflected in `2 .* x` being computed and stored in `output`, which also explains the observed memory allocation.

Importantly, **memory allocations under broadcasting arise even if the result isn't explicitly stored**. For example, computing `sum(2 .* x)` involves the computation and temporary storage of `2 .* x`, before the sum is performed.

REMARK (OPTIONAL) Differing Optimizations With Broadcasting and For-Loops

BROADCASTING: LOOP FUSION

While eager broadcasting makes results readily available, their outputs may not be important in themselves. Instead, they could represent intermediate steps in a larger computation, with these outputs eventually being passed as inputs to subsequent operations.

In the following, we address scenarios like this, where broadcasting is employed for intermediate results. The first approach we explore leverages a technique called **loop fusion**, which combines multiple broadcasting operations into a single loop. By doing so, the compiler can perform all operations in a single pass over the data. This not only eliminates the creation of multiple intermediate vectors, but also provides the compiler with a holistic view of the operations, thus allowing for further optimizations.

Remarkably, when all broadcasting operations are nested within a single operation, the compiler automatically implements loop fusion. However, for complex expressions, writing a single lengthy expression can be impractical. To overcome this limitation, we'll show a method that enables us to break down operations into partial calculations, while still preserving loop fusion. This approach is based on the lazy design of functions definitions, making it possible to delay operations until their eventual combination.

```
x       = rand(100)

foo(x)  = x .* 2 .+ x .* 3      # or @. x * 2 + x * 3
```

```
julia> @btime foo($x)
  35.361 ns (1 allocation: 896 bytes)
```

```
x        = rand(100)

function foo(x)
    a      = x .* 2
    b      = x .* 3

    output = a .+ b
end
```

```
julia> @btime foo($x)
  124.420 ns (3 allocations: 2.62 KiB)
```

```
x        = rand(100)

term1(a) = a * 2
term2(a) = a * 3

foo(a)   = term1(a) + term2(a)
```

```
julia> @btime foo.($x)
  34.330 ns (1 allocation: 896 bytes)
```

VECTOR OPERATIONS ALLOCATE AND BREAK LOOP FUSION

A common scenario where loop fusion is prevented is when a single expression combines broadcasting and vector operations. This possibility arises because **some vector operations yield similar results to their broadcasting equivalents**, making it possible to combine these operations without dimensional mismatches. For instance, we show below that adding two vectors using `+` produces the same result as summing them element-wise by employing `.+`.

```
x        = [1, 2, 3]
y        = [4, 5, 6]

foo(x,y) = x .+ y
```

```
julia> foo(x,β)
3-element Vector{Int64}:
 5
 7
 9
```

```
x        = [1, 2, 3]
y        = [4, 5, 6]

foo(x,y) = x + y
```

```
julia> foo(x,β)
3-element Vector{Int64}:
 5
 7
 9
```

The same occurs with a vector product when one of the operands is a scalar.

```
x        = [1, 2, 3]
β        = 2

foo(x,β) = x .* β
```

```
julia> foo(x,β)
3-element Vector{Int64}:
 2
 4
 6
```

```
x        = [1, 2, 3]
β        = 2

foo(x,β) = x * β
```

```
julia> foo(x,β)
3-element Vector{Int64}:
 2
 4
 6
```

OMITTING DOTS AVOIDS LOOP FUSION

Mixing vector operations and broadcasting is problematic for performance. The reason is that each vector operation will allocate memory, in a context that operations aren't fused. In particular, the issue arises when the following conditions are met:

- The final output requires combining multiple operations
- The operations yield the same result whether implemented through broadcasting or a vector operation
- We mix broadcasting and vector implementations, by omitting the inclusion of some dots

If all these conditions are satisfied, Julia will partition the operations and compute each separately. The consequence of this is the emergence of multiple temporary vectors, with each separately allocating memory.

The following example illustrates this possibility in the extreme case where *all* broadcasting dots `.` are omitted. It demonstrates that vector operations aren't fused, even when expressed in a single operation. Moreover, it establishes that vector operations are similar to obtaining the final result by separately calculating each operation.

```
x = rand(100)

foo(x)  = x * 2 + x * 3
```

```
julia> @btime foo($x)
  129.269 ns (3 allocations: 2.62 KiB)
```

```
x = rand(100)

function foo(x)
    term1  = x * 2
    term2  = x * 3

    output = term1 + term2
end
```

```
julia> @btime foo($x)
  130.798 ns (3 allocations: 2.62 KiB)
```

While the previous example exclusively consists of vector operations, the same principle applies when mixing broadcasting and non-broadcasting operations. In such cases, loop fusion is partially achieved, with only a subset of operations being internally computed through a single for-loop.

```
x      = rand(100)

foo(x) = x * 2 .+ x .* 3
```

```
julia> @btime foo($x)
  85.034 ns (2 allocations: 1.75 KiB)
```

```
x      = rand(100)

function foo(x)
    term1  = x * 2

    output = term1 .+ x .*3
end
```

```
julia> @btime foo($x)
  85.763 ns (2 allocations: 1.75 KiB)
```

Overall, the key takeaway from these examples is that **guaranteeing loop fusion requires appending a dot to every operator and function to be broadcasted**. Note that this can be error-prone, especially in large expressions where a single missing dot can be easily overlooked.

Fortunately, there are two alternatives that mitigate this risk.

One option is to prefix the expression with `@.`, as shown in the tab `Equivalent 1` below. This ensures that *all* operators and functions are broadcasted. Alternatively, all operations could be combined into a *scalar* function, which you eventually broadcast. This is presented in the tab `Equivalent 2` below.

```
x      = rand(100)

foo(x) = x .* 2 .+ x .* 3
```

```
julia> @btime foo($x)
  36.456 ns (1 allocation: 896 bytes)
```

```
x      = rand(100)

foo(x) = @. x * 2 + x * 3
```

```
julia> @btime foo($x)
  36.573 ns (1 allocation: 896 bytes)
```

```
x      = rand(100)

foo(a) = a * 2 + a * 3
```

```
julia> @btime foo.($x)
  34.536 ns (1 allocation: 896 bytes)
```

When multiple long operations are combined, the need to split operations is inevitable. In this case, we can apply a similar trick as we did before, where we leverage that function definitions are inherently lazy. Specifically, this allows us to achieve loop fusion by defining each operation as a separate scalar function.

▼ Loop Fusion Splitting Operations

```
x        = rand(100)

term1(a) = a * 2
term2(a) = a * 3

foo(a)   = term1(a) + term2(a)
```

```
julia> @btime foo.($x)
  35.346 ns (1 allocation: 896 bytes)
```

LAZY BROADCASTING

To handle intermediate computations, we can also transform broadcasting into a lazy operation. Such functionality is provided by the `LazyArrays` package, whose use requires prepending the `@~` macro to the broadcasting operation. Similar to a function definition, lazy broadcasting defers the actual computation of the operation until it's needed.

```
x         = rand(100)

function foo(x)
    term1  = x .* 2
    term2  = x .* 3

    output = term1 .+ term2
end
```

```
julia> @btime foo($x,$y)
  109.803 ns (3 allocations: 2.62 KiB)
```

```
x         = rand(100)

function foo(x)
    term1  = @~ x .* 2
    term2  = @~ x .* 3

    output = term1 .+ term2
end
```

```
julia> @btime foo($x,$y)
  37.304 ns (1 allocation: 896 bytes)
```

Note that, up to this point, all the cases considered had a vector as its end result. In this context, our goal becomes to reduce memory allocations to the single unavoidable allocation, which is given by storage of the final result.

Instead, when scalar values are the final output, lazy broadcasting offers a significant advantage: it enables us to completely remove memory allocations. This is because lazy broadcasting fuses broadcasting and reduction operations, allowing the former to be computed on-the-fly. This is illustrated in the example provided below.

```
# eager broadcasting (default)
x      = rand(100)

foo(x) = sum(2 .* x)
```

```
julia> @btime foo($x,$y)
  48.012 ns (1 allocation: 896 bytes)
```

```
using LazyArrays
x      = rand(100)

foo(x) = sum(@~ 2 .* x)
```

```
julia> @btime foo($x,$y)
  7.906 ns (0 allocations: 0 bytes)
```

Note that completely eliminating allocations can't be accomplished simply by using functions. This is because functions enable the splitting of broadcasting operations, but do not fuse them with reduction operations.

```
x = rand(100)

term1(a) = a * 2
term2(a) = a * 3
temp(a)  = term1(a) + term2(a)

foo(x)   = sum(temp.(x))
```

```
julia> @btime foo($x,$y)
  48.307 ns (1 allocation: 896 bytes)
```

```
x = rand(100)

term1(a) = a * 2
term2(a) = a * 3
temp(a)  = term1(a) + term2(a)

foo(x)   = sum(@~ temp.(x))
```

```
julia> @btime foo($x,$y)
  10.474 ns (0 allocations: 0 bytes)
```

```
x = rand(100)

function foo(x)
    term1  = @~ x .* 2
    term2  = @~ x .* 3
    temp   = @~ term1 .+ term2

    output = sum(temp)
end
```

```
julia> @btime foo($x,$y)
  13.766 ns (0 allocations: 0 bytes)
```

> **Remark**
>
> An additional advantage of `@~` is that it performs additional optimizations when possible. As a result, you'll typically observe that

`@~` is faster than alternatives like a lazy map, despite that neither allocates memory. This performance benefit can be appreciated in the following comparison.

```
x = rand(100)

term1(a) = a * 2
term2(a) = a * 3
temp(a)  = term1(a) + term2(a)

foo(x)   = sum(@~ temp.(x))
```

```
julia> @btime foo($x,$y)
  10.474 ns (0 allocations: 0 bytes)
```

```
x = rand(100)

term1(a) = a * 2
term2(a) = a * 3
temp(a)  = term1(a) + term2(a)

foo(x)   = sum(Iterators.map(temp, x))
```

```
julia> @btime foo($x,$y)
  28.909 ns (0 allocations: 0 bytes)
```