

9g. 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.

A key feature of broadcasting is its eager default behavior. This means that broadcasted operations compute and materialize outputs immediately upon execution. Thus, it inevitably leads to memory allocation when applied to objects such as vectors. Such behavior becomes especially relevant in scenarios with intermediate broadcasted operations, whose allocations are potentially avoidable.

To address the associated performance cost, we'll present two strategies. The first one highlights the notion of **loop fusion**. This enables the combination of multiple broadcasting operations into a single more efficient one. Its relevance lies in

minimizing memory allocations, rather than their entire elimination. After this, we'll explore the `LazyArrays` package, which evaluates broadcasting operations [lazily](#). When reductions require intermediate calculations, this technique can completely bypass memory allocations.

HOW DOES BROADCASTING WORK INTERNALLY?

Under the hood, broadcasting operations are converted into optimized for-loops. Indeed, broadcasting is essentially syntactic sugar for for-loops, sparing developers from writing them explicitly. This makes it possible to write code that's concise and expressive, without sacrificing performance.

Despite this, you'll often notice performance differences in practice. These discrepancies stem primarily from compiler optimizations, rather than inherent differences between broadcasting and for-loops. The reason is that an operation supporting a broadcasted form is automatically revealing additional information about its underlying structure. Consequently, the compiler is allowed to apply more aggressive optimizations. In contrast, for-loops are conceived as a general-purpose construct, precluding the compiler from automatically making such assumptions.

It's worth noting, though, that carefully optimized for-loops always match or exceed the performance of broadcasting. The following code snippets demonstrate this point. The first tab outlines the operation being performed, while the second tab provides a rough internal translation of broadcasting into a for-loop.

The third tab demonstrates the equivalence more directly, using a for-loop that resembles the broadcasted code more closely. Its implementation adds the `@inbounds` macro, which broadcasting always applies automatically. The definition of `@inbounds` will be studied in [a later section](#). Its inclusion here is only to illustrate the internal implementation of broadcasting.

```
x = rand(100)
```

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

```
julia> @btime foo($x)  
25.632 ns (2 allocations: 928 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)
25.771 ns (2 allocations: 928 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)
28.144 ns (2 allocations: 928 bytes)
```

Warning! - About `@inbounds`

In the example provided, `@inbounds` was solely added to demonstrate the internal implementation of broadcasting, not as a general recommended practice. In fact, `@inbounds` can cause serious issues when used incorrectly.

To understand the role of this macro, recall that Julia performs bounds checks in for-loops by default. This means Julia verifies the existence of elements accessed during every iteration. Bounds checking prevents out-of-range access, so that a vector `x` with 3 elements isn't accessed at index `x[4]`. Instead, placing `@inbounds` at the beginning of a for-loop instructs Julia to disable these checks.

Removing bounds checking can improve performance, but it comes at the cost of safety: an out-of-range access may result in program termination or trigger more severe issues. Out-of-bounds access can't occur when operations support broadcasting, since the broadcast mechanism guarantees a valid index range. Consequently, Julia safely omits these checks by automatically applying `@inbounds`.

REMARK (OPTIONAL) Other Optimization Differences Between Broadcasting and For-Loops

×

REMARK (OPTIONAL) OTHER OPTIMIZATION DIFFERENCES BETWEEN

BROADCASTING AND FOR-LOOPS

We shouldn't infer from the previous example that the compiler disregards the structure of for-loops, thereby missing opportunities for further optimizations. In fact, the compiler may automatically insert `@inbounds` in for-loops when it's safe to do so. This typically occurs in simple cases, such as when only `x` is indexed and the iteration range is given by `eachindex(x)`. In the previous example, `@inbounds` wasn't automatically applied because `output` was also being iterated.

```
x = rand(100)

function foo(x)
    output = zero(eltype(x))

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

    return output
end
```

```
julia> @btime foo($x)
25.388 ns (0 allocations: 0 bytes)
```

```

@inbounds for i in eachindex(x)
    x[i] = rand(100)
end

function foo(x)
    output = zero(eltype(x))

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

    return output
end

# iteration, unless explicitly instructed otherwise. A
typical example is when normalizing the values of x, so that
their sum equals unity. Broadcasting will then compute
sum(x) only once, while a for-loop will recompute it at every
iteration.

```

`julia> @btime foo($x);`
 24.878 ns (0 allocations: 0 bytes)

```
x = rand(100)

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

```
julia> @btime foo($x)
41.568 ns (2 allocations: 928 bytes)
```

**CO
IN**

```
x = rand(100)

function foo(x)
    output = similar(x)
    denominator = sum(x)

    @inbounds for i in eachindex(x)
        output[i] = x[i] / denominator
    end

    return output
end
```

Importantly, **memory allocations in broadcasting arise even when the result isn't explicitly stored**. For example,

```
julia> @btime foo($x)
42.270 ns (2 allocations: 928 bytes)
```



```

evaluating sum(x) * y involves storing internally the
x = rand(100)
inte

function foo(x)
Seco    output = similar(x)
with
x)
stra    for i in eachindex(x)
cor      output[i] = x[i] / sum(x)
julia    end
part
inte    return output
a trans end
a transformed version of x. This avoids materializing the
intermediate result altogether, thereby eliminating the
julia> @btime foo($x)
592.896 ns (2 allocations: 928 bytes)

```

LOOP FUSION

When working with long broadcasted operations, splitting them into smaller intermediate steps can significantly improve readability and reduce the likelihood of errors. However, this approach comes at the cost of separately allocating each broadcasting operation.

To mitigate unnecessary memory allocations, it's essential to preserve **loop fusion**: a compiler optimization that merges multiple element-wise operations into a single loop over the data. Thus, with loop fusion enabled, 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 unlocking further optimizations.

Loop fusion is applied automatically when all operations are expressed as a single broadcasted operation. Yet, as indicated before, writing a single lengthy monolithic expression is inconvenient for complex computations. To overcome this limitation, we can rely on the lazy design of function definitions. Below, we show how this approach can break down an operation into partial calculations, while still preserving loop fusion.

```
x = rand(100)
```

```
function foo(x)
```

```
    a = x .* 2
```

```
    b = x .* 3
```

```
    output = a .+ b
```

```
end
```

```
julia> @btime foo($x)
```

```
91.053 ns (6 allocations: 2.719 KiB)
```

```
x          = rand(100)

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

```
julia> @btime foo($x)
29.803 ns (2 allocations: 928 bytes)
```

```
x          = rand(100)

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

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

```
julia> @btime foo.($x)
28.252 ns (2 allocations: 928 bytes)
```

Even with a single simple operation, certain coding patterns can inadvertently break loop fusion. When this occurs, Julia will internally fall back to evaluating sub-expressions in separate for-loops, which are eventually combined to present the final output. In the following, we present two of these cases. The key practical takeaway is that you should broadcast via the macro `@.` when possible, rather than manually adding dots to each operator and function within an expression.

MIXING BROADCASTING WITH VECTOR

OPERATIONS BREAKS LOOP FUSION

To understand the issue, there are two facts to consider. The first one is that some vector operations produce an equivalent result to their broadcasted counterparts. For instance, adding two vectors with `+` yields the same result as summing them element-wise with `.+`.

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

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

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

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

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

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

Another example is with product operations.

```
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
```

The second fact to consider is that vector operations don't participate in loop fusion. Thus, even if all these operations were combined into a single expression, Julia will evaluate each sub-expression separately, allocating temporary vectors at every step.

```
x = rand(100)
```

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

```
julia> @btime foo($x)
99.169 ns (6 allocations: 2.719 KiB)
```

```
x = rand(100)
```

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

    output = term1 + term2
end
```

```
julia> @btime foo($x)
101.228 ns (6 allocations: 2.719 KiB)
```

Putting both facts together, mixing broadcasting with vector operations in the same expression may yield the correct result, but only achieve loop fusion partially. This means Julia will only fuse contiguous broadcasted segments, internally partitioning the computation into separate for-loops when a vector operation is encountered. Each of these for-loops will produce a temporary intermediate vector.

```
x = rand(100)

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

```
julia> @btime foo($x)
70.756 ns (4 allocations: 1.812 KiB)
```

```
x = rand(100)

function foo(x)
    term1 = x * 2

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

```
julia> @btime foo($x)
65.390 ns (4 allocations: 1.812 KiB)
```

This behavior leads to a clear and actionable guideline: for full loop fusion, every operator and function in the expression must be explicitly broadcasted. Yet manually adding dots throughout a large expression is both tedious and error-prone: one missing dot is enough to reintroduce unnecessary allocations. There are two coding practices that mitigate this risk.

One option is to broadcast via the macro `@.`, as shown below in the tab "Equivalent 1". By design, this adds a dot to *all* operators and functions within an expression, guaranteeing loop fusion. An alternative is to express the operation through

a *scalar* function, which we then broadcast. This is presented below in the tab "Equivalent 2".

```
x      = rand(100)

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

julia> @btime foo($x)
28.308 ns (2 allocations: 928 bytes)
```

```
x      = rand(100)

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

julia> @btime foo($x)
28.862 ns (2 allocations: 928 bytes)
```

```
x      = rand(100)

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

julia> @btime foo.($x)
28.100 ns (2 allocations: 928 bytes)
```

LAZY BROADCASTING

Broadcasting results in memory allocations since the technique is eager by default. This property means that its

output becomes readily available when a broadcasted expression is evaluated. Considering this, another way to achieve loop fusion is by evaluating all broadcasted sub-expressions lazily, until the final output is computed. The approach is similar in spirit to the use of helper functions to accomplish the same goal, but without cluttering the namespace with new functions.

The functionality is provided by the macro `@~` from the `LazyArrays` package. By prepending this macro to the broadcasting operation, its computation is deferred until its output is required.

```
x = rand(100)

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

    output = term1 .+ term2
end
```

```
julia> @btime foo($x)
95.868 ns (6 allocations: 2.719 KiB)
```

```

x          = rand(100)

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

    output = term1 .+ term2
end

```

```

julia> @btime foo($x)
28.945 ns (2 allocations: 928 bytes)

```

```

x          = rand(100)

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

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

```

```

julia> @btime foo($x)
27.621 ns (2 allocations: 928 bytes)

```

Beyond this resemblance with an approach based on functions, lazy broadcasting additionally addresses certain scenarios that functions can't handle. To understand this, note that the operations explored thus far resulted in a *vector* output. In those cases, memory allocations could at best be reduced to a single unavoidable allocation, necessary for storing the final output.

When the final output is instead given by a scalar, as occurs with reductions, lazy broadcasting is capable of entirely eliminating memory allocations. The reason is that lazy broadcasting fuses with reduction operations, letting the compiler apply a [non-allocating_procedure](#). This is illustrated in the example below.

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

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

```
julia> @btime foo($x)
41.702 ns (2 allocations: 928 bytes)
```

```
using LazyArrays
x      = rand(100)

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

```
julia> @btime foo($x)
7.456 ns (0 allocations: 0 bytes)
```

Lazy Broadcasting May Be Faster Than Other Lazy Alternatives

An additional advantage of `@~` is that it implements extra optimizations. This explains why `@~` tends to be faster than alternatives like a lazy map, even though neither allocates memory. This performance benefit can be noticed 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)
```

```
10.244 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)
```

```
29.894 ns (0 allocations: 0 bytes)
```