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.

A key aspect of broadcasting operations in Julia is their eager default behavior. This means that broadcasted operations compute their outputs immediately upon execution, inevitably leading to memory allocation when applied to objects such as vectors. This feature becomes especially relevant in scenarios with intermediate broadcasted operations, which result in multiple potentially avoidable allocations.

To mitigate this issue, we'll present various strategies for reducing allocations with intermediate broadcasted operations. One approach highlights the notion of **loop fusion**, which allows multiple broadcasting operations to be combined into a single more efficient operation. After this, we'll explore the `LazyArrays` package, which evaluates broadcasting operations in a lazy manner.

HOW DOES BROADCASTING WORK?

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. Essentially, broadcasting serves as syntactic sugar, eliminating the need for explicit for-loops. This allows users to write more concise and expressive code, without compromising performance.

Despite this equivalence, you'll often notice performance differences in practice. These discrepancies are largely driven by compiler optimizations, rather than inherent differences between the two approaches. The reason for this is that an operation supporting a broadcasted form reveals further information to the compiler about its underlying structure. In this way, the compiler is allowed to apply further optimizations. In contrast, for-loops are conceived as a more general construct, so that these additional assumptions shouldn't be taken for granted.

Note, though, that with careful manual optimization, for-loops can always match or surpass the performance of broadcasting. The following code snippets demonstrate this point.

The first tab describes the operation being performed, while the second tab provides a rough translation of broadcasting's internal implementation. The third tab demonstrates the equivalence by writing a for-loop mirroring the exact code used in broadcasting. This is achieved by adding the `@inbounds` macro, which is automatically applied with broadcasting. The specific role of `@inbounds` will be discussed in a later section. Its sole inclusion here is to illustrate the equivalence between the two approaches once we account for it.

```
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, not as a general recommended practice**. In fact, used incorrectly, `@inbounds` **can cause serious issues**.

> To understand what this macro does, Julia by default enforces bounds checks on array indices. For instance, it checks that a vector `x` with 3 elements isn't accessed at index `x[4]`. In this way, out-of-range access is prevented. Placing `@inbounds` on a for-loop instructs Julia to disable these checks to improve performance. That speed gain, though, comes at the cost of safety: an out-of-range access can silently produce incorrect results or lead to other issues.

A key implication of the example is that Julia's broadcasting is eager by default. In the example, this means that `2 .* x` is immediately computed and then stored in `output`. This 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 internal computation and temporary storage of `2 .* x`.

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 by themselves. Instead, they could represent intermediate steps in a larger computation, eventually passed as inputs to subsequent operations.

In the following, we address scenarios like this, where broadcasting is employed for intermediate results. The first approach 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 enabling further optimizations.

When all broadcasting operations are nested within a single operation, the compiler automatically implements loop fusion. For complex expressions, however, writing a single lengthy expression can be impractical. To overcome this limitation, we'll show how to break down an operation into partial calculations, while still preserving loop fusion. The method relieson the lazy design of functions definitions, allowing operations to be delayed until their final 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 situation that prevents loop fusion is when a single expression mixes broadcasting with vector operations. This occurs because **some vector operations produce the same results as their broadcasting equivalents**, thus precluding dimensional mismatches. 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,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 issue arises with vector-scalar multiplication. When one operand is a scalar, the vector product produces the same element-wise result as its broadcasting form.

```
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, since prevents loop fusion and forces memory allocations. Going beyond the examples presented before, you can identify this issue when when the following conditions are met:

- The final output requires combining multiple operations
- Broadcasting and vector operations would yield the same result
- Broadcasting and vector operations are effectively mixed, due to the omission of some broadcasting dots

If those conditions hold, Julia partitions the work and computes each part separately, producing multiple temporary vectors and extra allocations.

The following example illustrates this possibility in the extreme case where *all* broadcasting dots `.` are omitted. The example demonstrates that, since vector operations aren't fused, the final output is obtained by separately calculating each intermediate 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 we broadcast some operations and not others. 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 the macro `@.`, as shown below in tab "Equivalent 1". This ensures that *all* operators and functions are broadcasted. An alternative solution is to combine all operations into a *scalar* function, which you eventually broadcast. This is presented below in the tab "Equivalent 2".

```
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 several lengthy operations must be combined, the need to split them becomes inevitable. In such cases, we can still leverage the inherent laziness of function definitions. Loop fusion would then be achieved 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 efficiently, another possibility is to transform broadcasting into a lazy operation. Similar to a function definition, lazy broadcasting defers computations until their results are explicitly required.

The functionality is provided by the `LazyArrays` package, whose use requires prepending the `@~` macro to the broadcasting operation.

```
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)
```

All cases considered thus far have 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 enables the complete elimination of memory allocations. This is achieved because nesting lazy broadcasted operations implements loop fusion, as illustrated in the example 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 simply using functions isn't sufficient to completely eliminate allocations. Functions allow for the splitting of broadcasting operations, but don't 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 the extra optimizations that implements when possible. As a result, `@~` tends to be 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)
```