Multidimensional algorithms and iteration

01 Feb 2016 | Tim Holy

Starting with release 0.4, Julia makes it easy to write elegant and efficient multidimensional algorithms. The new capabilities rest on two foundations: a new type of iterator, called `CartesianRange`, and sophisticated array indexing mechanisms. Before I explain, let me emphasize that developing these capabilities was a collaborative effort, with the bulk of the work done by Matt Bauman (@mbauman), Jutho Haegeman (@Jutho), and myself (@timholy).

These new iterators are deceptively simple, so much so that I've never been entirely convinced that this blog post is necessary: once you learn a few principles, there's almost nothing to it. However, like many simple concepts, the implications can take a while to sink in. There also seems to be some widespread confusion about the relationship between these iterators and `Base.Cartesian`, which is a completely different (and much more painful) approach to solving the same problem. There are still a few occasions where `Base.Cartesian` is necessary, but for many problems these new capabilities represent a vastly simplified approach.

Let's introduce these iterators with an extension of an example taken from the manual.

eachindex, CartesianIndex, and CartesianRange

You may already know that, in julia 0.4, there are two recommended ways to iterate over the elements in an `AbstractArray`: if you don't need an index associated with each element, then you can use

```
for a in A    # A is an AbstractArray
    # Code that does something with the element a
end
```

If instead you also need the index, then use

```
for i in eachindex(A)
    # Code that does something with i and/or A[i]
end
```

In some cases, the first line of this loop expands to `for i = 1:length(A)`, and `i` is just an integer. However, in other cases, this will expand to the equivalent of

```
for i in CartesianRange(size(A))
    # i is now a CartesianIndex
    # Code that does something with i and/or A[i]
end
```

Let's see what these objects are:

```
julia> A = rand(3,2)

julia> for i in CartesianRange(size(A))
           @show i
       end
i = CartesianIndex{2}((1,1))
i = CartesianIndex{2}((2,1))
i = CartesianIndex{2}((3,1))
i = CartesianIndex{2}((1,2))
i = CartesianIndex{2}((2,2))
i = CartesianIndex{2}((3,2))
```

A `CartesianIndex{N}` represents an `N`-dimensional index. `CartesianIndex`es are based on tuples, and indeed you can access the underlying tuple with `i.I`. However, they also support certain arithmetic operations, treating their contents like a fixed-size `Vector{Int}`. Since the length is fixed, julia/LLVM can generate very efficient code (without introducing loops) for operations with N-dimensional `CartesianIndex`es.

A `CartesianRange` is just a pair of `CartesianIndex`es, encoding the start and stop values along each dimension, respectively:

```
julia> CartesianRange(size(A))
CartesianRange{CartesianIndex{2}}(CartesianIndex{2}((1,1)),CartesianIndex{2}((3,2)))
```

You can construct these manually: for example,

```
julia> CartesianRange(CartesianIndex((-7,0)), CartesianIndex((7,15)))
CartesianRange{CartesianIndex{2}}(CartesianIndex{2}((-7,0)),CartesianIndex{2}((7,15)))
```

constructs a range that will loop over `-7:7` along the first dimension and `0:15` along the second.

One reason that `eachindex` is recommended over `for i = 1:length(A)` is that some `AbstractArray`s cannot be indexed efficiently with a linear index; in contrast, a much wider class of objects can be efficiently indexed with a multidimensional iterator. (SubArrays are, generally speaking, a prime example.) `eachindex` is designed to pick the most efficient iterator for the given array type. You can even use

```
for i in eachindex(A, B)
    ...
```

to increase the likelihood that `i` will be efficient for accessing both `A` and `B`.

As we'll see below, these iterators have another purpose: independent of whether the underlying arrays have efficient linear indexing, multidimensional iteration can be a powerful ally when writing algorithms. The rest of this blog post will focus on this latter application.

Writing multidimensional algorithms with CartesianIndex iterators

A multidimensional boxcar filter

Let's suppose we have a multidimensional array `A`, and we want to compute the "moving average" over a 3-by-3-by-... block around each element. From any given index position, we'll want to sum over a region offset by `-1:1` along each dimension. Edge positions have to be treated specially, of course, to avoid going beyond the bounds of the array.

In many languages, writing a general (N-dimensional) implementation of this conceptually-simple algorithm is somewhat painful, but in Julia it's a piece of cake:

```
function boxcar3(A::AbstractArray)
    out = similar(A)
    R = CartesianRange(size(A))
    I1, Iend = first(R), last(R)
    for I in R
        n, s = 0, zero(eltype(out))
        for J in CartesianRange(max(I1, I-I1), min(Iend, I+I1))
            s += A[J]
            n += 1
        end
        out[I] = s/n
    end
    out
end
```

Let's walk through this line by line:

- `out = similar(A)` allocates the output. In a "real" implementation, you'd want to be a little more careful about the element type of the output (what if the input array element type is `Int`?), but we're cutting a few corners here for simplicity.
- `R = CartesianRange(size(A))` creates the iterator for the array, ranging from `CartesianIndex((1, 1, 1, ...))` to `CartesianIndex((size(A,1), size(A,2), size(A,3), ...))`. We don't use `eachindex`, because we can't be sure whether that will return a `CartesianRange` iterator, and here we explicitly need one.
- `I1 = first(R)` and `Iend = last(R)` return the lower (`CartesianIndex((1, 1, 1, ...))`) and upper (`CartesianIndex((size(A,1), size(A,2), size(A,3), ...))`) bounds of the iteration range, respectively. We'll use these to ensure that we never access out-of-bounds elements of `A`.

Conveniently, `I1` can also be used to compute the offset range.

- `for I in R`: here we loop over each entry of `A`.
- `n = 0` and `s = zero(eltype(out))` initialize the accumulators. `s` will hold the sum of neighboring values. `n` will hold the number of neighbors used; in most cases, after the loop we'll have `n == 3^N`, but for edge points the number of valid neighbors will be smaller.
- `for J in CartesianRange(max(I1, I-I1), min(Iend, I+I1))` is probably the most "clever" line in the algorithm. `I-I1` is a `CartesianIndex` that is lower by 1 along each dimension, and `I+I1` is higher by 1. Therefore, this constructs a range that, for interior points, extends along each coordinate by an offset of 1 in either direction along each dimension.

  However, when `I` represents an edge point, either `I-I1` or `I+I1` (or both) might be out-of-bounds. `max(I-I1, I1)` ensures that each coordinate of `J` is 1 or larger, while `min(I+I1, Iend)` ensures that `J[d] <= size(A,d)`.
- The inner loop accumulates the sum in `s` and the number of visited neighbors in `n`.
- Finally, we store the average value in `out[I]`.

Not only is this implementation simple, but it is surprisingly robust: for edge points it computes the average of whatever nearest-neighbors it has available. It even works if `size(A, d) < 3` for some dimension `d`; we don't need any error checking on the size of `A`.

Computing a reduction

For a second example, consider the implementation of multidimensional *reductions*. A reduction takes an input array, and returns an array (or scalar) of smaller size. A classic example would be summing along particular dimensions of an array: given a three-dimensional array, you might want to compute the sum along dimension 2, leaving dimensions 1 and 3 intact.

The core algorithm

An efficient way to write this algorithm requires that the output array, `B`, is pre-allocated by the caller (later we'll see how one might go about allocating `B` programmatically). For example, if the input `A` is of size `(l,m,n)`, then when summing along just dimension 2 the output `B` would have size `(l,1,n)`.

Given this setup, the implementation is shockingly simple:

```
function sumalongdims!(B, A)
    # It's assumed that B has size 1 along any dimension that we're summing
    fill!(B, 0)
    Bmax = CartesianIndex(size(B))
    for I in CartesianRange(size(A))
        B[min(Bmax,I)] += A[I]
    end
    B
end
```

The key idea behind this algorithm is encapsulated in the single statement `B[min(Bmax,I)]`. For our three-dimensional example where `A` is of size `(l,m,n)` and `B` is of size `(l,1,n)`, the inner loop is essentially equivalent to

```
B[i,1,k] += A[i,j,k]
```

because `min(1,j) = 1`.

The wrapper, and handling type-instability using function barriers

As a user, you might prefer an interface more like `sumalongdims(A, dims)` where `dims` specifies the dimensions you want to sum along. `dims` might be a single integer, like `2` in our example above, or (should you want to sum along multiple dimensions at once) a tuple or `Vector{Int}`. This is indeed the interface used in `sum(A, dims)`; here we want to write our own (somewhat simpler) implementation.

A bare-bones implementation of the wrapper is straightforward:

```
function sumalongdims(A, dims)
    sz = [size(A)...]
    sz[[dims...]] = 1
    B = Array(eltype(A), sz...)
    sumalongdims!(B, A)
end
```

Obviously, this simple implementation skips all relevant error checking. However, here the main point I wish to explore is that the allocation of `B` turns out to be type-unstable: `sz` is a `Vector{Int}`, the length (number of elements) of a specific `Vector{Int}` is not encoded by the type itself, and therefore the dimensionality of `B` cannot be inferred.

Now, we could fix that in several ways, for example by annotating the result:

```
B = Array(eltype(A), sz...)::typeof(A)
```

However, this isn't really necessary: in the remainder of this function, `B` is not used for any performance-critical operations. `B` simply gets passed to `sumalongdims!`, and it's the job of the compiler to ensure that, given the type of `B`, an efficient version of `sumalongdims!` gets generated. In other words, the type instability of `B`'s allocation is prevented from "spreading" by the fact that `B` is henceforth used only as an argument in a function call. This trick, using a function-call to separate a performance-critical step from a potentially type-unstable precursor, is sometimes referred to as introducing a *function barrier*.

As a general rule, when writing multidimensional code you should ensure that the main iteration is in a separate function from type-unstable precursors. Even when you take appropriate precautions, there's a potential "gotcha": if your inner loop is small, julia's ability to inline code might eliminate the intended function barrier, and you get dreadful performance. For this reason, it's recommended that you annotate function-barrier callees with `@noinline`:

```
@noinline function sumalongdims!(B, A)
    ...
end
```

Of course, in this example there's a second motivation for making this a standalone function: if this calculation is one you're going to repeat many times, re-using the same output array can reduce the amount of memory allocation in your code.

Filtering along a specified dimension (exploiting multiple indexes)

One final example illustrates an important new point: when you index an array, you can freely mix `CartesianIndex`es and integers. To illustrate this, we'll write an exponential smoothing filter. An efficient way to implement such filters is to have the smoothed output value `s[i]` depend on a combination of the current input `x[i]` and the previous filtered value `s[i-1]`; in one dimension, you can write this as

```
function expfilt1!(s, x, α)
    0 < α <= 1 || error("α must be between 0 and 1")
    s[1] = x[1]
    for i = 2:length(a)
        s[i] = α*x[i] + (1-α)*s[i-1]
    end
    s
end
```

This would result in an approximately-exponential decay with timescale `1/α`.

Here, we want to implement this algorithm so that it can be used to exponentially filter an array along any chosen dimension. Once again, the implementation is surprisingly simple:

```
function expfiltdim(x, dim::Integer, α)
    s = similar(x)
    Rpre = CartesianRange(size(x)[1:dim-1])
    Rpost = CartesianRange(size(x)[dim+1:end])
    _expfilt!(s, x, α, Rpre, size(x, dim), Rpost)
end

@noinline function _expfilt!(s, x, α, Rpre, n, Rpost)
    for Ipost in Rpost
        # Initialize the first value along the filtered dimension
        for Ipre in Rpre
            s[Ipre, 1, Ipost] = x[Ipre, 1, Ipost]
        end
        # Handle all other entries
        for i = 2:n
            for Ipre in Rpre
                s[Ipre, i, Ipost] = α*x[Ipre, i, Ipost] + (1-α)*s[Ipre, i-1, Ipost]
            end
        end
    end
    s
end
```

Note once again the use of the function barrier technique. In the core algorithm (`_expfilt!`), our strategy is to use *two* `CartesianIndex` iterators, `Ipre` and `Ipost`, where the first covers dimensions `1:dim-1` and the second `dim+1:ndims(x)`; the filtering dimension `dim` is handled separately by an integer-index `i`. Because the filtering dimension is specified by an integer input, there is no way to infer how many entries will be within each index-tuple `Ipre` and `Ipost`. Hence, we compute the `CartesianRange`s in the type-unstable portion of the algorithm, and then pass them as arguments to the core routine `_expfilt!`.

What makes this implementation possible is the fact that we can index `x` as `x[Ipre, i, Ipost]`. Note that the total number of indexes supplied is `(dim-1) + 1 + (ndims(x)-dim)`, which is just `ndims(x)`. In general, you can supply any combination of integer and `CartesianIndex` indexes when indexing an `AbstractArray` in Julia.

The AxisAlgorithms package makes heavy use of tricks such as these, and in turn provides core support for high-performance packages like Interpolations that require multidimensional computation.

Additional issues

It's worth noting one point that has thus far remained unstated: all of the examples here are relatively *cache efficient*. This is a key property to observe when writing efficient code. In particular, julia arrays are stored in first-to-last dimension order (for matrices, "column-major" order), and hence you should nest iterations from last-to-first dimensions. For example, in the filtering example above we were careful to iterate in the order

```
for Ipost ...
    for i ...
        for Ipre ...
            x[Ipre, i, Ipost] ...
```

so that `x` would be traversed in memory-order.

Summary

As is hopefully clear by now, much of the pain of writing generic multidimensional algorithms is eliminated by Julia's elegant iterators. The examples here just scratch the surface, but the underlying principles are very simple; it is hoped that these examples will make it easier to write your own algorithms.

Julia is a NumFocus project. We thank Fastly for their generous infrastructure support.

Donate