ArraysAndMatrices

November 12, 2018

1 More Details on Arrays and Matrices

1.1 Array Operations

One key feature for doing math in Julia are the `broadcast` and `map` operations. The `map` operation is like an R or MATLAB `apply` which applies a function to each element of an array. For example, we can apply the anonymous function `(x)->x^2` to each element via:

```
In [1]: map((x)->x^2,1:5)
```

```
Out[1]: 5-element Array{Int64,1}:
          1
          4
          9
         16
         25
```

The `broadcast` operation is similar except it is for "elements which have a shape" and it will "broadcast the shaped objects to a common size by expanding singleton dimensions". For example, the following broadcast's `+` onto `A` and `B`:

```
In [2]: A = 1:5 # Acts like a column vector, Julia is "column-major" so columns come first
        B = [1 2
             3 4
             5 6
             7 8
             9 10]
        broadcast(+,A,B)
```

```
Out[2]: 5×2 Array{Int64,2}:
          2   3
          5   6
          8   9
         11  12
         14  15
```

If `A` and `B` are the same size, then broadcasting is the same as mapping.

1.1.1 In-Depth Julia 1: Special Broadcasting Syntax

One major area (which is under a lot of active development) is the specialized broadcasting syntax. The short summary is, putting a `.` with a function or operator causes it to broadcast. For example, we can broadcast any function with the syntax `f.(x)`, and broadcast operators by `.+` and related. For example:

```
In [4]: A = 1:5
        B = [2;3;4;5;6]
        A.*B

Out[4]: 5-element Array{Int64,1}:
          2
          6
         12
         20
         30
```

People coming from MATLAB might recognize this as "element-wise multiplication". If this was a basic introduction to Julia, I'd say this was element-wise multiplication and be done with it. However, this is the non-trivial introduction. [Note: Some of this is not true right now (v0.5) but is becoming true...].

While it looks the same to the user, the implementation is very different In MATLAB and earlier versions of Julia, `.*` was an operator. In Julia's more refined world, we can explain this as `.*{T<:Number,N}(x::Array{T,N},y::Array{T,N})` being a function, and `A.*B` calling this function. However, if `.*` is just a function, then

```
In [5]: C = [3;4;5;2;1]
        A.*B.*C

Out[5]: 5-element Array{Int64,1}:
          6
         24
         60
         40
         30
```

the operation `A.*B.*C` actually expands into `.*(A,.*(B,C))`. Let's think of how we'd implement `.*`.

Question 1 How would you implement `broadcast_mult` as a function (not using broadcast)? Don't peak below!

```
In [10]: function broadcast_mult(x,y)
             output = similar(x) # Makes an array of similar size and shape as x
```

```
        for i in eachindex(x) # Let the iterator choose the fast linear indexing for x
            output[i] = x[i]*y[i]
        end
        output
    end
```

```
Out[10]: broadcast_mult (generic function with 3 methods)
```

Notice that `broadcast_mult` creates an array every time it is called. Therefore a naive approach where `.*` is a function creates two arrays in the call `A.*B.*C`. We saw earlier that reducing memory allocations leads to vastly improved performance, so a better implementation would be to do this all together as one loop:

```
In [11]: function broadcast_mult(x,y,z)
        output = similar(x) # Makes an array of similar size and shape as x
        for i in eachindex(x) # Let the iterator choose the fast linear indexing for x
            output[i] = x[i]*y[i]*z[i]
        end
        output
    end
```

```
Out[11]: broadcast_mult (generic function with 4 methods)
```

(but notice this doesn't really work because now `.*` isn't a binary operator and therefore the inline syntax won't work). This optimization is known as "loop fusing". Julia does this by searching for all of the broadcasts in a line and putting them together into one broadcast statement during parsing time. Therefore, in Julia `A.*B.*C` creates an anonymous function and broadcasts on it, like

```
In [12]: broadcast((x,y,z)->x*y*z,A,B,C)
```

```
Out[12]: 5-element Array{Int64,1}:
          6
         24
         60
         40
         30
```

Notice that this is equivalent to our 1-loop solution. However, because all array-based math uses this broadcasting syntax with a `.`, Julia can fuse the broadcasts on all sorts of mathematical expressions on arrays:

```
In [13]: A.*B.*sin.(C)
```

```
Out[13]: 5-element Array{Float64,1}:
           0.2822400161197344
          -4.540814971847569
         -11.507091295957661
          18.185948536513635
          25.244129544236895
```

One last thing to note is that we can also broadcast `=`. This would be the same thing is as the loop `A[i] = ...` and thus requires the array `A` to already be define. Thus for example, if we let

```
In [14]: D = similar(C)

Out[14]: 5-element Array{Int64,1}:
          139768703295680
          139768703295616
          139768864023840
          139768862959040
          139768703295488
```

then the operation (Using `@btime` in BenchmarkTools.jl to get an accurate measurement)

```
In [15]: using BenchmarkTools
         @btime D.=A.*B.*C

  518.325 ns (4 allocations: 96 bytes)


Out[15]: 5-element Array{Int64,1}:
           6
          24
          60
          40
          30
```

does not allocate any arrays. Reducing temporary array allocations is one way Julia outperforms other scientific computing languages.

Summary: `.` makes operations element-wise, but in a very smart way.

1.2 Vectors, Matrices, and Linear Algebra

Julia's linear algebra syntax follows MATLAB's to a large extent (it's just about the only thing MATLAB got right!). We already saw this a little bit by seeing Julia's array indexing syntax. For example, we can get the first three elements by `1:3`:

```
In [16]: A = rand(4,4) # Generate a 4x4 random matrix
         A[1:3,1:3] # Take the top left 3-3 matrix

Out[16]: 3×3 Array{Float64,2}:
          0.152913   0.858814  0.554534
          0.0951324  0.124213  0.982151
          0.451213   0.238936  0.610605
```

Note that Julia is column-major, meaning that columns come first in both indexing order and in the computer's internal representation.

1.2.1 In-Depth Julia 2: Views

Notice that `A[1:3,1:3]` returned an array. Where did this array come from? Well, since there was no 3x3 array before, `A[1:3,1:3]` created an array (i.e. it had to allocate memory)

```
In [17]: @time A[1:3,1:3]
```

```
  0.000006 seconds (7 allocations: 384 bytes)
```

```
Out[17]: 3×3 Array{Float64,2}:
          0.152913   0.858814  0.554534
          0.0951324  0.124213  0.982151
          0.451213   0.238936  0.610605
```

Do you always have to allocate memory when making new arrays? We saw before this wasn't the case when dealing with references. Recall the example where modifying one array modified another:

```
In [18]: a = [1;3;5]
         @time b = a
         a[2] = 10
         a
         @time c = copy(a)
```

```
  0.000002 seconds (5 allocations: 208 bytes)
  0.002538 seconds (28 allocations: 1.828 KiB)
```

```
Out[18]: 3-element Array{Int64,1}:
           1
          10
           5
```

Notice that in the first case making `b` didn't allocate an array: it just made an object with a pointer (an Int64), and had that pointer point to the same array as `a`. To better understand this behavior and exploit to for major performance gains, we need to make a distinction. The array itself is the memory layout. For Julia arrays, this is actually a C-pointer to a contiguous 1-dimensional slot of memory. The `Array` type in Julia (and thus `Vector` and `Matrix` which are type-alises for `Array{T,1}` and `Array{T,2}` respectively) is a "view" to to that actual array. A view is a type which points to an array, and has a compatibility layer that changes how things like the indexing works. For example: if we define the matrix

```
In [19]: A = rand(4,4)
```

```
Out[19]: 4×4 Array{Float64,2}:
          0.426805  0.244754     0.760274  0.0776648
          0.263629  0.992725     0.663553  0.642327
          0.220796  0.000123405  0.41826   0.344501
          0.502755  0.604712     0.868655  0.245527
```

then the array that we made is actually a 16-number long sequence (of 64-bit Floats) in memory, and `A` is a view to that array which makes it index "like" it was 2-dimensional (reading down the columns). This tells us one thing already: looping through the columns is faster than looping through the rows. Indeed we can easily test this:

```
In [20]: function testloops()
             b = rand(1000,1000)
             c = 0 # Need this so that way the compiler doesn't optimize away the loop!
             @time for i in 1:1000, j in 1:1000
                 c+=b[i,j]
             end
             @time for j in 1:1000, i in 1:1000
                 c+=b[i,j]
             end
             bidx = eachindex(b)
             @time for i in bidx
                 c+=b[i]
             end
         end
         testloops()

  0.001181 seconds
  0.000354 seconds
  0.000264 seconds
```

One should normally use the `eachindex` function since this will return the indices in the "fast" order for general iterator types.

In this terminology `A[1:3,1:3]` isn't a view to the same memory. We can check this by noticing that it doesn't mutate the original array:

```
In [21]: println(A)
         B = A[1:3,1:3]
         B[1,1]=100
         println(A)

[0.426805 0.244754 0.760274 0.0776648; 0.263629 0.992725 0.663553 0.642327; 0.220796 0.0001234
[0.426805 0.244754 0.760274 0.0776648; 0.263629 0.992725 0.663553 0.642327; 0.220796 0.0001234
```

If we instead want a view, then we can use the `view` function:

```
In [22]: B = view(A,1:3,1:3) # No copy involved
         B[1,1] = 100 # Will mutate A
         println(A)

[100.0 0.244754 0.760274 0.0776648; 0.263629 0.992725 0.663553 0.642327; 0.220796 0.000123405
```

There are many cases where you might want to use a view. For example, if a function needs the `i`th column, you may naively think of doing `f(A[i,:])`. But, if `A` won't be changed in the loop, we can avoid the memory allocation (and thus make things faster) by sending a view to the original array which is simply the column: `f(view(A,i,:))`. Two functions can be used to give common views. `vec` gives a view of the array as a Vector and `reshape` builds a view in a different shape. For example:

```
In [23]: C = vec(A)
         println(C)
         C = reshape(A,8,2) # C is an 8x2 matrix
         C

[100.0, 0.263629, 0.220796, 0.502755, 0.244754, 0.992725, 0.000123405, 0.604712, 0.760274, 0.6


Out[23]: 8×2 Array{Float64,2}:
          100.0          0.760274
            0.263629     0.663553
            0.220796     0.41826
            0.502755     0.868655
            0.244754     0.0776648
            0.992725     0.642327
            0.000123405  0.344501
            0.604712     0.245527
```

Since these operations do not copy the array, they are very cheap and can be used without worrying about performance issues.

1.2.2 Back to Linear Algebra

Julia performs functions on matrices by default for dispatches on matrices. For example, `+` is the matrix addition, while `*` is matrix multiplication. Julia's `*` calls into a program known as OpenBLAS so that way `*` is an optimized multithreaded operation. For example:

```
In [24]: A = rand(4,4); B = rand(4,4)
         C = A*B # Matrix Multiplication
         D = A.*B # Element-wise Multiplication
         C-D # Not zero

Out[24]: 4×4 Array{Float64,2}:
           1.34143   1.16673   0.467808  1.40722
           0.524589  1.26758   0.180655  1.63217
           1.69161   1.22343   0.607505  1.16599
          -0.118427  0.475244  0.160007  0.636878
```

A common operation is to solve the linear system `Ax=b`. In Julia this is done by `A\b`:

```
In [25]: b = 1:4
         A\b
```

```
Out[25]: 4-element Array{Float64,1}:
            9.356335977910366
           12.159190681773488
          -34.20454880191792
            9.715477816124533
```

Note that this uses a direct solver. Iterative solvers for linear equations can be found in IterativeSolvers.jl hosted by the JuliaMath organization.

1.3 A note about "Vectorization"

In MATLAB/Python/R you're told to "vectorize" your options, i.e. use `apply` or these `.*` operations, in order to speed up computations. This is because these operations call C programs which will be faster than any interpreted MATLAB/Python/R loop. In Julia, that's not the case: as long as your functions are type-stable, you will get maximal performance. Thus vectorization does not improve performance.

In fact, vectorization can reduce performance by creating "temporary arrays". Those are the intermediate array allocations that come up with doing operations like `C[i,:] = A[i,:] .* B[i,:]`. In general, for the best performance one should avoid vectorized calls or be careful to use the broadcast/view syntax to define a statement without temporaries:

```
In [27]: i = 1
         C[i,:] .= view(A,i,:) .* view(B,i,:)

Out[27]: 4-element view(::Array{Float64,2}, 1, :) with eltype Float64:
          0.01517038709424138
          0.2386832186363507
          0.024709948145356094
          0.5974524955186336
```

Note the odd quirk: array indexing is a view when on the left-hand side

Discussion: why is this the case?

1.4 Sprase Matrices

Sprase Matrix capabilities are provided by SuiteSparse. Note that these are saved in a table format, where there are triplets (i,j,value) which denote the existance of a non-zero element at `(i,j)` of value `value`. A sparse matrix can be created through the `sparse` command:

```
In [29]: using SparseArrays
         A = sparse([1;2;3],[2;2;1],[3;4;5])

Out[29]: 3×2 SparseMatrixCSC{Int64,Int64} with 3 stored entries:
           [3, 1]  =  5
           [1, 2]  =  3
           [2, 2]  =  4
```

They can be converted into a dense matrix with the `Array` command

```
In [31]: Array(A)

Out[31]: 3×2 Array{Int64,2}:
          0  3
          0  4
          5  0
```

The documentation shows a lot more that you can do.

1.5 Special Matrix Types

Like the rest of Julia, types and multiple dispatch is used to "secretly enhance performance". There are many matrix types, so I will just show a few and leave the rest to the documentation.

1.5.1 Matrix Forms

Many matrices follow specific forms: diagonal, tridiagonal, etc. Julia has special types for these common matrix forms. For example, we can define a `Tridiagonal` by giving it three vectors:

```
In [33]: using LinearAlgebra
         A = Tridiagonal(2:5,1:5,1:4)

Out[33]: 5×5 Tridiagonal{Int64,UnitRange{Int64}}:
          1  1
          2  2  2
            3  3  3
               4  4  4
                  5  5
```

We can inspect it to see its internal form:

```
In [36]: fieldnames(typeof(A))

Out[36]: (:dl, :d, :du, :du2)

In [37]: A.d

Out[37]: 1:5
```

Notice that what the array stores is the vectors for the diagonals themselves. It's clear to see that this gives a memory enhancement over a dense matrix, and it gives a performance advantage because a dense matrix would have an extra operation for each `0`. However, it's also faster than a sprase matrix since a sparse matrix is stored as a table `(i,j,value)` and retriving from the table has a bit of overhead, while this is stored as 3 (actually 4...) contiguous arrays. Therefore you get a performance boost by using a special matrix form like `Tridiagonal` whenever one is available. Note that these special matrix forms are outfitted with dispatches so that operations on them work seamlessly like with normal matrices. For example, we can multiply a Tridiagonal by a dense matrix:

```
In [38]: A*rand(5,5)
```

```
Out[38]: 5×5 Array{Float64,2}:
          0.496473  1.01384  0.973241  1.67338  0.684759
          1.86855   2.58954  2.4101    3.41502  2.01966
          2.89199   5.67443  2.91435   3.38253  2.52367
          4.46756   6.01644  2.68273   1.98607  5.28988
          3.39544   6.11589  2.19438   2.31193  4.98701
```

1.5.2 The UniformScaling Operator

One interesting type is the `UniformScaling` operator `I`. Essentially, I uses dispatches to cleverly act like the identity matrix without ever forming a matrix. For example, to mathematically subtract a scalar from a matrix `A` we use the notation

$$A - \lambda I$$

We can do this naturally with the `I` operator:

```
In [43]: A = rand(5,5)
          = 2
         A - *I
```

```
Out[43]: 5×5 Array{Float64,2}:
          -1.56585    0.907288    0.00306355   0.728916     0.240867
           0.810397  -1.60421     0.129275     0.00162653   0.840859
           0.609112   0.0177244  -1.65993      0.511456     0.97296
           0.175652   0.409159    0.611265    -1.48017      0.15766
           0.313506   0.375135    0.141591     0.539388    -1.72274
```

The MATLAB or NumPy way would be to create the identity matrix via a command like `eye(5)`, but notice this prevents the allocation of a 5x5 array. For large matrices, this operation is huge and thus this could lead to some good performance improvements.

1.5.3 Factorization Forms

One last type of interest are the factorization forms. In many cases, you factorize a matrix using some factorization command in order to speed up subsequence `A\b` calls. Normally you have to remember how this is done. For example, we can use a QR factorization to solve a linear system like:

```
In [44]: b = 1:5
         A = rand(5,5)
         Q,R = qr(A)
         println(A\b)
         println(inv(R)*Q'*b)
```

```
[3.2111, 0.194886, -3.45612, -0.380946, 3.49735]
[3.2111, 0.194886, -3.45612, -0.380946, 3.49735]
```

Thus we can save the variables `Q` and `R` and use `inv(R)*Q'*b` instead of `A\b` and get better performance. This is the NumPy/MATLAB way. However, that requires remembering the details of the factorization. Instead, we can have Julia return a factorization type:

```
In [46]: q = qr(A)
```

```
Out[46]: LinearAlgebra.QRCompactWY{Float64,Array{Float64,2}}
         Q factor:
         5×5 LinearAlgebra.QRCompactWYQ{Float64,Array{Float64,2}}:
          -0.459412   0.419128   0.705041  -0.110726  -0.32238
          -0.148085  -0.682807   0.512443   0.39177    0.309457
          -0.420809  -0.566188  -0.220962  -0.504135  -0.446513
          -0.555282   0.160027  -0.165325  -0.29989    0.740802
          -0.530654   0.109219  -0.405169   0.700121  -0.228355
         R factor:
         5×5 Array{Float64,2}:
          -1.25445  -0.656222  -0.629968   -1.02671   -1.51678
           0.0      -0.462833   0.0543968  -0.314465  -0.371835
           0.0       0.0        0.65199    0.830083   0.27148
           0.0       0.0        0.0        0.275926   0.447929
           0.0       0.0        0.0        0.0        0.222577
```

What this does is it internally stores `Qt` (`Qt = Q'`) and `Rinv` (`Rinv = inv(R)`). There is a dispatch defined for this type on which makes the `QRCompactWY` type perform the fast algorithm `Rinv*Qt*b`, giving you the performance without having to remember anything:

```
In [47]: q\b
```

```
Out[47]: 5-element Array{Float64,1}:
           3.211102680104207
           0.1948859185810493
          -3.4561161435917
          -0.38094578267516777
           3.497350976874699
```

The result is fast algorithms with clean code.

1.6 Random Numbers

One last little detail is for random numbers. Uniform random numbers are generated by the `rand` function, while normal random numbers are generated by the `randn` function.

```
In [48]: rand(5)
```

```
Out[48]: 5-element Array{Float64,1}:
          0.5481921969929076
          0.9319266861112498
          0.48822004014222653
          0.6741427500410946
          0.25974192551299624
```

```
In [49]: randn(5,5)

Out[49]: 5×5 Array{Float64,2}:
          -1.00122    0.244247     1.33512   -0.777973    1.03879
           2.39675    0.00128483  -0.604522  -0.0707705   0.282255
          -0.694438   0.68408      0.178648   0.320393   -1.12351
           0.760748  -2.19305      0.315202  -0.0435269   0.124363
           1.41287   -0.142549     0.142544  -0.460031   -1.38784
```

The argument is the size of the array. You can make random numbers which match another array with the `size` function:

```
In [50]: a = [1 2
              3 4]
         randn(size(a))

Out[50]: 2×2 Array{Float64,2}:
          -1.63844   0.1087
          -0.751702  1.11514
```