Array Programming

Section 1. What is Array Programming?

Let's start with trying to define what is array programming and array programming langauges?

To start off, let's say it is a programing language that allows you to work with arrays or array-like data.

Well, that's basically all languages - lists and vectors in Racket, tuples in Python, arrays in C and JS, etc.

But then this definition does not mean anything really if it includes basically every langauge.

I think the better definition is a languages that makes working with arrays easier than it is in most general-purpose languages.

Let's see what wikipedia has to say about it:

> In computer science, array programming refers to solutions that allow the application of operations to an entire set of values at once. Such solutions are commonly used in scientific and engineering settings.
>
> Modern programming languages that support array programming (also known as vector or multidimensional languages) have been engineered specifically to generalize operations on scalars to apply transparently to vectors, matrices, and higher-dimensional arrays. These include APL, J, Fortran, MATLAB, Analytica, Octave, R, Cilk Plus, Julia, Perl Data Language (PDL).

This list has a lot of different langauges, and all of them are very different, with a very disctinctive set of features. Let's go through the history of array PLs, look at some code examples and see if we can find any intersections.

Section 1.5. Example explanation detour

For the most part, the examples are going to be matrix multiplication. It's a small enough example making it easy and quick to explain. But it also is extremely important, so important that a lot of ML/AI optimizations come down to making matrix multiply faster and faster. There are other applications like polyhedral optimizations (constraints can be represented as matrices) and resource allocation in economics.

Section 2. History of Array Programming

Fortran, the Eternal As always, when we talk about history of programming languages, Fortran is there.

Originally developed in late 50s at IBM, it was designed to make number calculations and processing easier. (As is evident from the name, Formula Translator).

As for the feature set, it does not have many: for loops and indexing.

It's still actively developed, but it did not get any new features that makes it easier to program in (especially compared to other languages we'll see later on). It did get some features to make parallelizing easier, but they are just a different kind of for-loops.

It's also still actively used in the supercomputing field. A big part of scientific code actually is still written and uses Fortran (in big part because of the compiler and the total accumulated knowledge about making Fortran code run fast). Even outside supercomputing, a lot of number processing code is still written in it (also evident by the fact that Nvidia CUDA compiler supports Fortran).

```
do i = 0, n
  do j = 0, n
    Ans(i, j) = 0
    do k = 0, n
      Ans(i, j) = Ans(i, j) + A(i, k) * B(k, j)
    end do
  end do
end do
```

APL, programming math Stands for A Programming Language.

Developed in early 60s by Ken Iverson. The original design idea was that writing programs should be similar to writing math notation. (There will be some examples soon).

This is where we start to see some more interesting features that make programming with arrays nicer:

- Tacit/point free programming
- Function lifting
- Lots of other built-in niceties

```
A ← M +.× N
```

```
life ← {⊃1 ⍵ ∨.∧ 3 4 = +/ +⌿ ¯1 0 1 ∘.⊖ ¯1 0 1 ⌽¨ ⊂⍵}
```

From Wikipedia page on APL

```
CV←{0⌈(,[2 3 4]3 3⌺⍵)+.×,[0 1 2]⍺}
```

Aaron Hsu, ARRAY 2023 "U-Net CNN in APL: Exploring Zero-Framework, Zero-library Machine Learning"

But like always, it comes with cons: - Math is not ASCII, you would need some extended character set. - Functions can only have 1 or 2 arguments. - Dynamically typed (and extremely hard to statically typed) Types are extremely useful

for compiling efficient code and for specification. - Even parsing is delayed until runtime because of the irregular structure (The iteration structure and behavior is dependent on runtime values) - Due to all this, it can just be interpreted and compiling it still has not really been solved (and probably never will).

J, APL but ASCII Developed by the same person, Ken Iverson in the 90s.

More or less just APL, but in ASCII.

Has more or less the same problems as APL, but you can type it on a normal keyboard now.

Futhark, the modern view The modern iteration on array programming. The product of the PhD thesis of Troels Henriksen at DIKU (in 2014).

Functional programming view on array programming is the main design idea. It is statically typed, has explicit iteration structure.

This allows it to be efficiently compiled to both parallel CPUs and GPUs.

Has been is actively developed since, adding features like function lifting, automatic differentiation and others.

To understand the following code snippet, we need to introduce some of the functions used.

```
def map [m] (A : [m] 'a) (f: 'a -> 'b) : [m] 'b
def map2 [m] (A : [m] 'a) (B : [m] 'b) (f: 'a -> 'b -> 'c) : [m] 'c
def reduce [m] (A : [m] 'a) (f: 'a -> 'a -> 'a) (init: 'a) : 'a
def transpose [m] [n] (A : [m][n] 'a) : [n] [m] 'a
```

```
def matmul_i32 [n][m][p] (A: [n][m]i32) (B: [m][p]i32) : [n][p]i32 =
  map (\A_row ->
         map (\B_col ->
                reduce (+) 0 (map2 (*) A_row B_col))
             (transpose B))
      A
```

NumPy, the modern view 2 A Python library that has gained a lot of popularity in the last decade or so.

A lot of code has been written in it, and for prototyping it works great. The problem comes when you need to write something really performant. Python has only one real way to be fast: call C/C++ code. That's what most libraries that need performance do. What this means if you want to make an often-used function faster, you would probably need to write an implementation in C, and then connect the FFI (foreign function interface, calling functions written in one language from another). This kind of approach also lacks the ability to do a lot of optimizations, since it would happen at runtime, so you have very limited time to do it.

There are quite a few benefits though. First of all, you get access to all of Python, so a lot of things have already been implemented. Second of all, the community is rather large, so getting help is usually quite fast and easy. Third of all, even though I say the performance is not the best, it still is fast. All the primitive functions have been hand-written and hand-optimized in C/C++ so they are basically the best you can get. And Python serves as just glue for that code, so it is not a big performance hit (for smaller programs).

```
numpy.matmul(M, N) # feels a little unfair
np.sum(A[:, :, np.newaxis] * B.T[np.newaxis, :, :], axis=1)
```

Remora, the modern view 3 Developed by Justin Slepak at Northeastern over the course of his PhD.

Combines the niceties of APL with the flexibility of Scheme and a type system.

In general, not as expressive as Scheme, as Remora is not even Turing complete. But it has a lot of expressivity for working with arrays specifically.

This also brings up a good point about design of languages: it's always about the balance between how general purpose it is, how expressive it is and how easy it is to compile.

Section 3 Principal Frame

The core to the programming model that Remora provides is the idea of the principal frame.

To understand the principal frame, we first need to establish some definitions.

First of all, in Remora everything is an array. Array has 3 parts to it: data stored (all the elements), element type and shape. Shape is a sequence of dimensions. Rank is the length of shape. (Important: totally different from rank in linear algebra)

```
1 ; shape = [], type = int, data = [1]
[1 2 3] ; shape = [3], type = int, data = [1 2 3]
[[1 2 3] [4 5 6]] ; shape = [2 3], type = int, data = [1 2 3 4 5 6]
```

Functions specify the rank of the arguments they accept:

```
(define (add1 (n 0)
  (+ n 1))
(define (dot-product (v 1) ...)
(define (matrix-multiply (m 2) (n 2) ...)
```

In an application, principal frame is the argument frame with the longest shape. All other arguments must be the prefix of the principal frame.

Let's look at an example to see what's happening.

```
(add1 2) ; frame is []
(add1 [1 2 3]) ; frame is [3]
(add1 [[1 2 3] [4 5 6]]) ; frame is [2 3]
(dot-product [[1 2 3] [4 5 6]]) ; frame is [2]
(matrix-multiply [[1 2] [3 4] [5 6]] [[1 0] [0 1]]) ; frame = []
```

The cool thing is that function position in an application is also an array.

```
([add1 sub1] 2)
> [3 1]
```

The way this works is by 'ignoring' the suffix part of index for non-principal frame arguments. To understand what exacly happens, let's translate one of the calls into a langauge with for loops.

```
(add1 [[1 2 3] [4 5 6]])
->
let in = [[1 2 3] [4 5 6]]
for i = 0 to 1:
  for j = 0 to 2:
    result[i, j] = add1(in[i, j])
```

```
([add1 sub1] [[1 2 3] [4 5 6]])
->
let f = [add1, sub1]
let in = [[1 2 3] [4 5 6]]
for i = 0 to 1:
  for j = 0 to 2:
    result[i, j] = f[i](in[i, j])
```

Section 4. Remora demo

Let's start with some linear algebra functions that will give us some practice.

```
(define (dot-product (v 1) (w 1))
  (reduce + 0 (* v w)))

(dot-product [1 0 2] [2 3 1])
> 4
```

```
(define (matrix*vector (a 1) (B 2))
  (reduce + 0 (* a B)))

(define (matrix-multiply (A 2) (B 2))
  (matrix*vector A B))

> (matrix-multiply (iota [2 2]) (iota [2 3]))
[[3 4 5] [9 14 19]]
```

One important thing to notice here is that `dot-product` and `matrix*vector` look exactly the same. Can we replace one with the other?

```
(define (matrix-multiply-2 (A 2) (B 2))
  (dot-product A B))

> (matrix-multiply-2 (iota [2 2]) (iota [2 3]))
error: incompatible argument frames '(#() #(2) #(3))
```

There is also a tiny definition of matrix multiply.

```
(define (matrix-mutliply-3 (A 2) (B 2))
  (#r(0 0 2)reduce + 0 (#r(1 2)* A B)))

> (matrix-multiply-3 (iota [2 2]) (iota [2 3]))
[[3 4 5] [9 14 19]]
```

We used a new language construct called `rerank`. It forces the function after it to use specific ranks for arguments. The multiply case says "Multiply each row in A by entire matrix B". The reduce case says "Sum rows (point-wise) in each matrix that's the result of the multiplication." You could also achieve by inline application:

```
((fn ((A 1) (B 2)) (* A B)) A B) ; instead of (#r(1 2)* A B)
```

For fun, let's try some statistics.

```
(define (mean (arr 1))
  (define sum (reduce + 0 arr))
  (define n (length arr))
  (/ sum n))

> (mean [1 2 3 4 5])
3
```

But what if we want to get mean of an array of rank more than 1? We can do it, but we need two more pieces: `ravel` and `shape-of`. `ravel` turns a nested array into a vector. `shape-of` returns a vector representing the shape of the array

```
(define (mean (arr all))
  (define n (reduce * 1 (shape-of arr)))
  (define data (ravel arr))
  (define sum (reduce + 0 data))
  (/ sum n))

> (mean [1 2 3 4 5])
3
> (mean [[1 2] [3 4]])
5/2 = 2.5
```

```
> (#r(1)mean [[1 2] [3 4]])
[1.5 3.5]
```

```
(define (variance (arr all))
  (define mean-arr (mean arr))
  (define diff (- arr mean-arr))
  (define n (reduce * 1 (shape-of arr)))
  (define sqr-sum (reduce + 0 (ravel (* diff diff))))
  (/ sqr-sum (sub1 n)))
```

```
> (variance [1 2 3 4 5])
5/2 = 2.5

> (variance [[1 2] [3 4]])
5/3

> (#r(1)variance [[1 2] [3 4]])
[1/2 1/2]
```

To end this section, we are going to compare convolution written in numpy and in Remora.

Remora:

```
(define (conv-2d (in 2) (weights 2))
  (define all-windows (windows in (shape-of weights)))
  (#r(0 0 1)reduce + 0 (#r(0 0 1)reduce + 0 (#r(2 2)* all-windows weights))))

(define (conv-2d-pad (in 2) (weights 2) (pad 0))
  (define origin [(- pad) (- pad)])
  (define padded-in (subarray/fill in origin (+ (* 2 pad) (shape-of in)) 0))
  (conv-2d padded-in weights))
```

And here's a bonus version that can handle n-dimensional convolution (can only happen in untyped):

```
(define (conv-nd (in all) (weights all))
  (define n (length (shape-of in)))
  (define all-windows (windows in (shape-of weights)))
  (define outer-frame (take n (shape-of all-windows)))
  (define replicated-weights ((fn ((_ 0)) weights) (iota outer-frame)))
  (reduce-n + 0 (* all-windows replicated-weights) n))

(define (conv-nd-pad (in all) (weights all) (pad 0))
  (define origin ((fn ((_ 0)) (- pad)) (shape-of in)))
  (define padded-in (subarray/fill in origin (+ (* 2 pad) (shape-of in)) 0))
  (conv-nd padded-in weights))
```

NumPy code for convolution (credit to NumberSmithy)

```
def asStride(arr, sub_shape, stride):
    s0, s1 = arr.strides[:2]
    m1, n1 = arr.shape[:2]
    m2, n2 = sub_shape[:2]

    view_shape = (1+(m1-m2)//stride, 1+(n1-n2)//stride, m2, n2)+arr.shape[2:]
    strides = (stride*s0, stride*s1, s0, s1)+arr.strides[2:]
    subs = np.lib.stride_tricks.as_strided(
        arr, view_shape, strides=strides, writeable=False)

    return subs

def conv3D3(var, kernel, pad=0):
    kernel = checkShape(var, kernel)
    if pad > 0:
        var_pad = padArray(var, pad, pad)
    else:
        var_pad = var

    view = asStride(var_pad, kernel.shape, 1)
    conv = np.sum(view*kernel, axis=(2, 3))

    return conv
```

Couple of notes: - NumPy version is supports strides, while Remora does not. It's not a very big add (and I've done it before), but it requires indexing manipulation that's not very well done right now. - There are a lot of different implementations of convolution in Python, I chose this one because it's the closest in style to the Remora code (even though it's slower). - If you need more exotic reduction operator in Python, you are out of luck. You would need to implement it yourself (vs just passing a different function in Remora)

Section 5. Where are the types?

I've mentioned before that Remora is a statically typed language. So where are all the types? The Remora code we have been working on is actually untyped Remora. It's a hash lang in Racket and uses a lot of Racket's built-ins to provide all the nice utilities.

There exists a definition for a typed Remora. It's the key to compiling a language like Remora.

Typed Remora has a dependent type system. I won't go into too much technical detail, because it will take at least a lecture to go through it. Put simply, now you can put arbitrary expression/programs into your types. That makes it undecidable in general, which is why usually you work with restricted dependent

types.

In Remora, types contain shape information like `[int]` is a scalar integer, `[float [2]]` is an array containing two floats, and `[int [10 12]]` is a 10 by 12 integer matrix.

Dimension is either constant, variable, or a sum of dimensions. Shape is either a sequence of dimensions, variable or append of shapes.

Why is it so restrictive, what if I want to do multiplication? Well, then it becomes undecidable.

Then why do we want such a type system? We can't even type something like `ravel` that we used in the demo earlier. It becomes way easier to compile it, because it shows you the entirety of the iteration structure. It's one of the differences between Remora and APL that allows the former to be compiled, even though it has a lot of similar features. And you can still compile something like that, you just don't know the dimensions of arrays anymore (which makes compiling way harder).

Section 6. Parallelism

First of all, what is parallelism? The parallelism is a property of programs that allows you to run parts of the program in parallel. Running in parallel means allocating work to independent execution units to speed up execution.

Why do we care? Well, if you have 1000 units of computation, all independent (like a map), and 10 execution units. Assume 1 unit of computation takes 1 second. If you can perfectly schedule all 10 execution units, it will take only 100 seconds vs 1000 seconds using a single one.

New hardware contains hundreds of CPU cores (like server CPUs, Xeons and EPYCs) and thousands of GPU cores. If you can do perfect parallelism, you can speed up program execution by a massive factor.

Why other languages do not do this? Well, a variety of reasons. First of all, it interacts in a weird way with side effects. The order of execution becomes non-deterministic so you cannot rely on that anymore. Secondly, most languages don't reason at all about parallelism, and the programmer does not provide annotations for what can and cannot be parallel. So, something like a compiler needs to take this unparallelized code and try to reason about it to find

Why Remora then? The design of type system, operators and core languages exposes very regular and parallelizable iteration structure, so it's much easier to compile parallel code compared to something like C++ or Fortran.

Section 7. So, what is array programming about?

Manipulating data to fit the computation better. In both Futhark and Remora, you don't write loops around your data/arrays. You instead restructure the data so that it fits the computational model you have, like maps, reduces, etc and avoiding using filters and such because they make all of your shapes unknown.

Section 8. Why isn't everyone using these languages then?

Fortran Well, it is totally used. It is very popular in the scientific and supercomputing space. One of the main reasons is that it has an incredible optimizing compiler that has been in development and a center of attention in compilers research.

APL There is variety of reasons why it failed to become popular (even though Ken Iverson got a Turing Award for it). First, it's quite slow because you cannot really compile it. Second, the language was not really designed to grow at all. Two quotes here:

> "APL is like a diamond. It has a beautiful crystal structure; all of its parts are related in a uniform and elegant way. But if you try to extend this structure in any way - even by adding another diamond - you get an ugly kludge. LISP, on the other hand, is like a ball of mud. You can add any amount of mud to it and it still looks like a ball of mud."
>
> – Joel Moses
>
> APL was designed by one man, a smart man—and I love APL—but it had a flaw that I think has all but killed it: there was no way for a user to grow the language in a smooth way.
>
> – Guy Steele Jr., Growing a Language, OOPSLA 1998

APL examples I showed exactly point to the problem with APL - you cannot really extend it in an organic way. Function names that a user defines are written almost always with ASCII characters, while built-in functions use glyphs. So, now when you try to write a library, calling library code looks very different from built-in code.

There is also an issue of code itself. APL is extremely terse, and while it's nice to showcase how powerful and expressive it is, it becomes really hard to maintain and debug.

J Similar to APL. Even though it's easier to grow, it was closed-source until 2011 (and licence was quite expensive otherwise). But Ken Iverson did do a lot of consulting at Wall Street using J.

Futhark Futhark is still a young language, and it's a research language. No company will use a research language for critical things. But it doesn't have the same flaws as APL does.

Remora Even younger research project, and does not have a good compiler yet.

Section 9. Conclusion

Array Programming is very specialized niche in Programming Languages. It's not even close to being a general purpose language, but what they are made for, they do well. As an example of this, neither Futhark nor Remora have any recursion or for-loops, they are both terminating languages (which plays to their advantage).