Embedding Pointful Array Programming in Python

Jakub Bachurski
kbachurski@gmail.com
University of Cambridge

Abstract

Multidimensional array operations are ubiquitous in machine learning. The dominant ecosystem in this field is centred around Python and NumPy, where programs are expressed with elaborate and error-prone calls in the *point-free* array programming model. Such code is difficult to statically analyse and maintain. Other array programming paradigms offer to solve these problems, in particular the *pointful* style of Dex. However, only limited approaches – based on Einstein summation – have been embedded in Python. We introduce Ein, a pointful array DSL embedded in Python. We also describe a novel connection between pointful and point-free array programming. Thanks to this connection, Ein generates performant, optimised and type-safe calls to NumPy. Ein reconciles the readability of comprehension-style definitions with the capabilities of existing array frameworks.

1 Introduction

The importance of the *array programming model* nowadays is difficult to overstate. Originally introduced in APL by Iverson [14], it shifts the spotlight from imperative-style loops (typical of languages like FORTRAN or C) to whole-array operations. It is universal across many domains, particularly due to its presence in Python – a language highly influential in areas like machine learning and scientific computing. Python's ecosystem in these areas is centred on the NumPy library, which implements this model [11]. What is more, many other of Python's array frameworks also derive from NumPy – e.g. TensorFlow, PyTorch, and JAX [1, 7, 17]. The array programming model has seen overwhelming success in practical applications, and significant engineering effort has been made across the industry to make it efficient [18].

However, the array programming model also suffers from many problems. It is often difficult to write, maintain and statically analyse. We focus on an alternative – *pointful array programming*, as introduced by Paszke et al. [18] in the Dex language. In this style, arrays can be seen as functions from indices to elements, and a particular focus is given to *array comprehensions*. On the other hand, the array programming model is seen as *point-free* – as it focuses on whole-array operations. The pointful style is often clearer and more expressive, but has so far required bespoke compilers.

In this work, I introduce **Ein** – a pointful array language embedded in Python. Ein executes programs by calling NumPy (or PyTorch/JAX) routines, leveraging work in the array programming model. This is possible thanks to a new formal connection between pointful and point-free array programs.

2 Background

2.1 Array programming model in NumPy

Programming in Python's leading array framework – NumPy – revolves around the **`ndarray`** data structure [11]. It is an n-dimensional, rectangular and homogenous *array* of primitive data types (usually just fixed-size integers and floats). Since the array is rectangular, we define its *shape* to be the list of sizes of its dimensions (or *axes*). We say the number of axes of an array its *rank*. Arrays of ranks 2, 1 and 0 are called *matrices*, *vectors* and *scalars* respectively. Axes are indexed from 0 – e.g. a matrix has axes 0 (rows) and 1 (columns).

Broadcasting. Elementwise operations are the arguably the most fundamental in array programming. In NumPy, a core feature of these is *broadcasting*, which generalises the notion of elementwise operations on arrays with mismatched ranks (e.g. scalar plus matrix). The crux of the idea is that axes of size 1 can be expanded into any other size by repeating elements, after which we operate elementwise. When ranks are mismatched, we prefix the shape of the lower-ranked array with 1-dimensions. For example:

array	shape
a	(100, 200, **1**)
b	(200, 300)
$a + b$	(100, 200, 300)

$$(a + b)_{i,j,k} = a_{i,j,\mathbf{0}} + b_{j,k}$$

To unlock broadcasting's potential, we use *shape manipulation* primitives. Squeezing (`squeeze`) and unsqueezing (`expand_dims`) are used to remove and add axes of size 1. Transposition permutes axes of an array. NumPy also offers primitives for indexing and constructing arrays (e.g. `arange(n) = [0 1 ... n-1]`).

Reductions. Though NumPy does not forbid simply looping in Python, the idiomatic approach to accumulating data is a *reduction* along an axis. NumPy can be said to be a **first-order** interface, as reductions are limited to an ad-hoc set of operations (e.g. `numpy.sum`).

Summary. NumPy's primitives are highly performant in comparison to pure Python. Therefore, Python loops operating on individual elements must be avoided, so as to ensure most of the runtime is spent inside NumPy routines. Instead, one manipulates array shapes to obtain the wanted operation. This approach is hard to reason about in complex cases. Figure 1 displays an example operation (based on Frostig et al. [7]) and its NumPy implementation – showcasing the problematic shape manipulation and axis indexing.

$$\texttt{pairwiseL1(A)}_{i,j} = \sum_k \left|\texttt{A}_{i,k} - \texttt{A}_{j,k}\right|$$

```
def pairwiseL1(A):
  return sum(
    abs(transpose(A, (1, 0))
        - expand_dims(A, axis=2)),
    axis=1
  )
```

Figure 1. An index-oriented specification of `pairwiseL1`, followed by a Python implementation using NumPy.

2.2 Pointful array programming

In functional programming, one can distinguish a **pointful** style and contrast it with the **point-free** one. This distinction is based on whether data-flow is given by variable names, or driven with combinators. A classic example is that of λ and SKI calculi, which are respectively pointful and point-free. Both have the same expressive power and one can compile λ to SKI via *bracket abstraction*. The main result of this work can be seen as **bracket abstraction for array programs**.

Paszke et al. [18] draw an analogous distinction between *pointful* (index-oriented) and *point-free* (array-oriented) array programming in the context of their work on Dex (example in Figure 2). This pointful style strives to think about arrays as functions, thus enabling a functional style of array programming. We contrast this with point-free whole-array computations in the array programming model. With pointful programs, one makes the data-flow with respect to array dimensions explicit. This leads to clearer and more composable programs.

2.3 Embedding domain-specific languages

Creating domain-specific languages (DSLs) has a long history [13]. They are by motivated by the need to create languages which achieve specific goals – for instance, Ein aims to be a pointful array language. Careful design choices simplify compilation and improve the programming experience.

A domain-specific language can be *standalone*, in which case it functions as an independent language. However, a refined approach is **embedding** a DSL in a host language. We focus on *deep-embedded* DSLs that work on the basis of term constructors [9]. The host language builds up programs in the DSL as values, and later evaluates them [2]. In the context of Python, we take JAX to be a leading example of an embedded DSL [7]. It uses the *tracing* technique to implement the deep-embedding. At runtime, lazy (deferred) computations are built up, producing a *term graph* of the underlying language. This representation can then be compiled for execution.

```
pairwiseL1 :: n=>d=>Real -> n=>n=>Real
pairwiseL1 x = for i j.
  sum (for k. abs (x.i.k - x.j.k))
```

Figure 2. Dex implementation of `pairwiseL1` [15].

3 Related work

This work builds on tools applying the array programming model in Python, the practice of embedding DSLs in it, and the recent developments in pointful array programming. Our approach to embedding the DSL follows previous work, e.g. JAX [7]; and we use existing point-free array libraries (Numpy, PyTorch, and JAX [7, 11, 17]) to execute Ein.

Ein's design as a pointful array language also follows prior art. It only has a part of Dex's features [18], as some would be too impractical to embed in Python – especially its dependent type system and effects. Some of Ein's mechanisms, like size inference, are inspired by Dex.

Ein programs are expressed in a core language similar to $\tilde{F}$ [23]. $\tilde{F}$ also revolves around comprehensions and indexing. These have been long explored [22], also e.g. with pull arrays [24] and in SAC [20, 21]. However, usually array languages have bespoke compilers, with low-level targets like LLVM (Dex) or C-like languages ($\tilde{F}$, SAC, Futhark [12]).

Uniqueness. The unique approach of Ein is in connecting the pointful and the point-free – by compiling the former to the latter, thus avoiding a full compiler to a low-level target. Existing pointful methods compatible with NumPy-like array libraries are very limited. In Python, what few approaches exist revolve around *Einstein summation*: starting from the `numpy.einsum` routine, through Tensor Comprehensions [25], finishing with the moderately successful `einops` [19]. However, `einops` is a limited "stringly-typed" DSL, which mostly allows expressing shape manipulations in an index-oriented fashion, e.g.:

```
einops.rearrange(x, "b h w c -> b c h w ()")
⇝ expand_dims(transpose(x, (0, 3, 1, 2)), 4)
```

Ein builds on this legacy by enabling **general pointful array programs in Python** for the first time, mirroring existing pointful array languages. This has many benefits. For instance, Ein admits type checking with standard tools (e.g. `mypy`), in contrast to NumPy, the API of which is too complicated. Furthermore, it has a more flexible type system, including arrays of *Pythonic* (idiomatic) record types, while array libraries tend to just offer `ndarrays` of primitives.

Axials, which form the connection between pointful and point-free array programs, are formulated in the language of applicatives [8, 16]. Recently introduced *named tensors* have some similarities to Axials, as they generalise arrays in a similar manner – by adding *named* axes besides *positional* ones [4]. However, to the author's knowledge no connection with pointful-style programs has been made before.

4 Language overview

Ein is a new purely functional and strongly-typed DSL embedded and fully implemented in Python. It follows the pointful array programming paradigm. A deep embedding is used – Ein programs are built up through its API in the `ein` Python library. Afterwards, we explicitly *evaluate* it. Evaluation first *compiles* the program, optimising it and *generating* array code. The generated code is then *executed* by an *execution backend*, with the default one calling NumPy routines.

To demonstrate Ein, we give various examples. These are valid Python, with `from ein import *` presumed.

4.1 Array comprehensions

Ein's key construct is the array comprehension, written like:

```
array(lambda i: f(i), size=n)
```

The `array` primitive constructs an array of a given size, which at index `i` is equal to `f(i)`, like the Python list comprehension `[f(i) for i in range(n)]`. For instance:

```
array(lambda i: i, size=5)
```

computes the array `[0, 1, 2, 3, 4]`. Similarly:

```
array(lambda i, j: 3*i + j, size=(2, 3))
```

equates `[[0, 1, 2], [3, 4, 5]]`.

Indexing is a first-class construct in Ein. Say we have vectors $\mathbf{u}$ and $\mathbf{v}$. Then their outer product $\mathbf{u}^T\mathbf{v}$ is given by:

```
A = array(lambda i, j: u[i] * v[j])
```

This example makes use of Ein's *size inference* – where the array `size` is omitted, it is inferred from places where an index is used to directly index into an array.

Ein evaluates programs by generating calls to NumPy. For the outer product, Ein would generate:

```
A = numpy.expand_dims(u, axis=1) * v
```

which computes the required quantity using broadcasting.

As a second example, consider $(A + A^T)/2$:

```
array(lambda i, j: (A[i, j] + A[j, i]) / 2)
```

which generates `(A + transpose(A, (1,0)) / 2`. Though we used `A` twice, Ein computes it just once in generated code.

Furthermore, Ein allows arbitrary index expressions. We can compute a *difference array* for a vector *a* like so:

```
array(
  lambda i: a[i + 1] - a[i],
  size=a.size(0) - 1
)
```

Here, the Ein compiler generates *slicing* – `a[1:] - a[:-1]`.

4.2 Folds

Ein's `fold` construct expresses iteration. It is written like so:

```
fold(init, lambda i, acc: f(i, acc), count=n)
```

Fold corresponds to the following Python `for` loop:

```
acc = init
for i in range(n): acc = f(i, acc)
return acc
```

A simple example of a `fold` is the function:

```
def fold_sum(a):
  return fold(0.0, lambda i, acc: acc + a[i])
```

Using `fold_sum`, we can finally express `pairwiseL1`:

```
array(lambda i, j:
  fold_sum(lambda k: abs(A[i, k] - A[j, k])))
```

4.3 Embedding

Since Ein is embedded in Python, we can easily define the distance `L1` as a function to separate concerns of `pairwiseL1`:

```
def L1(u, v):
  return fold_sum(lambda i: abs(u[i] - v[i]))
def pairwiseL1(A):
  return array(lambda i, j: L1(A[i], A[j]))
```

This functional pattern is particularly tricky in NumPy-like libraries, which do not offer a first-class, simple solution.

4.4 Evaluation

To evaluate Ein programs we call `eval`, and `numpy.ndarrays` may be passed in with `wrap` (similar to Accelerate [3]), e.g.:

```
u = wrap(numpy.array([1, 2, 3]))
v = wrap(numpy.array([-1, 1]))
w = array(lambda i, j: u[i] * v[j])
exp = [[-1, 1], [-2, 2], [-3, 3]]
numpy.testing.assert_allclose(w.eval(), exp)
```

An alternative to this is the `@function` decorator. Decorated functions can be called directly with array arguments.

4.5 Advanced features

Ein supports *record types*, allowing interleaving of Ein primitives and Python containers, and *associative reductions*. Using these, `argmin(a)` can be given as a *list homomorphism* [5]:

```
array(lambda i: {"val": a[i], "idx": i}).reduce(
  {"val": float("inf"), "idx": 0},
  lambda x, y: where(x["val"] < y["val"], x, y)
)["idx"]
```

5 Compiler

We shall now overview Ein's compiler, including our two intermediate languages for array programs – the pointful *Phi calculus* and the point-free *Yarr*. We then describe the structure of the *Axial applicative*, which formally connects pointful and point-free array programming. Thus, we describe Ein's code generation phase, translating Phi to Yarr.

5.1 Languages

Phi calculus – pointful source. Ein's underlying language is the *Phi calculus* (Figure 3). Phi's main feature are the *array comprehensions* Φ, which are the array introduction form. For instance, $\Phi\, i[3].\, i$ is the array $[0, 1, 2]$. The elimination form is indexing $e[e']$, such that $(\Phi\, i[5].\, 2 \cdot i)[4] = 2 \cdot 4 = 8$. Phi's scalar operators and folds correspond to those in Ein.

$$
\begin{array}{rll}
\kappa & ::= & \text{Float} \mid \text{Int} \mid \text{Bool} && \text{(scalars)} \\
\tau & ::= & \kappa \mid \tau \times \tau \mid \square\tau && \text{(scalars, pairs, vectors)} \\
e & ::= & \Phi\, i[e].\, e && \text{(array comprehension)} \\
& \mid & e[e] && \text{(indexing)} \\
& \mid & \sigma(e, \ldots, e) && \text{(scalar operator)} \\
& \mid & \text{fold}\, x[e]\, \text{init}\, x = e\, \text{by}\, e && \text{(indexed fold)} \\
& \mid & \text{size}_n\, e && \text{(size along axis } n \in \mathbb{N}\text{)} \\
& \mid & \langle e, e \rangle && \text{(pair construction)} \\
& \mid & \text{fst}\, e \quad \mid \quad \text{snd}\, e && \text{(pair projections)} \\
& \mid & \text{let}\, x = e\, \text{in}\, e && \text{(let-binding)} \\
& \mid & x \quad \mid \quad i \quad \mid \quad c && \text{(variable, index, constant)}
\end{array}
$$

Figure 3. Types τ and expressions e in the Phi calculus.

In Phi, the type of multidimensional arrays α is formed by scalars κ (the base case) and vectors of arrays $\square\alpha$. Phi also has pair types, which can be used unconstrained. Arrays of pairs are usually not supported in Python's array libraries, and to address this in Ein we apply a classic struct-of-arrays to array-of-structs program transformation [23].

A key design choice in Phi is the separation of identifiers into variables $\{x, y, z, \ldots\}$ and **indices** $\{i, j, k, \ldots\}$. Indices are only introduced by array comprehensions – this turns out to be vital. The typing judgement $\Gamma; \Delta \vdash e : \tau$ is non-standard owing to this, with environments Γ for variables and Δ for indices. Consider the comprehension typing rule:

$$\frac{\Gamma; \diamond \vdash e' : \text{Int} \qquad \Gamma; \Delta, i \vdash e : \tau}{\Gamma; \Delta \vdash \Phi\, i[e'].\, e : \square\tau}$$

We require that array sizes have no free indices, i.e. we type them under $\Delta = \diamond$. We use the same constraint for the iteration count in folds. This ensures beneficial *regularity* of parallelism and *rectangularity* of arrays. Hence, $\Phi\, i[5].\, \Phi\, j[i].\, i + j$ does not type, as it would be a jagged (non-rectangular) array. Other rules are standard, carrying through Γ and Δ.

Yarr – point-free target. Figure 4 gives the syntax of Yarr, which is our code generation target from Phi. In practice, we extend it with e.g.: reductions, indexing routines (like slicing), padding, and Einstein summations. Yarr strives to provide an interface through which various backends can be implemented, directly mirroring operations in NumPy – for instance, the elementwise operator performs broadcasting. Note the only significant similarity between Yarr and Phi is the indexed fold – array comprehensions are erased entirely.

5.2 Axials – pointful to point-free

Motivation. To motivate Axials, we appeal to Phi's *denotational semantics*. For a Phi term e, we have its denotation:

$$[\![e]\!] : \text{Env} \to \text{Value}$$

$$
\begin{array}{rll}
\tau' & ::= & \kappa \mid \square\tau' && \text{(arrays)} \\
\tau & ::= & \tau' \quad \mid \quad \tau' \times \tau' \quad \mid \quad \ldots && \text{(tuples of arrays)} \\
E & ::= & \text{range}(E) && \text{(naturals up to } n\text{)} \\
& \mid & \text{transpose}(E, (n, \ldots)) && \text{(permute axes)} \\
& \mid & \text{squeeze}(E, n) && \text{(remove 1-axis)} \\
& \mid & \text{unsqueeze}(E, n) && \text{(add 1-axis)} \\
& \mid & \text{repeat}(E, E, n) && \text{(repeat along axis)} \\
& \mid & \text{gather}(E, E, n) && \text{(indexing along axis)} \\
& \mid & \text{elementwise}_\sigma(E, \ldots) && \text{(elementwise op.)} \\
& \mid & \text{fold}\, x[E]\, \text{init}\, x = E\, \text{by}\, E && \text{(indexed fold)} \\
& \mid & \text{size}(E, n) && \text{(array size along axis)} \\
& \mid & \langle E, \ldots \rangle \quad \mid \quad \text{proj}_n(E) && \text{(tuples and projections)} \\
& \mid & \text{let}\, x = E\, \text{in}\, E && \text{(let-binding)} \\
& \mid & x \quad \mid \quad c && \text{(variables, constants)}
\end{array}
$$

Figure 4. Types τ and expressions E in Yarr; τ' are those (Phi) types which do not contain products.

Since Phi splits identifiers into variables and indices, we can similarly split the environment, and apply currying:

$$[\![e]\!] : \text{VarEnv} \to (\text{IndexEnv} \to \text{Value})$$

Observe that IndexEnv actually has a lot of structure. Since every index i is introduced in an array comprehension, it must range over $i \in [0, s_i)$, where s_i is the size of the range of i. Thus, IndexEnv corresponds to the **index space of an array**. If $\iota(e) = \{i, j, k \ldots\}$ are the free indices in e, then:

$$\text{dom IndexEnv} \simeq [0, s_i) \times [0, s_j) \times [0, s_k) \times \cdots$$

Through this idea we have shown pointful programs can be *partially evaluated* to remove the presence of indices. Axials encapsulate this in a composable, well-behaved manner.

Axial applicative. We define Axials by first setting that $\text{Axial}\, \alpha \equiv \text{List Index} \times \text{Array}\, \alpha$. We call the List the *axes* of the Axial, and take it to be a permutation of Index $(i, j, \ldots)$. The number of axes equals the rank of the Array *(values)*. We argue this structure is well-behaved by showing it is an *applicative* (a functional programming pattern with a categorical foundation [16]). Additionally, Axials are innately connected to pointful and point-free array programming, as well as common existing applicatives (`List` and `ZipList`).

To show Axials are applicatives, we define the following:

$$
\begin{aligned}
&\text{pure} : \alpha \to \text{Axial}\, \alpha \\
&\quad \text{lift} : (\alpha \to \beta \to \gamma) \to (\text{Axial}\, \alpha \to \text{Axial}\, \beta \to \text{Axial}\, \gamma)
\end{aligned}
$$

which satisfy the *applicative laws* (behave well under composition). We set the result of pure to have no axes, so

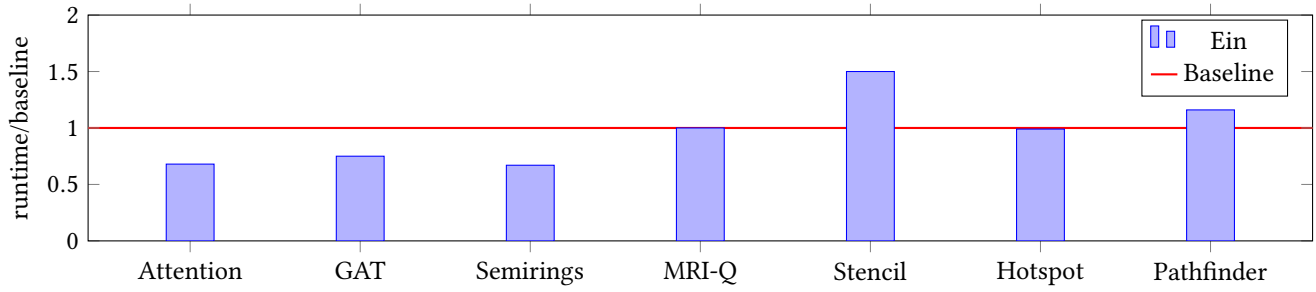

Figure 5. Ratio of the running time of Ein-generated NumPy programs, and the baseline code using NumPy directly. *Lower is better.* Large problem sizes are used (about 1 s) taking the minimum time across 5 runs.

pure $a = \langle [], a \rangle$. For lifting, we follow formulation of Naperian functors [8] (which generalise arrays):

$$\text{lift}\, f\, a\, b \sim \lambda\iota. f(a[\iota], b[\iota])$$

The idea is as follows. Say a has axes $\alpha = (i, j)$, and b has axes $\beta = (j, k)$. We *unify their axes*, producing $c = \text{lift}\, f\, a\, b$ with axes (i, j, k). Writing axial values in `monospace`:

$$\mathsf{c}[i, j, k] = f(\mathsf{a}[i, j], \mathsf{b}[j, k])$$

which is (almost) an instance of **broadcasting** f, as long as we perform *shape manipulation* to *align* `a` and `b`.

5.3 Code generation – Phi to Yarr

The compilation scheme effectively *lifts* Phi expressions into the Axial applicative, at which point they can be reified into Yarr. To this end, we introduce a transformation $\mathcal{A}[\![e]\!]$ mapping Phi terms e to an Axial of Yarr terms E:

$$\mathcal{A}[\![e]\!] = \langle \pi, E \rangle$$

where π (called *axes*) is a permutation of the free indices $\iota(e)$ of e. Thanks to the fact Axials form an applicative, this transformation $\mathcal{A}[\![e]\!]$ composes from $\mathcal{A}[\![e']\!]$ for subterms e' of e. We follow this semantic equivalence property:

$$\left[\!\!\left[\Phi\, \pi^{(1)}[s_{\pi^{(1)}}]. \ldots \Phi\, \pi^{(k)}[s_{\pi^{(k)}}]. e\right]\!\!\right] = [\![E]\!]$$

In essence, we require $\mathcal{A}$ produces a Yarr term E equivalent to the Phi term e wrapped in array comprehensions corresponding to each axis in $\pi = \left[\pi^{(1)}, \ldots, \pi^{(k)}\right]$. Thus, $\mathcal{A}[\![i]\!] = \langle [i], \text{range}(s_i) \rangle$, as $[\![\Phi\, i[s_i].\, i]\!] = [\![\text{range}(s_i)]\!]$. For a scalar operator $e = \sigma(e_1, e_2)$, we need to *align* the respective encodings of E_1 and E_2 ($\mathcal{A}[\![e_t]\!] = \langle \pi_t, E_t \rangle$). This is done by transpositions and unsqueezes, resulting in E_1' and E_2' so that:

$$\mathcal{A}[\![\sigma(e_1, e_2)]\!] = \left\langle \pi, \text{elementwise}_\sigma(E_1', E_2') \right\rangle$$

We have similar rules for the rest of Phi, preserving the Axial encoding. We thus obtain an efficient Yarr program with a high degree of data parallelism.

Besides generating Yarr code, Ein's compiler performs many important optimisations, e.g.: common subexpression elimination, loop-invariant code motion, synthesis of matrix multiplication calls and more efficient indexing operations.

6 Runtime

The result of Ein's code generation phase is a Yarr program, which is interpreted at runtime by the *NumPy backend*, calling appropriate NumPy routines. I further implemented PyTorch and JAX backends, allowing Ein to take advantage of their automatic differentiation and hardware acceleration.

Benchmarks. Ein's compilation target is unusual – our 'machine code' is NumPy, so we are limited by its performance. I evaluate how well Ein generates NumPy programs by benchmarking against *idiomatic* baseline NumPy implementations.

The benchmark cases are as follows:

- MRI-Q and Stencil from Parboil; Hotspot and Pathfinder from Rodinia. Based on Futhark implementations [10].
- Attention and GAT are deep learning programs translated from open-source PyTorch and JAX code.
- 'Semirings' is non-standard. The NumPy baseline is a standard $O(n^3)$ all-pairs shortest paths. In Ein we formulated a `Semiring` interface following Dolan [6], building abstractions difficult to produce in NumPy.

The benchmark results can be seen in Figure 5. Results are promising, with Ein's performance within [60%, 160%] of the baseline. Where Ein outperforms the baseline, it is generally thanks to avoiding allocations of temporary arrays. Performance deficits are especially due to backend overheads and missing advanced indexing strategies – particularly in Stencil. However, the Stencil baseline suffers from error-prone expressions like `A[1:-1, :-2, 1:-1]` (some 10 times). Hence, Ein offers better code clarity and ability to build abstractions, with a good trade-off on performance.

7 Conclusions

In this paper I have described Ein – a new pointful array language embedded in Python. Ein is compiled into calls in the point-free style, making efficient use of ubiquitous Python array libraries like NumPy, PyTorch and JAX. Thus, we leverage the immense existing work in the array programming model. I made the first such general approach to formally connecting pointful and point-free array programming, reconciling research and practice.

Acknowledgments

I am especially grateful to my project supervisor, Professor Alan Mycroft, for his time spent discussing this project. I would also like to thank Euan Ong for his many discussions on the topic of this work.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. TensorFlow: a system for large-scale machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*. 265–283.

[2] Robert Atkey, Sam Lindley, and Jeremy Yallop. 2009. Unembedding domain-specific languages. In *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*. 37–48.

[3] Manuel MT Chakravarty, Gabriele Keller, Sean Lee, Trevor L McDonell, and Vinod Grover. 2011. Accelerating Haskell array codes with multicore GPUs. In *Proceedings of the sixth workshop on Declarative aspects of multicore programming*. 3–14.

[4] David Chiang, Alexander M Rush, and Boaz Barak. 2022. Named Tensor Notation. *Transactions on Machine Learning Research* (2022).

[5] Murray Cole. 1993. *Parallel programming, list homomorphisms and the maximum segment sum problem.* Citeseer.

[6] Stephen Dolan. 2013. Fun with semirings: a functional pearl on the abuse of linear algebra. In *Proceedings of the 18th ACM SIGPLAN international conference on Functional programming*. 101–110.

[7] Roy Frostig, Matthew James Johnson, and Chris Leary. 2018. Compiling machine learning programs via high-level tracing. *Systems for Machine Learning* 4, 9 (2018).

[8] Jeremy Gibbons. 2016. Aplicative programming with naperian functors. In *Proceedings of the 1st International Workshop on Type-Driven Development*. 13–14.

[9] Jeremy Gibbons and Nicolas Wu. 2014. Folding domain-specific languages: deep and shallow embeddings (functional pearl). In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming*. 339–347.

[10] The Futhark Hackers. 2024. *Futhark Benchmarks.* https://github.com/diku-dk/futhark-benchmarks

[11] Charles R Harris, K Jarrod Millman, Stéfan J Van Der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J Smith, et al. 2020. Array programming with NumPy. *Nature* 585, 7825 (2020), 357–362.

[12] Troels Henriksen, Niels GW Serup, Martin Elsman, Fritz Henglein, and Cosmin E Oancea. 2017. Futhark: purely functional GPU-programming with nested parallelism and in-place array updates. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 556–571.

[13] Paul Hudak. 1996. Building domain-specific embedded languages. *ACM Computing Surveys (CSUR)* 28, 4es (1996), 196.

[14] Kenneth E Iverson. 1962. A programming language. In *Proceedings of the May 1-3, 1962, spring joint computer conference*. 345–351.

[15] Dougal Maclaurin, Alexey Radul, Matthew J Johnson, and Dimitrios Vytiniotis. 2019. Dex: array programming with typed indices. In *Program Transformations for ML Workshop at NeurIPS 2019*.

[16] Conor McBride and Ross Paterson. 2008. Applicative programming with effects. *Journal of functional programming* 18, 1 (2008), 1–13.

[17] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. PyTorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019).

[18] Adam Paszke, Daniel D Johnson, David Duvenaud, Dimitrios Vytiniotis, Alexey Radul, Matthew J Johnson, Jonathan Ragan-Kelley, and Dougal Maclaurin. 2021. Getting to the Point: index sets and parallelism-preserving autodiff for pointful array programming. *Proceedings of the ACM on Programming Languages* 5, ICFP (2021), 1–29.

[19] Alex Rogozhnikov. 2021. Einops: Clear and reliable tensor manipulations with einstein-like notation. In *International Conference on Learning Representations*.

[20] Sven-Bodo Scholz. 2003. Single Assignment C: efficient support for high-level array operations in a functional setting. *Journal of functional programming* 13, 6 (2003), 1005–1059.

[21] Sven-Bodo Scholz and Artjoms Šinkarovs. 2019. Tensor comprehensions in SaC. In *Proceedings of the 31st Symposium on Implementation and Application of Functional Languages*. 1–13.

[22] Amir Shaikhha, Andrew Fitzgibbon, Simon Peyton Jones, and Dimitrios Vytiniotis. 2017. Destination-passing style for efficient memory management. In *Proceedings of the 6th ACM SIGPLAN International Workshop on Functional High-Performance Computing*. 12–23.

[23] Amir Shaikhha, Andrew Fitzgibbon, Dimitrios Vytiniotis, and Simon Peyton Jones. 2019. Efficient differentiable programming in a functional array-processing language. *Proceedings of the ACM on Programming Languages* 3, ICFP (2019), 1–30.

[24] Bo Joel Svensson and Josef Svenningsson. 2014. Defunctionalizing push arrays. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Functional High-performance Computing*. 43–52.

[25] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. Tensor Comprehensions: Framework-agnostic high-performance machine learning abstractions. *arXiv preprint arXiv:1802.04730* (2018).