The Key to a Data Parallel Compiler

Aaron W. Hsu
Indiana University, USA
awhsu@indiana.edu

Abstract

We present a language-driven strategy for the construction of compilers that are inherently data-parallel in their design and implementation. Using an encoding of the inter-node relationships between nodes in an AST called a Node Coordinate Matrix, we demonstrate how an operator called the Key operator, that applies a function over groupings of array cells grouped and ordered by their keys, when used in conjunction with a Node Coordinate Matrix, permits arbitrary computation over sub-trees of an AST using purely data-parallel array programming techniques. We discuss the use of these techniques when applied to an experimental commercial compiler called Co-dfns, which is a fully data-parallel compiler developed using the techniques discussed here.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Compilers

Keywords APL, Parallel, Compilers, Key, Node Coordinates

1. Introduction

Compilers represent a peculiar class of tree/graph algorithms that greatly alter the underlying structure of the input graph into something very different in structure, but equivalent by some measure of interpretation. These algorithms are widely applicable, but have stubbornly resisted a general approach to parallelization. Most compilers are single-threaded or make very limited use of parallelism in an ad hoc way. The design and analysis of compilers almost universally deals with compilers as recursive traversals over ASTs. Such formulations usually include heavy reliance on branching, recursion, and sometimes very sophisticated control flow in practice. All of this contributes to make it difficult to effectively parallelize such algorithms onto architectures that are sensitive to branching and recursion, such as GPGPUs or highly vectorized CPUs.

There has been some success in efficiently executing parts of a compiler on the GPU, such as the parser [8], tokenization [4], and certain compiler analyses [21, 22]. However, a generalized framework or strategy for parallel compilation remains elusive. One of the major difficulties is the recursive and highly branch dependent nature of the algorithms. These present challenges that must be overcome in vector-centric architectures.

We present a language-driven strategy for creating an inherently parallel compiler. We select a set of well known data-parallel primitives, and then restrict program construction to the composition of these operations into functions, without any forms of non-linear control flow, such as branching, recursion, or pattern matching. Programs constructed in such a style are data-parallel by construction. A key problem to the construction of compilers in such a restricted environment is dealing with the inherently recursive and nested structure of the AST. By working over a linearized representation of the AST as a matrix, combined with a structure we call a Node Coordinate Matrix, and using data-parallel primitives, we are able to tame the recursive and nested nature of the AST into something that can be processed handily using only data-parallel, data-flow programs, without requiring branching or other forms of non-data parallel control flow. The methods are general and independent of the language being compiled.

In the following sections we describe the core methods and idioms surrounding the AST encoding, construction of the Node Coordinate Matrix, and the use of these structures in handling nested, recursive AST relationships. This enables us to perform arbitrary computation over arbitrary sub-trees of an AST selected by parent-child relationships in the manner often seen in compiler transformations. These ideas have been further implemented and tested through the implementation of a complete data-parallel by construction compiler, called Co-dfns, that compiles a lexically scoped, functionally oriented dialect of APL with nested functions [13, 14]. The compiler targets both the GPU and the CPU, and its core is implemented in this pure, restricted language that uses only function composition over data-parallel primitives.

Contributions

- We describe a method of computing over arbitrary sub-trees selected by their parent-child relationships in a data-parallel manner using the Key operator and Node Coordinates.
- We situation this technique into a broader, language-driven strategy for compiler construction that enables the development of parallel by construction compilers that are high-level in their implementation, exceptionally concise, and independent of the language being compiled.
- We provide analysis of these techniques as used in a commercial compiler project called Co-dfns and report on the results, including the overall architecture of the compiler, the uses of these techniques within the compiler, and the demonstration of these techniques applied to two specific compiler passes.

2. Notational Conventions

For space and convenience, we use a concise array notational convention to describe our techniques and approach. The notation is executable and well established in the array community (it is a limited subset of APL). This is in fact the same notation used within the Co-dfns compiler itself, whose source code is available online and provides a complete example of the use of both these algorithms and this notation in the large. All code examples are given in the following form:

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

Copyright is held by the owner/author(s). Publication rights licensed to ACM.

ARRAY'16, June 14, 2016, Santa Barbara, CA, USA
ACM. 978-1-4503-4384-8/16/06...$15.00
http://dx.doi.org/10.1145/2935323.2935331

```
      5×3+4 ⍝ Right to Left Evaluation
35
```

That is, we indent the expression by 6 spaces, followed by the value of the expression without indent. All expressions are evaluated right to left and all function application is infix; that is, a function application may apply a function (such as `+`) to one or two arguments, called monadic and dyadic application respectively, which must appear on the right and (optionally) on the left of the function name, as in the above example. Tables 2 to 4 provide a listing of all of the primitives used in the Co-dfns compiler, a selection of which are used here.

We separate out higher-order functions called operators, from functions that operate over arrays that we call simply functions. Operators take a function argument or two, and bind more strongly to the left than the right. An operator may take one or two operands, either on the left, or on the right and left, and will return a function. In the following example, we apply the reduction operator (`/`) to a function created using the composition operator (`∘`), as an example:

```
      +/1 2 3 4 5
15
      1+-(2+-(3+-(4+-(5))))
3
      1 +∘- 2 +∘- 3 +∘- 4 +∘- 5
3
      +∘-/1 2 3 4 5
3
```

We introduce notation as needed throughout the exposition, so the reader is encouraged to refer to the tables as necessary to recall particular operations.

3. Data Parallel Sub-tree Computation

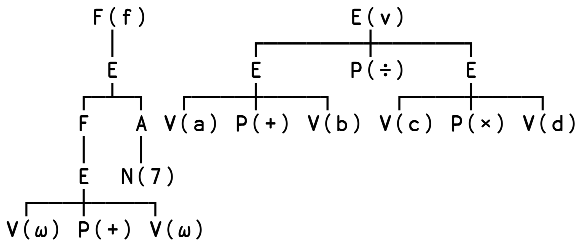
Computing over sub-trees normally involves recursing down the structure of the AST, identifying the root nodes of each sub-tree, and then dispatching to a handler for that sub-tree, which will continue the recursion at that point and return the modified tree, which will replace the previous sub-tree. More complicated transformations may involved moving the root nodes around in the tree or other large, distant structural modifications.

We divide the work of sub-tree computation into three basic phases: we maintain a node coordinate with each node that uniquely identifies it relative to others, we use these coordinates to select and group nodes for work as a single sub-tree, and then we operate over these groupings using the Key operator (`⌸`).

3.1 Encoding the AST

We represent the AST as a 3 column matrix with one row per node in the tree. The first column contains the inter-node relationships in the form of a depth vector. The second column is a vector of the node types, while the third contains the "value" of the node, such as the name in a variable.

We will use the following two running examples throughout. The `F` tree is an example nested function, while the `E` tree is an example nested expression.



The depth vectors for these trees we name `Fd` and `Ed`, respectively:

```
Fd←0 1 2 3 4 4 4 2 3
Ed←0 1 2 2 2 1 1 2 2 2
```

The node types we call `Ft` and `Et`:

```
Ft←'FEFEVPVAN'
Et←'EEVPVPEVPV'
```

And finally, we call the values vectors `Fv` and `Ev`:

```
Fv←'f' 0 0 0 '⍵' '+' '⍵' 0 7
Ev←'v' 0 'a' '+' 'b' '÷' 0 'c' '×' 'd'
```

We combine these to form the respective AST matrices. We write `A,B` to catenate arrays `A` and `B` along their last axes and `⍪A` to turn a vector into a 1-column matrix. Thus, the two AST matrices are given by the following expressions:

```
      Fd,Ft,⍪Fv     │       Ed,Et,⍪Ev
0 F f               │  0 E v
1 E 0               │  1 E 0
2 F 0               │  2 V a
3 E 0               │  2 P +
4 V ⍵               │  2 V b
4 P +               │  1 P ÷
4 V ⍵               │  1 E 0
2 A 0               │  2 V c
3 N 7               │  2 P ×
                    │  2 V d
```

The depth vector stores all edge information for the tree, but this information requires non-local access to utilize, such as traversing potentially the entire vector to determine the children of a specific node. Node Coordinates fix this issue.

3.2 Node Coordinates

Every node in an AST has a Node Coordinate, named because a coordinate is a precise location in a space. We can imagine all the nodes arranged inside some multi-dimensional space, leading to a specific set of node coordinates. Many such arrangements exist, of varying usefulness. In our case, any arrangement should allow us to answer the following questions about any two arbitrary nodes in a tree:

1. Are the nodes the same?
2. Are they siblings?
3. Does one appear "earlier" in the tree?
4. Are they at the same depth?
5. Is one an ancestor of another?

In short, we care about the relative position of each node in a tree relative to any other. We define a node coordinate as follows.

A node coordinate is a vector whose length is the depth of the tree. Its elements are natural numbers. The count of non-zero elements in the vector is equal to the depth of that node in the tree. All zero elements appear after the non-zero elements. That is, a coordinate is zero-padded on the right. When ordered lexicographically, the nodes for each coordinate appear in order according to a depth-first pre-order traversal of the tree. Each coordinate uniquely identifies a single node. Every ancestor's coordinate is a prefix of any child's coordinate, ignoring zeros. From the above it follows that every node is lexicographically greater than its left sibling and differs from it by exactly one non-zero element, and that this element is the last non-zero element in the coordinate.

3.2.1 Constructing Node Coordinates

We construct the Node Coordinate Matrix from the depth vector, thereby encoding the depth vector into a more useful format. The matrix has `N` rows and `D` columns, where `N` is the node count and `D` is the depth of the tree.

We write `f⌿A` to reduce the first axis of `A` using function `f`. Thus, `+⌿V` is the sum of the elements in vector `V`. The function `x⌈y` gives the maximum of its two arguments. We compute the depth of each tree as follows:

```
      1+⌈⌿Fd
5
      1+⌈⌿Ed
3
```

We can obtain the ordered sequence by writing `⍳n`:

```
      ⍳3
0 1 2
```

So the depths of all nodes that appear in the depth vectors is thus:

```
      ⍳1+⌈⌿Fd
0 1 2 3 4 5
      ⍳1+⌈⌿Ed
0 1 2
```

The function table or outer product of `f` over vectors `U` and `V` is written `U ∘.f V` giving a `U  V` shaped matrix as a result. Thus, `(⍳3)∘.×⍳3` gives a small multiplication table:

```
      (⍳3)∘.×⍳3
0 0 0
0 1 2
0 2 4
```

If we use `=` instead, we have a Boolean identity matrix:

```
      (⍳3)∘.=⍳3
1 0 0
0 1 0
0 0 1
```

If we use `∘.=` on the depth vector and its set of depths instead, we see an expanded Boolean representation of the depth vector:

```
      Fd∘.=⍳1+⌈⌿Fd
1 0 0 0 0
0 1 0 0 0
0 0 1 0 0
0 0 0 1 0
0 0 0 0 1
0 0 0 0 1
0 0 0 0 1
0 0 1 0 0
0 0 0 1 0
      Ed∘.=⍳1+⌈⌿Ed
1 0 0
0 1 0
0 0 1
0 0 1
0 0 1
0 1 0
0 1 0
0 0 1
0 0 1
0 0 1
```

These matrices let us see the nesting features of each tree more visually, but also suggest another step. We can compute a sum scan with `+⍀`, also called a prefix sum, along the first axis. Applying this function on the above matrices leads to an interesting result:

```
      +⍀Fd∘.=⍳1+⌈⌿Fd
1 0 0 0 0
1 1 0 0 0
1 1 1 0 0
1 1 1 1 0
1 1 1 1 1
1 1 1 1 2
1 1 1 1 3
1 1 2 1 3
1 1 2 2 3
      +⍀Ed∘.=⍳1+⌈⌿Ed
1 0 0
1 1 0
1 1 1
1 1 2
1 1 3
1 2 3
1 3 3
1 3 4
1 3 5
1 3 6
```

These matrices are lexicographically ordered, and each ancestor shares a common prefix with its descendants. They are also unique coordinates. Only the spurious digits at the end of each coordinate prevent these matrices from meeting all our requirements for valid node coordinates.

The expression `V f⍤¯1⊢M` applies `f` to corresponding elements of `V` and rows of `M`:

```
      3 3⍴⍳9
0 1 2
3 4 5
6 7 8
      (⍳3)+⍤¯1⊢3 3⍴⍳9
0 1  2
4 5  6
8 9 10
```

If `n↑V` takes the first `n` elements of `V`, then we can obtain coordinate matrices from the prefix sums by noting that the spurious digits all come after column `d+1` where `d` is the depth corresponding to that coordinate. The following gives a complete expression for computing a node coordinate matrix from a depth vector, shown using `Fd` and `Fe`:

```
      ⊢Fc←(1+Fd)↑⍤¯1⊢+⍀Fd∘.=⍳1+⌈⌿Fd
1 0 0 0 0
1 1 0 0 0
1 1 1 0 0
1 1 1 1 0
1 1 1 1 1
1 1 1 1 2
1 1 1 1 3
1 1 2 0 0
1 1 2 2 0
      ⊢Ec←(1+Ed)↑⍤¯1⊢+⍀Ed∘.=⍳1+⌈⌿Ed
1 0 0
1 1 0
1 1 1
1 1 2
1 1 3
```

```
1 2 0
1 3 0
1 3 4
1 3 5
1 3 6
```

A careful study of the definition of a node coordinate and the above construction should reveal why this works. Intuitively, we are creating a multi-dimensional space or a number system in which each digit place or dimension contains or circumscribes a smaller space in which are contained all the descendant nodes that appear lower in the tree. Each coordinate is a sort of special path through the tree encoded to have desirable properties relative to other paths.

3.2.2 Operations on Node Coordinates

The simplest operation over a node coordinate is to extract the depth of the node. The dyadic expression `C⍳0` finds the first occurrence of `0` in `C` and returns the index of that occurrence. Thus, we compute the depth of a node as follows:

```
      C←1 1 2 0 0
      ¯1+C⍳0
2
```

In many compiler passes, we primarily care about which nodes are ancestors of other nodes. To determine whether one node is a child of another, we compute whether one node is a prefix of the other, ignoring zeros. We write `(f g h)` to represent the composition of functions `f`, `g`, and `h` as a function train. Function trains obey the following equivalences, where `A` and `B` are arrays.

```
A(f g h)B ↔ (A f B) g (A h B)
A(0 f h)B ↔ 0 f (A h B) ⍝ Constant Case
A(f g)B   ↔ f (A g B)
```

With this, we write a function to compute whether a given coordinate prefixes another.

```
      P←1 1 0 0 0
      C(=∨0=⊢)P
1 1 1 1 1
      ∧⌿C(=∨0=⊢)P
1
```

The above determines whether `P` is an ancestor to `C`. The logical functions `=`, `∧`, and `∨` are all extended point-wise over arrays. The function `⊢` returns its right argument. The function `(=∨0=⊢)` reads as, "equal or a zero right argument." We take this point-wise operation and reduce it with `∧⌿` (For All). This pattern is a special case of inner product, written `f.g`. For example, `+.×` is matrix multiplication. We transform our reduction and Boolean function to a single predicate with the inner product operator:

```
      C∧.(=∨0=⊢)P
1
```

The use of inner product extends to cases where `C` or `P` are matrices. For our examples, we want to lift the functions and flatten expressions, so we will select the `F` nodes and `E` nodes, respectively, as our parent/ancestor nodes in `Fp` and `Ep`. Given a boolean vector `BV`, the expression `BV⌿M` selects rows from `M` based on the non-zero elements of `BV`.

```
      ⊢Fp←('F'=Ft)⌿Fc
1 0 0 0 0
1 1 1 0 0
      ⊢Ep←('E'=Et)⌿Ec
1 0 0
1 1 0
1 3 0
```

We could use `Fp` and `Ep` to compare against `Fc` and `Ec` to determine which nodes belong to which parent in `Fp` or `Ep`, but our prefix function returns 1 when `C≡P`. Instead, we drop the last non-zero element from each coordinate. This will permit matching against all ancestors, but not against itself. We use the depth vector and the Take (`↑`) function to take all but the last non-zero element. We extend the resulting array with an extra zero column to ensure we have a compatible shape.

```
      ⊢Fcp←(Fd↑⍤¯1⊢Fc),0
0 0 0 0 0
1 0 0 0 0
1 1 0 0 0
1 1 1 0 0
1 1 1 1 0
1 1 1 1 0
1 1 1 1 0
1 1 0 0 0
1 1 2 0 0
      ⊢Ecp←(Ed↑⍤¯1⊢Ec),0
0 0 0
1 0 0
1 1 0
1 1 0
1 1 0
1 0 0
1 0 0
1 3 0
1 3 0
1 3 0
```

We use `Fcp` and `Ecp` to determine the `F` and `E` ancestors for each node (`⍉A` transposes `A`):

```
      Fcp∧.(=∨0=⊢)⍉Fp
0 0
1 0
1 0
1 1
1 1
1 1
1 1
1 0
1 0
      Ecp∧.(=∨0=⊢)⍉Ep
0 0 0
1 0 0
1 1 0
1 1 0
1 1 0
1 0 0
1 0 0
1 0 1
1 0 1
1 0 1
```

In the above, the closest ancestor is the rightmost 1 in each row. We can extract the column number of the rightmost 1 by replacing each 1 with its column number and selecting the maximum of each row with `⌈/`. For space we will show only the `E` example. In the following, `V×⍤1⊢M` gives a matrix where each row is given by `V×i⌷M` where `i⌷M` is the ith row of `M`.

```
      (⍳3)×⍤1⊢Ecp∧.(=∨0=⊢)⍉Ep
0 0 0
0 0 0
0 1 0
```

```
0 1 0
0 1 0
0 0 0
0 0 0
0 0 2
0 0 2
0 0 2
      ⊢Ei←⌈/(⍳3)×⍤1⊢Ecp∧.(=∨0=⊢)⍉Ep
0 0 1 1 1 0 0 2 2 2
```

We use these column numbers to index into `Ep` to obtain an ancestor matrix where the node coordinate of the closest ancestor is given for each node, one per row. The expression `I⌷⍤0 2⊢M` gives a matrix, one row per index in `I`, consisting of rows of `M` selected by row indices in `I`.

```
      ⊢Ek←Ei⌷⍤0 2⊢Ep
1 0 0
1 0 0
1 1 0
1 1 0
1 1 0
1 0 0
1 0 0
1 3 0
1 3 0
1 3 0
```

At this point we have two values, `Ek` and `Fk`, which indicate the closest containing node that we care about for each node in the tree, using its node coordinate.

```
      Fi←⌈/(⍳2)×⍤1⊢Fcp∧.(=∨0=⊢)⍉Fp
      ⊢Fk←Fi⌷⍤0 2⊢Fp
1 0 0 0 0
1 0 0 0 0
1 0 0 0 0
1 1 1 0 0
1 1 1 0 0
1 1 1 0 0
1 1 1 0 0
1 0 0 0 0
1 0 0 0 0
```

We use these keys to compute over the AST, particularly for function lifting and expression flattening as demonstrated in the next section. The repeating pattern is to leverage node coordinates and array operations to do stackless reasoning about inter-node relationships. This pattern occurs often in our compiler and is particularly useful to enable straightforward data-parallel transformations over an AST.

3.3 The Key Operator

First introduced in the J language [15], the Key operator (written `⌸`) is central to our strategy for compilation. It allows us to use the Node Coordinates to their full effect. The expression `K f⌸ M` groups rows of `M` by their corresponding keys in `K` and computes `k f m` for each unique key `k` in `K` and the matrix `m` of rows of `M` with key `k`. The corresponding rows of matrices `K` and `M` form key-value pairs. Let's compute a histogram using Tally (`≢`) and Key (`⌸`):

```
      10⍴5
5 5 5 5 5 5 5 5 5 5
      ⊢X←?10⍴5
1 1 4 0 1 3 1 1 2 1
      ≢⍳5
5
      X(⊣,(≢⊢))⌸X
1 6
4 1
0 1
3 1
2 1
```

To understand a bit better how the Key operator applies its function, consider the function `{⍺ ⍵}` which returns the pair of its right and left arguments. If we apply it to the same value as above, we get the following:

```
      X{⍺ ⍵}⌸X
1  1 1 1 1 1 1
4  4
0  0
3  3
2  2
```

In our case, we use either `Fk` or `Ek` as our keys applied to the corresponding AST. We also will drop off the first row in each AST using `1↓` since this node "contains" everything. In a complete AST this is usually the Module boundary node which contains the entire set of functions and values in the module.

4. Case Studies

The following compiler passes are found in any compiler from a nested function, nested expression language to a language without nested functions or expressions. They were the chief motivating examples for developing the techniques presented above as a part of developing the Co-dfns compiler, and represent a challenging problem to data-parallel compilation without the above techniques, but fall out almost effortlessly once the above patterns are available.

4.1 Function Lifting

If we use the Key operator with `Fk`, we get the following (the monadic use of `↓` converts `Fc` from a matrix to a vector of vectors):

```
      (1↓Fk){⍺ ⍵}⌸1↓Fd,Ft,Fv,⍪↓Fc
┌─────────┬───────────────────┐
│1 0 0 0 0│┌─┬─┬─┬─────────┐  │
│         ││1│E│0│1 1 0 0 0│  │
│         │├─┼─┼─┼─────────┤  │
│         ││2│F│0│1 1 1 0 0│  │
│         │├─┼─┼─┼─────────┤  │
│         ││2│A│0│1 1 2 0 0│  │
│         │├─┼─┼─┼─────────┤  │
│         ││3│N│7│1 1 2 2 0│  │
│         │└─┴─┴─┴─────────┘  │
├─────────┼───────────────────┤
│1 1 1 0 0│┌─┬─┬─┬─────────┐  │
│         ││3│E│0│1 1 1 1 0│  │
│         │├─┼─┼─┼─────────┤  │
│         ││4│V│⍵│1 1 1 1 1│  │
│         │├─┼─┼─┼─────────┤  │
│         ││4│P│+│1 1 1 1 2│  │
│         │├─┼─┼─┼─────────┤  │
│         ││4│V│⍵│1 1 1 1 3│  │
│         │└─┴─┴─┴─────────┘  │
└─────────┴───────────────────┘
```

Notice that we have now grouped all of the relevant parts of the tree according to which nodes would appear in their respective functions after lifting. Refer to the original lifting example in the introduction to verify this. Indeed, the second row in the above example

Pass	Description	Core Traversal Pattern	Uses coordinates?
Record Node Coordinates	Adds a new field to each node containing that node's node coordinate.	Outer Product/Scan	Yes
Record Function Depths	Adds a new field to each node recording how many functions surround the node.	Inner Product	Yes
Drop Unnamed Functions	Eliminates some code that will not be evaluated at the top-level.	Filter	No
Drop Unreachable Code	Eliminates some unreachable code.	Filter/Inner Product	Yes
Lift Functions	Moves all functions to the top-level.	Key	Yes
Drop Redundant Nodes	Eliminates unnecessary nodes/nesting.	Filter	No
Flatten Expressions	Removes nesting from expressions.	Key	Yes
Compress Atomic Nodes	Atomizes nested atom nodes.	Amend	No
Propagate Constants	Inlines all references to literal values.	Amend/Rank	Yes
Fold Constants	Converts constant expressions to literals.	Amend	No
Compress Expressions	Converts expression sub-trees into single nodes.	Amend	No
Record Final Return Value	Records the value returned by each function.	Key	No
Normalize Values Field	Normalizes the shape and size of the values field.	Amend	No
Lift Type-checking	Infers some type information at compile time.	Power Limit/Rank	No
Allocate Value Slots	Does a form of frame allocation for variables.	Key	No
Anchor Variables	Resolves lexically scoped variables.	Key	Yes
Record Live Variables	Records the variables that are live at each point of execution.	Key	No
Fuse Scalar Loops	Identifies Fusion opportunities and fuses expressions.	Key	No
Type Specialization	Specializes each function for a series of potential inputs.	Key	No

Table 1. A listing of some compiler passes in the Co-dfns compiler and their relationship with the Key operator and associated tree computation techniques

shows the internal function complete and ready to name. Each element in the second column corresponds to the body of one of our lifted functions. In the case of the first function, the outer function, we have a spurious function node in the body. This is intentional. When we lift these functions, we will replace each spurious function node with a variable node referring to the function's generated name.

Each of these function bodies has a specific coordinate associated with it. Because these coordinates are uniquely identifying, we can use these as input into a name generator to generate names that we know are unique for each function body. Furthermore, because we retain this information in the corresponding function nodes that appear in the body of each function to be lifted, we know exactly what name that function has been given, and we can replace the function node with a variable node referencing that name instead, without referring to any state outside of the immediate information given to the function lifter. Indeed, each row in the above matrix represents a function lifting task that can be completed without any additional information. That is, there are no dependencies between rows to perform lifting. This gives us a straightforward parallel execution of function lifting.

The final task to complete function lifting of each function body before lifting is to shift the depths in the depth vectors to correspond to those of a function lifted to the top-level and to attach a function node to the top of each of the bodies. At that point, we simply recombine all of the newly created functions into a single top level.

This function lifting works even with lexically scoped functions, as is the case for the Co-dfns compiler. We can do this because the node coordinates maintain the lexical information for each variable reference, enabling us to construct the appropriate lexical binding for each variable much later than would normally be done in a compiler. In other words, the above lifting operation loses no information that would prevent us from handling lexical scoping, but decouples lexical scoping from the act of lifting functions.

4.2 Expression Flattening

If we use `Ek` as the key for the expression example, we get the following:

```
      (1↓Ek){α ω}⌸1↓Ed,Et,Ev,⍪↓Ec
┌─────┬─────────────┐
│1 0 0│┌─┬─┬─┬─────┐│
│     ││1│E│0│1 1 0││
│     │├─┼─┼─┼─────┤│
│     ││1│P│÷│1 2 0││
│     │├─┼─┼─┼─────┤│
│     ││1│E│0│1 3 0││
│     │└─┴─┴─┴─────┘│
├─────┼─────────────┤
│1 1 0│┌─┬─┬─┬─────┐│
│     ││2│V│a│1 1 1││
│     │├─┼─┼─┼─────┤│
│     ││2│P│+│1 1 2││
│     │├─┼─┼─┼─────┤│
│     ││2│V│b│1 1 3││
│     │└─┴─┴─┴─────┘│
├─────┼─────────────┤
│1 3 0│┌─┬─┬─┬─────┐│
│     ││2│V│c│1 3 4││
│     │├─┼─┼─┼─────┤│
│     ││2│P│×│1 3 5││
│     │├─┼─┼─┼─────┤│
│     ││2│V│d│1 3 6││
│     │└─┴─┴─┴─────┘│
└─────┴─────────────┘
```

Again, we can see immediately that we have grouped each set of nodes according to the expressions that are to be lifted. Just as in the case of function lifting, we can adjust the depths of each expression

to the correct depth and we can replace each expression node with a variable reference based on that node's coordinate. Each expression can be given a unique name based on its coordinate. A later compiler pass can reduce these names down to the minimum actually required to represent the expression if desired.

The only extra issue involved here is to ensure that the order of evaluation matches. In our case, we are assuming that the order of evaluation is right to left, which means that the above order is actually backwards of the desired order. During recombination, we simply reverse these orders and this fixes that problem. More work would be required to take into consideration a specific precedence hierarchy.

5. The Co-dfns Compiler and Other Passes

The Co-dfns compiler is an experimental commercial compiler funded by Dyalog, Ltd. [13, 14] It is intended to provide increased scaling and performance for APL programmers using the dfns syntax (lexically scoped, functionally oriented APL). These techniques form one of the foundational elements to the architecture of the compiler, which is built in the style of a Nanopass [20] compiler, with very small passes chained together through composition, with the added requirements that each compiler pass must be fully data parallel.

The compiler produces good code that performs closely with hand-written C code for the same programs on the GPU and CPU. It produces platform independent code, meaning that the compiler is able to target either the GPU or CPU from the same program source and produce reasonable performance on both.

The compiler itself is composed of three main parts: the core compiler, the parser, and the code generator. The code generator includes the runtime library of the compiler. Parsing APL code in parallel is a well-studied problem. The compiler currently uses parsing combinators to do its parsing, though we plan to implement the Two-by-Two parser given in the literature [8]. The code generator itself is a simple parallel map over each node, as each node may be generated independent from the other nodes, leading to a fully parallel generator, though there are some branching elements in the generator that still remain, currently.

The core of the Co-dfns compiler is written entirely using function trains and parallel operations in the style described above. We use these techniques extensively throughout the compiler, and the code is publicly available at our repository. The core of the compiler, including some of the large tables comes in at around 250 lines of code. Without the tables, the core logic comes in at around 150 lines of code.

The entire compiler, together with its runtime libraries, parser, generator, and core, comes in at around 1500 lines of commented code.

Table 1 contains a selection of passes from the Co-dfns compiler and breaks down their features based on how they traverse the tree and whether or not they use the Key operator or Node Coordinates. The traversal pattern indicates the primary operator used to traverse the AST and perform the transformation, most of which are primitives; the Amend pattern is a "replacement" pattern where specific nodes are replaced in the AST with other nodes, and can be thought of as a slightly modified filter and map. The table orders the passes with front-end passes higher in the table and passes further down the compiler chain at the bottom of the table. These passes give a good idea of how the Key operator plays a part in the complete compiler as well as how Node Coordinates help to deal with complex sub-tree selections.

We make a few notes on the table and the analysis. The Key operator can be used with or without the node coordinates structure. In cases where the AST is sufficiently flat and arranged in a sufficiently convenient order, the grouping operations are very simple. In these cases, the Key operator amounts to a map operation over some selected sub-trees selected at a specific depth. These occur later in the compiler where the reader will note appearances of the Key operator without the use of Node Coordinates.

Likewise, node coordinates find their place early on in the compiler where it is necessary to deal with the more highly nested AST. They are useful for identifying this information regardless of whether the Key operator is used to handle the grouping or not. Thus passes like "Drop Unreachable Code" that remove whole sub-trees from the AST may use the Node Coordinates to do the grouping once the correct parent node has been identified.

In short, the goal of the structure and design of the compiler passes given in Table 1 is to utilize Node Coordinates to remove as quickly as possible the nested structure of the AST into a flatter form that does not require complex analysis to traverse.

6. Future Work

The current compiler represents a non-trivial instance of the successful application of these techniques, but it focuses on an untyped, functionally oriented language. The type inference in the current version of the compiler is somewhat naïve. More work is required to expand the type inferencer to include more sophisticated type inference. The compiler does not currently handle errors such as signals or exceptions, nor does it implement some more sophisticated compiler optimizations. Further work remains to implement these techniques on an imperative and object-oriented language instead of a functional one. It would also be educational to implement a version for a lazy language.

We do not have a large scale understanding of the performance characteristics of these techniques in real life outside of daily use of the compiler. Further analysis is required to understand what sort of compiler optimizations and analyses are necessary to compile the Co-dfns compiler efficiently for the GPU and CPU. We currently lack a good comparison against existing techniques both for the compiler design and the generated code. Some existing compilers are able to compile variants of APL to target the GPU and the CPU, but there are subtle differences that require care when comparing generated code, and the architecture of these compilers is often quite different. Comparing the relative performance both of the compiler designs (human factors and execution speed) and generated code is important to understanding these techniques more fully.

While we have found this method of compiler construction to be very compact and to permit working with a data-parallel compiler as easily or nearly as easily as with a standard Nanopass compiler, we have not done extensive studies to determine just how comparable in programmability, ease of maintenance, or extensibility these techniques are with established compiler construction methods, such as OOP Visitors, Nanopass, or the type-directed functional style.

7. Related Work

Iverson introduced the idea of the Rank operator while at Sharp Associates [17] as a part of "Rationalized APL." The rank operator (⍤) used here derived from the J language, a continuation of the rationalized APL idea, with many ideas such as rank percolating back into APL implementations. [1, 16] The J programming language [15] was the first practical, general-purpose programming language to introduce the Key operator as a primitive operator with the presumption of its general usefulness.

Bernecky [5] argues that the increased programmability and desirable human factors of APL-style array programming can be implemented with competitive performance relative to more traditional techniques. This suggests that programs written in this highly abstract style may not suffer the performance gap traditionally assumed to go with their desirable features. This work is bolstered by

previous work on the high-performance implementation of array-oriented languages. [3, 9–11, 18, 19, 23]

Fritz Henglein demonstrated a class of operations, called discriminators, of which the Key operator is a member [12] , namely, a discriminator performs the same grouping computation as Key, but does not apply a function over these groups with their keys. Henglein provides a linear implementation of these operations.

The EigenCFA effort [22] demonstrated significant performance improvements of a 0-CFA flow analysis by utilizing similar techniques to those demonstrated here. In particular, encoding the AST and using accessor functions have a very similar feel to the node coordinates and AST encoding given here, though they have a different formulation and spend considerable effort understanding the trade-offs of performance associated with the different encodings, whereas the encodings here were chosen for their clarity and directness, rather than their performance.

Mendez-Lojo, et al. implemented a GPU version of Inclusion-based Points-to Analysis [21] that also focuses on adapting data structures and algorithms to efficiently execute on the GPU. In particular, they use similar techniques of prefix sums and sorts to achieve some of their adaptation to the GPU, Additionally, they have clever and efficient methods of representing graphs on the GPU which enable dynamic rewriting of the graph.

The APEX compiler [2] developed vectorized approaches to handling certain analyses to compile traditional APL, including a SIMD tokenizer [4]. It also uses a matrix format to represent the AST. Traditional APL did not have nested function definitions, however, and thus the APEX compiler does not have any specific approaches to dealing with function lifting.

Timothy Budd implemented a compiler [6, 7] for APL which targeted vector processors as well as C. Budd provided thoughts and some ideas on how the compiler might be implemented in parallel as well.

J. D. Bunda and J. A. Gerth presented a method for doing table driven parsing of APL which suggested a parallel optimization for parsing, but did not elucidate the algorithm [8].

8. Conclusion

We have derived a method of performing computation over subtrees selected on the basis of inter-node properties through the use of the Key (⌸) operator and node coordinates, which enable local computation of these inter-node properties. This method is both general and direct, and when combined with traditional and more mundane array programming, suffices to implement the complete core of a compiler, modulo parsing and code generation. The method requires no special operations or unique special casing primitives in the language. Moreover, it is strictly data-parallel and data-flow, without any complex control flow, which results in exceptionally concise code. These methods permit a general, high-level approach to constructing a compiler that is parallel by construction by constricting the language to only a data parallel subset of a normal array language.

We have demonstrated the technique and the core insights behind the data structures involved. It presents a solution to a very old and traditional problem in a very uncommon light, by eschewing the common practices that underlie every other significant and general solution found in modern compilers today and replacing them with an entirely different paradigm centered on parallelism and aggregate operations.

References

[1] R. Bernecky. An introduction to function rank. *ACM SIGAPL APL Quote Quad*, 18(2):39–43, 1987.

[2] R. Bernecky. Apex: The apl parallel executor. 1997.

[3] R. Bernecky. Reducing computational complexity with array predicates. *ACM SIGAPL APL Quote Quad*, 29(3):39–43, 1999.

[4] R. Bernecky. An spmd/simd parallel tokenizer for apl. In *Proceedings of the 2003 conference on APL: stretching the mind*, pages 21–32. ACM, 2003.

[5] R. Bernecky and S.-B. Scholz. Abstract expressionism for parallel performance. In *Proceedings of the 2Nd ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, ARRAY 2015, pages 54–59, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3584-3. doi: `10.1145/2774959.2774962`. URL `http://doi.acm.org/10.1145/2774959.2774962`.

[6] T. Budd. *An APL compiler*. Springer Science & Business Media, 2012.

[7] T. A. Budd. An apl compiler for a vector processor. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 6(3):297–313, 1984.

[8] J. Bunda and J. Gerth. Apl two by two-syntax analysis by pairwise reduction. In *ACM SIGAPL APL Quote Quad*, volume 14, pages 85–94. ACM, 1984.

[9] W.-M. Ching. Automatic parallelization of apl-style programs. In *ACM SIGAPL APL Quote Quad*, volume 20, pages 76–80. ACM, 1990.

[10] W. M. Ching and A. Katz. An experimental apl compiler for a distributed memory parallel machine. In *Proceedings of the 1994 ACM/IEEE conference on Supercomputing*, pages 59–68. IEEE Computer Society Press, 1994.

[11] W.-M. Ching, P. Carini, and D.-C. Ju. A primitive-based strategy for producing efficient code for very high level programs. *Computer languages*, 19(1):41–50, 1993.

[12] F. Henglein and R. Hinze. Sorting and Searching by Distribution: From Generic Discrimination to Generic Tries. In *Programming Languages and Systems*, pages 315–332. Springer, Dec. 2013.

[13] A. W. Hsu. Co-dfns: Ancient language, modern compiler. In *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, page 62. ACM, 2014.

[14] A. W. Hsu. Accelerating information experts through compiler design. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, pages 37–42. ACM, 2015.

[15] R. Hui. Essays/key. URL `http://www.jsoftware.com/jwiki/Essays/Key`.

[16] R. K. Hui. Rank and uniformity. In *ACM SIGAPL APL Quote Quad*, volume 25, pages 83–90. ACM, 1995.

[17] K. E. Iverson. *Rationalized APL*. IP Sharp Associates, 1983.

[18] D.-C. Ju and W.-M. Ching. Exploitation of apl data parallelism on a shared-memory mimd machine. In *ACM SIGPLAN Notices*, volume 26, pages 61–72. ACM, 1991.

[19] D.-c. Ju, W.-M. Ching, and C.-l. Wu. On performance and space usage improvements for parallelized compiled apl code. *ACM SIGAPL APL Quote Quad*, 21(4):234–243, 1991.

[20] A. W. Keep and R. K. Dybvig. A nanopass framework for commercial compiler development. In *ACM SIGPLAN Notices*, volume 48, pages 343–350. ACM, 2013.

[21] M. Mendez-Lojo, M. Burtscher, and K. Pingali. A gpu implementation of inclusion-based points-to analysis. *ACM SIGPLAN Notices*, 47(8): 107–116, 2012.

[22] T. Prabhu, S. Ramalingam, M. Might, and M. Hall. Eigencfa: accelerating flow analysis with gpus. *ACM SIGPLAN Notices*, 46(1):511–522, 2011.

[23] W. Schwarz. Acorn run-time system for the cm-2. In *Arrays, Functional Languages, and Parallel Systems*, pages 35–57. Springer, 1991.

Symbol	Monadic	Dyadic	Symbol	Monadic	Dyadic
		Scalar Functions			
+	Identity	Plus (Add)	~	Not	
−	Negative	Minus (Subtract)	?	Roll	
×	Direction (Signum)	Times (Multiply)	∧		And
÷	Reciprocal	Divide	∨		Or
|	Magnitude	Residue (Modulo)	⍲		Nand
⌊	Floor	Minimum	⍱		Nor
⌈	Ceiling	Maximum	<		Less
*	Exponential	Power	≤		Less Or Equal
⍟	Natural Logarithm	Logarithm	=		Equal
○	Pi Times	Circular (Trigonometric)	≥		Greater Or Equal
!	Factorial	Binomial	>		Greater
≠		Not Equal			
	Selection Mixed Functions			Structural Mixed Functions	
⊃	Disclose	Pick	⍴	Reshape	
↑		Take	,	Ravel	Catenate/Laminate
↓		Drop	⍪	Table	Catenate First/Laminate
/		Replicate	⌽	Reverse	Rotate
⌿		Replicate First	⊖	Reverse First	Rotate First
\		Expand	⍉	Transpose	Transpose
⍀		Expand First	↑	Mix	
~		Without (Excluding)	↓	Split	
∩		Intersection	⊂	Enclose	Partitioned Enclose
∪	Unique	Union	∊	Enlist	
⊣	Same	Left			
⊢	Identity	Right			
	Selector Mixed Functions			Miscellaneous Mixed Functions	
⍳	Index Generator	Index Of	⍴	Shape	
∊		Membership	≡	Depth	Match
⍋	Grade Up	Grade Up	≢	Tally	Not Match
⍒	Grade Down	Grade Down	⍎	Execute	Execute
?		Deal	⍕	Format	Format
⍷		Find	⊥		Decode (Base)
			⊤		Encode (Representation)
			⌹	Matrix Divide	Matrix Inverse

Table 2. Primitive Functions

Symbol	Name	Description
⍨	Commute	Swaps arguments or distributes right argument to both sides
¨	Each	Applies its operand point-wise over the left/right arguments
/	Reduce	Reduce along the last axis
⌿	Reduce First	Reduce along the first axis
\	Scan	Scan along the last axis
⍀	Scan First	Scan along the first axis
⌸	Key	Apply operand once for each sub-array grouped by key

Table 3. Primitive Monadic/Unary Operators, each takes a single left operand and describes a function operating over one or two arguments

Symbol	Name	Description
∘	Compose	Composes two operands as in traditional mathematics
.	Inner Product	Inner product operation, e.g. +.× for matrix multiplication
∘.	Outer Product	Cartesian product or "function table"
⍣	Power	Iteration, Limited use only
⍤	Rank	Apply a function along cells of an array

Table 4. Primitive Dyadic/Binary Operators, each takes a left and right operand and describes a function operating over one or two arguments