The Key to a Performant Data Parallel Compiler

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 Path 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 Path 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, a fully data-parallel compiler developed using the techniques discussed here. On transformation heavy operations such as expression flattening or function lifting, these techniques, when compared against pure and effectful recursive implementations, outperform the CPU-based algorithms by a factor of 10 - 58 on the GPU.

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

General Terms Algorithms, Performance, Design, Languages

Keywords APL, Parallel, Compilers, Key, Paths

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 semantic 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. 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 vectorised CPUs.

There has been some success in efficiently executing parts of a compiler on the GPU, such as parsing [8], tokenization [4], and certain compiler analyses [22, 23]. 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 Path 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 transformation or computation.

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. We extend and improve on prior work [15] by simplifying the core expressions and examining specific case studies in greater detail than previously done. We extend the methodology with performance-centered optimizations of the Path Matrix and benchmark a transformation heavy expression lifting algorithm that is found almost universally in compilers. This benchmark shows performance speedups of up to a factor of 58 when compared against pure, functionally oriented algorithms.

Contributions

- We expound on a method [15] of computing over arbitrary sub-trees selected by their parent-child relationships in a data-parallel manner using the Key operator and a Path Matrix.
- We situate 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 in full detail.
- We compare the performance of the Key-based algorithms against traditional recursive methods for the task of lifting/flattening expressions, such as commonly done for function lifting and expression flattening. Our results demonstrate consistent speedups over the traditional methods.

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:

```
      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. The appendix provides a listing of all of the primitives used in the Co-dfns compiler, a selection of which are used here.

We differentiate 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 appendix as necessary to recall particular operations.

3. Data Parallel Sub-tree Computation

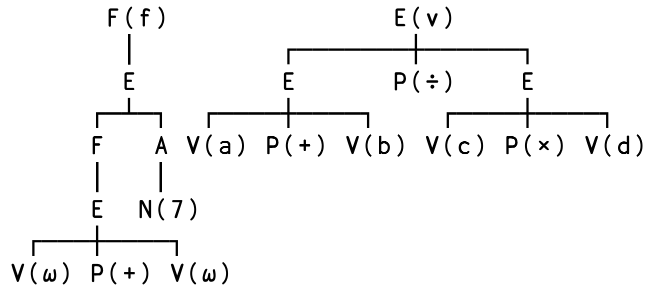
Computing over sub-trees normally involves recursing down the structure of the AST, identifying the root nodes of each sub-tree, and then dispatching to a handler for that sub-tree, which will continue the recursion at that point and return the modified tree, which will replace the previous sub-tree. More complicated transformations may involve 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 Path Vector with each node that uniquely identifies it relative to others, we use these Paths 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 'ω' '+' 'ω' 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 the last axis 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 ω            │  2 V b
4 P +            │  1 P ÷
4 V ω            │  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. Path Vectors fix this issue.

3.2 Path Matrix

Every node in an AST has a Path Vector, together forming a matrix. We can imagine all the nodes arranged inside some multi-dimensional space, leading to a specific set of Paths or 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 path vector as 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 path is zero-padded on the right. When ordered lexicographically, the nodes for each path appear in order according to a depth-first pre-order traversal of the tree. Each path uniquely identifies a single node. Every ancestor's path is a prefix of any child's path, ignoring zeros. From the above it follows that every node is lexicographically greater than its left sibling and differs only in the last non-zero element.

3.2.1 Constructing the Path Matrix

We construct the Path 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  │       1+⌈⌿Ed
5             │  3
```

We can obtain an 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 │       ⍳1+⌈⌿Ed
0 1 2 3 4 5   │  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 │       ⍳3
0 0 0           │  0 1 2
0 1 2           │       ⍴⍳3
0 2 4           │  3
```

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:

```
   Fb←Fd∘.=⍳1+⌈⌿Fd │  Eb←Ed∘.=⍳1+⌈⌿Ed
1 0 0 0 0          │ 1 0 0
0 1 0 0 0          │ 0 1 0
0 0 1 0 0          │ 0 0 1
0 0 0 1 0          │ 0 0 1
0 0 0 0 1          │ 0 0 1
0 0 0 0 1          │ 0 1 0
0 0 0 0 1          │ 0 1 0
0 0 1 0 0          │ 0 0 1
0 0 0 1 0          │ 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:

```
      +⍀Fb     │      +⍀Fb
1 0 0 0 0      │ 1 0 0
1 1 0 0 0      │ 1 1 0
1 1 1 0 0      │ 1 1 1
1 1 1 1 0      │ 1 1 2
1 1 1 1 1      │ 1 1 3
1 1 1 1 2      │ 1 2 3
1 1 1 1 3      │ 1 3 3
1 1 2 1 3      │ 1 3 4
1 1 2 2 3      │ 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. We generate a mask to remove these spurious digits by using a different outer product on our original depth vector:

```
      Fm←Fd∘.≥⍳1+⌈⌿Fd │   Em←Ed∘.≥⍳1+⌈⌿Ed
1 0 0 0 0             │   1 0 0
1 1 0 0 0             │   1 1 0
1 1 1 0 0             │   1 1 1
1 1 1 1 0             │   1 1 1
1 1 1 1 1             │   1 1 1
1 1 1 1 1             │   1 1 0
1 1 1 1 1             │   1 1 0
1 1 1 0 0             │   1 1 1
1 1 1 1 0             │   1 1 1
                      │   1 1 1
```

We use this mask with the `×` function to get a complete expression for computing a path matrix from a depth vector, shown using `Fd` and `Fe`:

```
        Fc←Fm×+⍀Fb │       Ec←Fm×+⍀Fb
1 0 0 0 0          │ 1 0 0
1 1 0 0 0          │ 1 1 0
1 1 1 0 0          │ 1 1 1
1 1 1 1 0          │ 1 1 2
1 1 1 1 1          │ 1 1 3
1 1 1 1 2          │ 1 2 0
1 1 1 1 3          │ 1 3 0
1 1 2 0 0          │ 1 3 4
1 1 2 2 0          │ 1 3 5
                   │ 1 3 6
```

A careful study of the definition of a path vector 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 vector is a path through the tree encoded to have desirable properties relative to other paths.

Note that there are other valid path matrix encodings. The one above is used throughout most of the rest of this paper because of its simplicity and easy of exposition. We use it to demonstrate the full generality of the structure. However, certain more complex matrix encodings permit certain operations at much lower cost while preserving the full generality.

3.2.2 Operations on Path Vectors

The simplest operation over a Path Vector 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 path vector 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
      Ep←('E'=Et)⌿Ec
      Fp     |      Ep
1 0 0 0 0    |  1 0 0
1 1 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 path. This will permit matching against all ancestors, but not against itself. We use the same masking technique as above, but adjust the comparison to use > instead of ≥ to leave off the last non-zero element of Fc and Ec:

```
      Fcp←Fc×Fd∘.>⍳1+⌈⌿Fd
      Ecp←Ec×Ed∘.>⍳1+⌈⌿Ed
      Fcp    |      Ecp
0 0 0 0 0    |  0 0 0
1 0 0 0 0    |  1 0 0
1 1 0 0 0    |  1 1 0
1 1 1 0 0    |  1 1 0
1 1 1 1 0    |  1 1 0
1 1 1 1 0    |  1 0 0
1 1 1 1 0    |  1 0 0
1 1 0 0 0    |  1 3 0
1 1 2 0 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   |    Ecp∧.(=∨0=⊢)⍉Ep
0 0                    | 0 0 0
1 0                    | 1 0 0
1 0                    | 1 1 0
1 1                    | 1 1 0
1 1                    | 1 1 0
1 1                    | 1 0 0
1 1                    | 1 0 0
1 0                    | 1 0 1
1 0                    | 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 ⌈.×. The function ≢ gives the number of rows of its argument.

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

```
      ⊢Fi←(Fcp∧.(=∨0=⊢)⍉Fp)⌈.×⍳≢Fp
0 0 0 1 1 1 1 0 0
```

We use these column numbers to index into Ep to obtain an ancestor matrix where the path vector of the closest ancestor is given for each node, one per row.

```
      Fk←Fp[Fi;]     |        Ek←Ep[Ei;]
1 0 0 0 0            |     1 0 0
1 0 0 0 0            |     1 0 0
1 0 0 0 0            |     1 1 0
1 1 1 0 0            |     1 1 0
1 1 1 0 0            |     1 1 0
1 1 1 0 0            |     1 0 0
1 1 1 0 0            |     1 0 0
1 0 0 0 0            |     1 3 0
1 0 0 0 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 path vector. 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 path vectors 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 [16], the Key operator (written ⌸) is central to our strategy for compilation. It allows us to use the path vectors 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{α ω}⌸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.

Each of the following passes operates over an AST represented as the catenation of the field vectors above including the Path Matrix.

```
Fast←Fd,Ft,Fv,⍪↓Fc
East←Ed,Et,Ev,⍪↓Ec
```

The function `↓` converts the path matrices from matrices to vectors of vectors, so that the coordinates can be catenated to the other fields, leading to a 4-column matrix for the ASTs, whose fourth column is a vector of vectors.

4.1 Function Lifting

Function lifting takes an AST with nested functions (functions appearing inside of other functions) and lifts these functions out of their local scopes and puts them all at the top level. Normally, function lifting must consider both the core operation of lifting function as well as the lexical scoping of the functions and variables that appear inside of a function. As seen below, our treatment uses a slightly different method. By using the path matrix, we are able to maintain the relevant information throughout the compiler and avoid resolving scope until around the same time that we handle other variable resolution tasks.

If we use the Key operator (`⌸`) with the pair function (`{α ω}`) using `Fk` as the keys and `Fast` as the AST, we group all of the relevant parts of the tree according to which nodes would appear in their respective functions after lifting. Refer to the original AST given above to verify this. The second column in the following result shows the body of each function, complete and ready to name. Each element in the second column corresponds to the body of one of our to be lifted functions.

```
      (1↓Fk){α ω}⌸1↓Fast
┌─────────┬────────────────────┐
│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│   │
│         │└─┴─┴─┴─────────┘   │
└─────────┴────────────────────┘
```

In the case of the first function, the outer function, notice the 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 grouped sub-trees will be given a new local root `F` node that is at the appropriate depth (0) and uses the key as its path vector (`⊣` or `α`).

```
      (1↓Fk){⊂(0 'F' 0 α)⍪⍵}⌸1↓Fast
┌───────────────────┬───────────────────┐
│┌─┬─┬─┬─────────┐  │┌─┬─┬─┬─────────┐  │
││0│F│0│1 0 0 0 0│  ││0│F│0│1 1 1 0 0│  │
│├─┼─┼─┼─────────┤  │├─┼─┼─┼─────────┤  │
││1│E│0│1 1 0 0 0│  ││3│E│0│1 1 1 1 0│  │
│├─┼─┼─┼─────────┤  │├─┼─┼─┼─────────┤  │
││2│F│0│1 1 1 0 0│  ││4│V│⍵│1 1 1 1 1│  │
│├─┼─┼─┼─────────┤  │├─┼─┼─┼─────────┤  │
││2│A│0│1 1 2 0 0│  ││4│P│+│1 1 1 1 2│  │
│├─┼─┼─┼─────────┤  │├─┼─┼─┼─────────┤  │
││3│N│7│1 1 2 2 0│  ││4│V│⍵│1 1 1 1 3│  │
│└─┴─┴─┴─────────┘  │└─┴─┴─┴─────────┘  │
└───────────────────┴───────────────────┘
```

At this point, the internal function nodes refer by path vector to the newly added local roots. This does not require traversing or maintaining these connections, but falls out as a natural artifact of using the Key (`⌸`) operator. At this point we are ready to fix up the sub-tree to be correct. Firstly, we need a function to encapsulate the function header information.

```
      hd←0 'F' 0,∘⊂⊣      ⍝ Func. Header
```

Next, we write a function to adjust the depths, which is one more than the difference between each node's depth and the depth of the first node of the sub-tree (`n⌷⍤1` gives the nth column).

```
dp←(1+⊢-⊃)0⌷⍤1⊢      ⍝ New Depths
```

Finally, we use the `@` operator to replace occurances of `F` in the Type column with `V` elements.

```
rf←'V'@('F'∘=)1⌷⍤1⊢ ⍝ F → V
```

Once we put all this together, we can construct the new sub-trees with the appropriate depths, types, references, and links with the following expression.

```
(1↓Fk)(⊂hd⍪dp,rf,2↓⍤1⊢)⌸1↓Fast
┌───────────────────┬───────────────────┐
│┌─┬─┬─┬─────────┐  │┌─┬─┬─┬─────────┐  │
││0│F│0│1 0 0 0 0│  ││0│F│0│1 1 1 0 0│  │
│├─┼─┼─┼─────────┤  │├─┼─┼─┼─────────┤  │
││1│E│0│1 1 0 0 0│  ││1│E│0│1 1 1 1 0│  │
│├─┼─┼─┼─────────┤  │├─┼─┼─┼─────────┤  │
││2│V│0│1 1 1 0 0│  ││2│V│⍵│1 1 1 1 1│  │
│├─┼─┼─┼─────────┤  │├─┼─┼─┼─────────┤  │
││2│A│0│1 1 2 0 0│  ││2│P│+│1 1 1 1 2│  │
│├─┼─┼─┼─────────┤  │├─┼─┼─┼─────────┤  │
││3│N│7│1 1 2 2 0│  ││2│V│⍵│1 1 1 1 3│  │
│└─┴─┴─┴─────────┘  │└─┴─┴─┴─────────┘  │
└───────────────────┴───────────────────┘
```

Each of these function bodies has a specific path vector associated with it. Because these paths 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.

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

Flattening expressions is almost identical to lifting functions, in part because we have delayed the lexical scope resolution aspect of lifting functions to a later pass that will handle this explicitly. In addition to lifting the expressions themselves, however, we will also take the opportunity to handle precedence and ordering. In our examples, we are assuming that all expressions evaluate right to left. Because the Key (`⌸`) operator is a stable operation, and we require that nodes have a depth-first pre-order traversal pattern, then we can trivially handle precedence order of this type, including handling parentheses, by reversing the results of flattening.

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

```
(1↓Ek){⍺ ⍵}⌸1↓East
┌─────┬─────────────────┐
│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 reference based on that node's path. Each expression has a unique name based on its path. A later compiler pass minimizes the names used in expressions (like register allocation). As with the function lifting case, we need some functions to help abstract each new field. In our case, we use the same functions as with function lifting but alter the `hd` and `rf` functions to use the right node types:

```
hd←0'E'0,∘⊂⊣
rf←'V'@('E'∘=)1⌷⍤1⊢
```

The resulting expression lifting operation is almost identical to that of function lifting. Notice that we reverse the 3-D matrix result at the end along the first axis to preserve order of evaluation correctly.

```
⊖(1↓Ek)(hd⍪dp,rf,2↓⍤1⊢)⌸1↓East
```

```
┌─┬─┬─┬─────┐
│0│E│0│1 3 0│
├─┼─┼─┼─────┤
│1│V│c│1 3 4│
├─┼─┼─┼─────┤
│1│P│×│1 3 5│
├─┼─┼─┼─────┤
│1│V│d│1 3 6│
└─┴─┴─┴─────┘

┌─┬─┬─┬─────┐
│0│E│0│1 1 0│
├─┼─┼─┼─────┤
│1│V│a│1 1 1│
├─┼─┼─┼─────┤
│1│P│+│1 1 2│
├─┼─┼─┼─────┤
│1│V│b│1 1 3│
└─┴─┴─┴─────┘

┌─┬─┬─┬─────┐
│0│E│0│1 0 0│
├─┼─┼─┼─────┤
│1│V│0│1 1 0│
├─┼─┼─┼─────┤
│1│P│÷│1 2 0│
├─┼─┼─┼─────┤
│1│V│0│1 3 0│
└─┴─┴─┴─────┘
```

The only extra issue involved here over function lifting 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 given by ⌸ 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 [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 [21] 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 Path Matrix. 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 Path vectors 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 path matrix. 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 Path Vectors.

Likewise, path vectors 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 Path

Pass	Description	Core Traversal Pattern	Uses paths?
Record Path Vectors	Adds a new field to each node containing that node's path vector.	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

Matrix 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 Path Vectors 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. Performance

To examine the performance characteristics of the above techniques, we prototyped a micro-benchmark that flattens expressions on an AST. This mimics the core functionality of both the function lifting and expression flattening passes described in more detail above and used in the Co-dfns compiler. It also serves to compare the performance of using Key (⌸) based algorithms to perform tree transformation compared to recursively defined algorithms. We implemented the above lifting algorithm using CUDA 8.0 and Thrust primitives.

We implemented the same algorithm using two traditional recursive methods. The first is a “Functional” analogue, that avoids mutation of the result and instead uses a pure-functional interface to pass nodes to be lifted back up the call stack. This is a very common method, taught in compiler courses that teach flattening passes, as well as used in various compilers. The benefit of the functional algorithm is the concision with which it may be written in most higher-level languages, particularly those which eschew mutation and favor a pure coding style. The disadvantage is that it has a higher memory usage requirement and may copy data as it moves up through the call stack to reassemble various containers for child nodes.

The second algorithm is a common optimization on the functional recursive style that makes use of an accumulator container that threads through the computation. The traversal functions will then mutate this accumulator as nodes are discovered that require lifting. The advantage over the functional method is that you can avoid excess copying overheads and in some languages that are more favorable to mutation, it may even be easier to write. A disadvantage is that you must be carefully aware of the order of traversal through the AST, since a change in the ordering of traversal can affect the order in which traversal functions write to the accumulator. This makes it less friendly to multi-threading. It can also be cumbersome or less desirable to write in this style in some

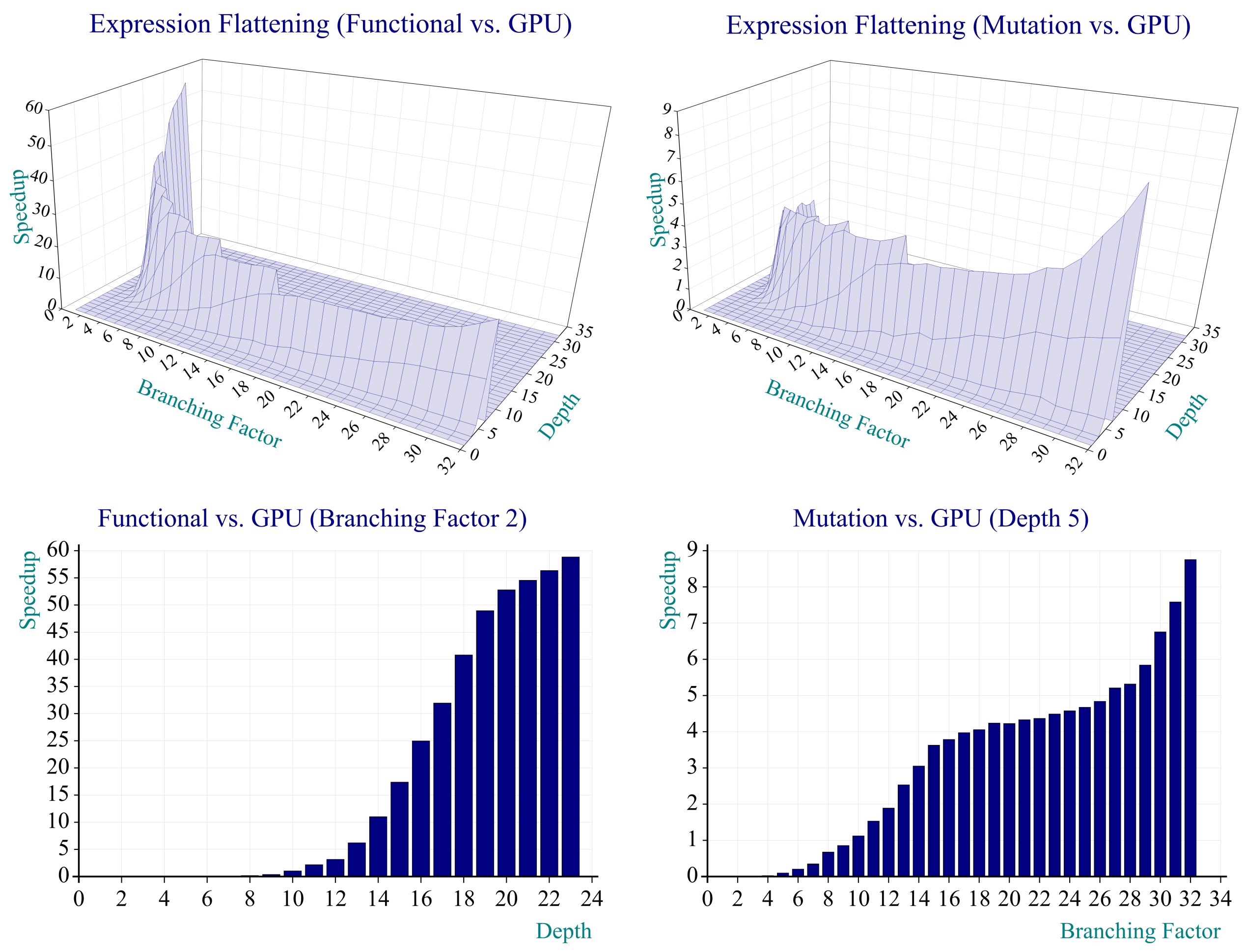


Figure 1. Performance of Key-based Expression Flattening compared to traditional methods

programming languages in which the functional algorithm finds natural expression.

We use a slightly more complex, but optimized path matrix than the one given above. In particular, we require that each non-zero element of a path vector be a valid row index into the path matrix, and that this index be a reference to the corresponding ancestor of the given depth for the node corresponding to the given path vector. By ensuring that each element is a valid index, we are able to optimize the question of whether a given parent has a given node type. Instead of the more expensive `∧.(=∨0=⊢)` based general solution, we can use the expression `⌈/C×t=T[C]` to acquire suitable keys to group our sub-trees for a type `t`, path matrix `C`, and type vector `T`. This optimized expression is not as general, but it covers a common case. The critical path for computing the nearest parent of a given type for the AST is less than depth of the AST, depending on how one implements the above expression.

We benchmarked the GPU algorithm against the traditional methods by depth of the AST and branching factor. We chose to examine depth as a specific factor because the path matrix is inherently dependent on the depth. Furthermore, the mutation algorithm is optimized in such a way that fewer expression nodes may result in fewer writes to the accumulator, while our algorithm has no such optimization, and traverses the entire tree independent of the number of expression nodes. We ran the benchmark to compute the time to flatten an AST on all combinations of depth and branching factor along the interval $[0, 32)$ that were of a suitable size (that is, fit within the machine constraints). The GPU was an NVIDIA Tesla K40c, and the CPU was an Intel Xeon E5-1620v3.

Figure 1 shows the results of this experiment. It is interesting to note the difference in shapes between the functional and mutation algorithms relative to the GPU algorithms when mapped along both the depth and branching factor. As expected, the speedups against the functional algorithm are higher on the whole because of the higher memory demands of that algorithm. All these algorithms should be memory bound, so a reduction in overall memory traffic benefits the mutation algorithm quite a bit.

Our peak speedups are around 58× comparing against the functional algorithm, and around 9× when comparing against the mutation algorithm. However, the “performance” zone for the GPU algorithm is fairly similar in shape between the two. Particularly, this ”curved mountain peak” represents the boundaries where the number of AST elements in the whole AST becomes large enough to attain sufficient parallelism on the GPU.

The second row of graphs in Figure 1 highlights a single plane of the mountain where we fix the branching factor or the depth. This gives a better picture of the behavior of the algorithm along the most pronounced areas of the area graph. These are also of particular interest because they represent two common cases found in real code. When the branching factor is 2, we have a reasonable analogue of common expressions found in code such as APL or other mathematically intense code which may have long strings of dense expressions, leading to a higher depth, but a low branching factor (since most operators of this sort take one or two arguments). When the Depth is fixed at 5, but we increase the branching factor, this more closely simulates code that may have functions nested throughout the code that need to be lifted which contain many child statements, but may not be particularly deep. In such cases the branching factor would increase, but the ASTs themselves may be relatively low depth.

It should also be noted that while the mutation-oriented algorithm is obviously faster than the functional algorithm on the CPU, it comes at the cost of stylistic clarity and the capability to reason locally about computation. Our Key-based algorithm remains pure and functional while still being fast. It is also exceptionally concise relative to the other two algorithms.

7. Discussion

There are many options for representing graphs on data-parallel architectures. Most of them tend to optimize traversal instead of transformation. Because of the transformation heavy nature of compiler-style algorithms, these other representations can introduce work that complicates both the algorithms themselves, as well as the work necessary to maintain those structures over the lifetime of the compilation phase.

The use of a path matrix provides benefits to locality, which allows for increased parallelism for transformation passes such as the flattening passes mentioned above. While this can be used to increase the performance of a single compiler pass, the above treatment is not intended primarily for highly optimizing a single pass in a compiler. Instead, the design of the above algorithms and techniques are designed to improve the data parallelism and ease of expression across the entire compiler, not just a single pass.

To illustrate the difference, consider a few other possible representations. One possibility is the use of a single parent vector for the entire AST. We could also work directly off of the depth vector or the traditional method of listing pointers to the children and/or parents. In the case of the expression flattening above, we could likely achieve equal or better performance from using a single parent vector for this one computation. However, we would increase costs elsewhere, particularly in maintenance. A path matrix provides both parent information, as well as information about the depth of the node, the siblings, position in the tree, and a unique identifier in a single structure that is cache/memory friendly. Since many operations on trees are memory bound, this memory optimization becomes important. In the Co-dfns compiler, we are able to generate the path matrix once, at the beginning of the compilation, and pass them around freely, and even duplicate them when appropriate, to communicate information without requiring cross thread or non-local memory access patterns.

This means that we never need to recompute the matrix, and indeed, doing so would defeat the purpose. In contrast, if you require a parent vector to organize your AST, then that parent vector may need to be shuffled, extended, and permuted each time the AST changes, and fails to maintain the original edge information of the graph. Keeping the original parent vector but maintaining a set of links would be a potential solution, but removes the locality of the access patterns, making it impossible to do a coalesced read of all of edge information needed for a single computation as a single access. Using a path matrix allows us to avoid that and increase the amount of stride-1 regular accesses and reduces memory indirections.

The cost of a path matrix is its size. Many other representations are size efficient, depending only on the number of nodes in the AST. A path matrix on the other hand, in the worst case, is dependent both on the depth of the AST and the number of nodes in the AST. Depending on the specific application, it is possible to choose different path matrix representations that optimize certain cases. For instance, one may not require that traversing the parent path be fast, and thus, we may reduce the cost of generating the path matrix by removing that requirement. Removing this requirement may also allow a path matrix to use a smaller sized integer representation for the matrix, further reducing the size at the cost of increased computation for certain operations. We have shown general path matrix techniques above that will work regardless of these particular choices in creating the path matrix.

8. 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.

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.

9. Related Work

Iverson introduced the idea of the Rank operator while at Sharp Associates [18] 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, 17] The J programming language [16] 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, 19, 20, 24]

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 [23] 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 [22] 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].

10. Conclusion

We have derived a method of performing computation over sub-trees selected on the basis of inter-node properties through the use of the Key (⌸) operator and path vectors, which enable local computation about 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. This new paradigm shows improved performance for transformation heavy algorithms which form the backbone of any non-trivial compiler on data-parallel or vector hardware such as GPUs.

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] A. W. Hsu. The key to a data parallel compiler. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, ARRAY 2016, pages 32–40, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4384-8. doi: `10.1145/2935323.2935331`. URL `http://doi.acm.org/10.1145/2935323.2935331`.

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

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

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

[19] 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.

[20] 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.

[21] 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.

[22] 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.

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

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