Runtime Compilation of Array-Oriented Python Programs

by

Alex Rubinsteyn

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

New York University

September 2014

Professor Dennis Shasha

Dedication

This thesis is dedicated to my parents and to the area code 60076.

Acknowledgements

When I came to New York in 2007, I brought with me a Subaru Outback (mostly full of books), a thinly acquired degree in Neuroscience, a rapidly shrinking bank account, and a nebulous plan to become a mathematician. When I wrote to a researcher at MIT, seeking a position in his lab, I had to admit that: “my GPA is horrible, my recommendations grudgingly extracted from laughable sources.” To my earnest surprise, he never replied. Undeterred and full of confidence in the victory of my enthusiasm over my historical inability to get anything done, I applied to Courant’s Masters program in Mathematics and was promptly rejected. In a panic, I applied to Columbia’s School of Continuing Education and was just as quickly turned away. I peppered them with embarrassing pleas to reconsider, until one annoyed administrator replied that “inconsistency and concern permeate each semester” of my transcript. Ouch.

That I still ended up having the privilege to pursue my curiosity feels like a miracle and I owe a large debt of gratitude to many people. I would like to thank:

- My former project-mate Eric Hielscher, with whom I carved out many of the ideas present in this thesis.

- My advisor, Dennis Shasha, who gave us guidance, support, discipline and chocolate almonds.

- Professor Alan Siegel, who helped me get started on this grad school adventure, taught me about algorithms, and got me a job which both paid the tuition for my Masters and trained me in “butt-time” (meaning, I needed to learn how sit for more than an hour).

- The job that Professor Siegel conjured for me was reading for Nektarios Paisios, who became my friend and collaborator. We worked together until he graduated, and I think both benefited greatly from the arrangement.

- Professor Amir Pnueli, who was a great teacher and whose course in compilers strongly influenced me.
- My floor secretary, Leslie, who bravely shields us all from absurdities so we can get work done. Without you, I probably would have dropped out by now.
- Ben, for being a great friend and making me leave my office to eat dinner at Quantum Leap.
- Geddes, for demolishing the walls we imagine between myth and reality. Stay stubborn, reality doesn't stand a chance.
- Most of all, I am grateful for a million things to my parents, Irene Zakon and Arkady Rubinsteyn.

Abstract

The Python programming language has become a popular platform for data analysis and scientific computing. To mitigate the poor performance of Python's standard interpreter, numerically intensive computations are typically offloaded to library functions written in high-performance compiled languages such as Fortran or C. When there is no efficient library implementation available for a particular algorithm, the programmer must accept suboptimal performance or switch to a low-level language to implement the routine.

This thesis seeks to give Python programmers a means to implement high-performance algorithms in a high-level form. We present Parakeet, a runtime compiler for an array-oriented subset of Python. Parakeet selectively augments the standard Python interpreter by compiling and executing functions explicitly marked for acceleration by the programmer. Parakeet uses runtime type specialization to eliminate the performance-defeating dynamicism of untyped Python code. Parakeet's pervasive use of data parallel operators as a means for implementing array operations enables high-level restructuring optimization and compilation to parallel hardware such as multi-core CPUs and graphics processors. We evaluate Parakeet on a collection of numerical benchmarks and demonstrate its dramatic capacity for accelerating array-oriented Python programs.

Contents

List of Figures

List of Tables

List of Code Listings

List of Algorithms

Chapter 1

Introduction

I'm not a programmer, I'm a statistician. I sit in a room with pencil and paper coming up with new models and working out the mathematics behind them. I only program because no one else is going to code my models for me.

— Comment on reddit.com

It is a hallmark of our data saturated age that many self identified non-programmers find themselves trying to tell a computer what to do. Scientists need to distill the numerical deluge spewing from their experiments, the quants of Wall Street want to predict the movement of chaotic markets, and seemingly everyone wants to analyze your data on the internet. Similarly, many professional programmers (who don't identify as scientists or statisticians) are wading into the numerical fray to filter, transform and summarize an increasingly quantified world.

Against this backdrop of non-programmers programming and non-numericists numericizing, the Python programming language [VRDJ95] has emerged as a popular environment for number crunching and data analysis. This may come as a surprise since, unlike Mathematica [Wol99], Matlab [MAT10] or R [IG96], Python was not originally conceived of as a "mathematical", "numerical" or "statistical" programming language. However, by virtue of Python's semantic flexibility and easy interoperability with lower-level languages, Python has grown an extremely rich ecosystem of scientific libraries. These include NumPy [Oli06] and Pandas [McK11], which provide expressive numerical data structures, SciPy [JOP01], a generous repository of mathematical functions, and matplotlib [Hun07], a flexible plotting package. The versatility of these tools, along with vibrancy of the communities which develop and use them, has led to Python becoming the *lingua franca* in several scientific disciplines [MGS07, PGH11, Gre07, Blo03].

But isn't Python slow? In fact, yes, the standard language implementation (a bytecode interpreter called *CPython*) can be orders of magnitude slower than statically compiled lower-level languages such as C and Fortran. If your goal is to quickly perform some repetitive arithmetic on a large collection of numbers, then ideally those numbers would be stored contiguously in memory, loaded into registers in small groups, and acted upon by a compact set of native machine instructions. The Python interpreter, on the other hand, represents values using bulky heap allocated objects and even simple operations, like multiplying two numbers, are implemented using extensive indirection.

If the Python interpreter's inefficiencies could not be somehow eliminated then would-be users of numerical Python tools would balk at sub-standard performance. In fact, a recent survey [PJR+11] of computational practices among scientists noted that:

> *Nothing evoked stronger reactions during the interviews than questions regarding the impact of faster computation...Faster computation enables accurate scientific modeling within time scales previously thought unattainable.*

The key to satisfying these performance-hungry scientists and getting good numerical performance from Python is to off-load computationally intensive tasks to lower-level library code. If an algorithm spends most of its time performing some common operation through an efficient library – matrix multiplication using ATLAS [WPD01], or Fourier transforms through FFTW [FJ05] – then there should be no significant performance difference between a Python implementation and an equivalent program using the same libraries from a more efficient language.

The NumPy array library [Oli06] plays an important role in enabling the use of efficient algorithmic implementations from within Python by providing a high level Pythonic interface around unboxed arrays. These arrays can be passed easily into precompiled C or Fortran code. The NumPy array is the *de facto* numerical container for Python programs, on top of which other data structures are typically constructed.

NumPy's mode of library-centric development provides acceptable performance as long as the algorithm you are implementing can be expressed primarily in terms of existing compiled primitives. However, a developer may need to implement a novel "inner loop", for which no precompiled equivalent exists. In this case, they are faced with two frustrating choices: either implement their desired computation in pure Python (and thus suffer a severe performance penalty) or write in some lower-level language, and face a dramatic loss of productivity [Pre03]. For example, the developers of the widely used scikit-learn [PVG$^+$11] machine learning library spent about half a year of development time reviewing and integrating a statically compiled implementation of decision tree ensembles. A pure Python version would have been dramatically shorter and simpler, but had little chance of attaining sufficiently fast performance.

To make things worse, merely moving a bottleneck into a lower-level language is unlikely to fully utilize a computer's available computational resources. A modern desktop computer will typically have somewhere between 2-8 cores and a graphics processor (GPU) capable of general-purpose computation. GPUs are highly parallel "many-core" architectures which, depending on the task, can be 10x-100x faster than multi-core CPU algorithms [OHL$^+$08, Kin12, KY13]. Several fields of research, such as deep neural networks [KSH12], have become entirely reliant on the computational power of GPUs to perform tasks which would be infeasibly slow on the CPU. Unfortunately taking advantage of all this parallel hardware is not easy. Splitting a computation across multiple cores necessitates either manually using a threading library [NBF96] or a parallel API such as OpenMP [DM98]. GPU acceleration necessitates even more programmer effort, requiring intimate knowledge of the graphics hardware and the use of a specialized language such as CUDA [Nvi11] or OpenCL [SGS10]. What may have began as an elegant prototype in Python runs the risk of devolving into a complex and error-prone exercise in parallel programming.

In between the unsatisfactory performance/productivity profiles of purely high level and purely low level implementations runs the middle road of runtime compilation. Common sources of inefficiency in a high level language's implementation – such as dynamic dispatch and tagged heap-allocated data representations – can be bypassed if native code gets generated at runtime, when sufficient information is present for specialized compilation.

It may even be possible to use high level algorithmic descriptions, extracted from a language such as Python, to generate code which is faster than an implementation in C or Fortran. There are several factors which could contribute to a significant speed-up over manually crafted low level code:

- Pre-compiled performance primitives must be separately compiled and thus cannot perform optimizations which arise when these routines are used together. Thus, the composition of multiple such functions may incur wasteful overhead. For example, a quantity may be computed repeatedly, once per function being called or large temporary values may be created where a more holistic view of the code could have avoided any extra allocation or data traversal. On the other hand, operations in a high level description of a numerical algorithm can be fused and rearranged to ultimately attain a faster implementation.

- Implementing a performance-sensitive portion of an algorithm in C or Fortran does not, by itself, do anything to harness the woefully underutilized parallel hardware found in most modern machines. If, for example, a programmer wants to move some computation to their GPU, they must learn an unfamiliar and tricky programming model such as OpenCL [SGS10] or CUDA [Nvi11]. High level algorithmic descriptions, on the other hand, are rife with flexible evaluation orders and opportunities for parallel execution. If there is any hope of making general-purpose GPU programming an easy and accessible activity, it will likely come from the automatic translation of high level programs to the GPU.

- Since the actual values of the computational inputs are available at runtime, it is possible to partially evaluate the programmer's code on portions of those values which may significantly impact performance. For example, many routines in the NumPy array library must either explicitly provide specializations for particular data layouts or fall back upon slower layout-agnostic implementations. Runtime value specialization, enabled in the context of a runtime compiler, allows for the possibility of generating conservative layout-agnostic code and then specializing it for the "array strides" of a particular input.

This thesis seeks to give Python programmers a means to implement high-performance parallel algorithms in a high level form. We present Parakeet [RHWS12], a runtime compiler for an array-oriented subset of Python. Parakeet aims to allow compact high level specification of numerically intensive algorithms which attain performance comparable to more verbose implementations in lower-level languages. Toward this performance goal we employ type specialization and the permissive semantics of data parallel operators which enable sophisticated optimizations and parallel execution across multiple cores and on the GPU.

The major contributions of this thesis are:

1. A demonstration that runtime-compiled implicitly parallel numerical programming can significantly outperform precompiled NumPy library functions and run in parallel across multiple cores and on the GPU.

2. A type inference algorithm which propagates input types throughout a function to derive specialized code.

3. A rich system of *high order data parallel operators*, such as the familiar **Map**, **Reduce**, **Scan** but also more esoteric or previously unseen operators such as **IndexReduce**, **FilterReduce**, and **OuterMap**. Parakeet unifies the closely related paradigms of array languages and data-parallel programming by desugaring high level array constructs into explicit uses of data parallel operations.

The combined effect of Parakeet's typed representation, high level optimizations and use of the GPU or multi-core parallelism is a significant and consistent acceleration of numerical Python programs. For those programs which consist primarily of calls to NumPy library functions, Parakeet may speed them up by an order of magnitude. For programs which perform most of their computations within the Python interpreter, Parakeet's speedup can reach into the tens of thousands.

Chapter 2

Overview of Parakeet

Parakeet is a parallelizing runtime compiler which coexists with Python interpreter, selectively taking over execution of user-specified functions. If you want Parakeet to compile a particular function, then wrap that function with the `@jit` function decorator, such as:

```
@jit
def avg(X,Y):
  return (X+Y) / 2.0
```

Listing 2.1: Averaging Two Arrays

If the decorator `@jit` were removed from Listing 2.2, then `avg` would run as ordinary Python code. Without Parakeet, the function `avg` would execute by having the Python interpreter would call the `__add__` and `__divide__` methods of array objects `X`. Since NumPy's implementation of the array operations + and / are compiled separately, they can't help but allocate result arrays. In this case, however, an intermediate array is clearly wasteful since the result of `x+y` is immediately consumed by the division.

On the other hand, Parakeet specializes `avg` for any distinct input type, optimizes its body into a single combined array operation (avoiding unnecessary allocation) and executes it as parallel native code.

The `@jit` decorator cannot be used on any arbitrary Python function since Parakeet is not a general-purpose compiler for all of Python. The job of the `@jit` decorator is to intercept calls into a wrapped function and then to initiate the following chain of events:

1. Translate the function into an untyped representation, from which we'll later derive multiple type specializations.

2. Specialize the untyped function for the current argument types, creating a typed version of the function.

3. Aggressively optimize the typed code, leveraging the high-level intermediate representation to perform optimizations which would be more difficult in C or Fortran.

4. Lower the optimized code into a parallel low-level implementation using either OpenMP (for multi-core execution) or CUDA (for the GPU).

Parakeet only supports a handful of Python's data types (numbers, tuples, slices, and NumPy arrays). To manipulate these values, Parakeet lets the programmer use any of the usual math and logic operators, along with a subset of built-in and NumPy library functions. If your performance bottleneck doesn't fit neatly into Parakeet's restrictive universe then you might benefit from a faster Python implementation such as PyPy [RP06], or alternatively you could outsource some of your functionality to native code via Cython [BBC$^+$11] or an extension module written in C or Fortran.

2.1 Typed Intermediate Representation

Parakeet uses a typed intermediate representation: unlike Python, every variable is annotated with a type annotation. Furthermore, operations such as adding are significantly more restricted (or perhaps, "static") than their equivalents in Python. For example, unlike Python, the addition operator + does *not* signify dynamic dispatch on the `__add__` method. Rather, in Parakeet's intermediate representation, each occurrence of + can only act on scalar values of a certain type. Arithmetic operators between array values must expressed more explicitly using *data parallel operators* such as **Map**. For example, if the function in Listing 2.2 were passed two float vectors, then Parakeet's typed intermediate representation of this would look like:

```
def avg(X :: array1(float32), Y :: array1(float32)):
  temp :: float32 =  Map(lambda xi, yi: xi + yi, X, Y)
  return Map(lambda xi: xi / 2.0, temp)
```

Listing 2.2: Averaging Two Arrays With NumPy

2.2 Data Parallel Operators

Parallelism in Parakeet is achieved through the implicit or explicit use of data parallel operators. Some examples of these operators are listed below:

- **Map**(f, $X_1, \ldots, X_n$, *axis*=None)

  Apply the function f to each element of the array arguments. By default f is passed each scalar element of the array arguments. The *axis* keyword can be used to specify a different iteration pattern (such as applying f to all columns).

- **Reduce**(*combine*, $X_1, \ldots, X_n$, *axis*=None, *init*=None)

Combine all the elements of the array arguments using the binary commutative function *combine*. The *init* keyword is an optional initial value for the reduction. Examples of reductions are the NumPy functions `sum` and `product`.

- **Scan**$(combine,\ X_1, ..., X_n,\ axis{=}\text{None},\ init{=}\text{None})$
  Combine all the elements of the array arguments and return an array containing all cumulative intermediate values of the combination. Examples of scans are the NumPy functions `cumsum` and `cumprod`.

For each occurrence of a data parallel operator in a program, Parakeet may choose to synthesize parallel code which implements that operator combined with its function argument. It is not always necessary, however, to explicitly use one of these operators in order to achieve parallelization. Parakeet implements NumPy's array broadcasting semantics by implicitly inserting calls to **Map** into a user's code. Furthermore, NumPy library functions are reimplemented in Parakeet using the above data parallel operators and thus expose opportunities for parallelism.

2.3 Compilation Process

We will refer to the following code example to help illustrate the process by which Parakeet transforms and executes code.

```
@jit
def norm(x):
  return np.sqrt(sum(x*x))
```

Listing 2.3: Vector Norm in Parakeet

When the Python interpreter reaches the definition of `norm`, it invokes the `@jit` decorator which parses the function's source and translates it into Parakeet's untyped internal representation. There is a fixed set of builtin functions (such as `sum`) and NumPy helpers (such as `np.add`) primitive functions from NumPy and library, such as `np.sqrt`, which are translated directly into Parakeet syntax nodes. In particular, `sum(x*x)` is rewritten into `Reduce(lambda acc,xi: acc+xi, Map(lambda xi, yi: xi*yi, x, x))`.

In general, the `@jit` decorator will raise an exception if it encounters a call to a non-primitive function which either can't be parsed or violates Parakeet's semantic restrictions. Lastly, before returning execution to Python, Parakeet converts its internal representation to a structured form [BM94] form of Static Single Assignment [CFR$^+$91].

2.3.1 Type Specialization

Parakeet intercepts calls to `norm` and uses the argument types to synthesize a typed version of the function. During specialization, all functions called by `norm` are themselves specialized for particular argument types. In our code example, if `norm` were called with a 1D `float` array then sum would also be specialized for the same input type, whereas the anonymous function created to add the elements of `x*x` would be specialized for pairs of scalar `floats`.

In Parakeet's typed representation, every function must have unambiguous input and output types. To eliminate polymorphism Parakeet inserts casts and **Map** operators where necessary. When `norm` is specialized for vector arguments, its use of the multiplication operator is rewritten into a 1D **Map** of a scalar multiply.

The actual process of type specialization is implemented by interleaving an abstract interpreter, which propagates input types to infer local types, and a rewrite engine which inserts coercions where necessary.

2.3.2 Optimization

In addition to standard compiler optimizations (such as constant folding, function inlining, and common sub-expression elimination), we employ fusion rules [AS79, JTH01, KM93] to combine array operators. Fusion enables us to increase the computational density of generated code and to avoid the creation of unnecessary array temporaries.

Using Parakeet can be as simple as calling a Parakeet library function from within existing Python code. For example, the first call to `parakeet.mean(matrix)` will compile a small program to efficiently average the rows of a matrix in parallel. Repeated calls will not incur further compilation costs. If a Python function is wrapped with the `@parakeet.jit` decorator, then its body will be parsed by Parakeet and prepared for later compilation. When such a function is finally called, its untyped syntax will be specialized for the types of the given arguments and then compiled and executed. For example, consider the simple function shown in Figure 2.4.

```
@parakeet.jit
def add1(x):
  return x+1
```

Listing 2.4: Simple Parakeet function

If `add1` is called with an integer argument, then it will be compiled to return an integer result. If, however, `add1` is later called with a floating point input then a new native implementation will be compiled that computes a floating point result.

This example does not make use of any data parallel operators. In fact, it is possible to generate code with Parakeet using only its capacity to efficiently compile loops and scalar operations. However, even greater performance gains can be achieved through either the explicit use of data parallel operators or, commonly, the use of constructs which implicitly generate data parallel constructs. For example, if you were to call `add1` with an array, then Parakeet would automatically generate a specialized version of the function whose body contains a **Map** over the elements of `x`. This can also be written explicitly, as shown in Figure 2.5.

```
@parakeet.jit
def add1_map(x):
  return parakeet.map(lambda xi: xi + 1, x)
```

Listing 2.5: Explicit map, adds 1 to every element

2.4 Backends

Parakeet allows you to select between different kinds of native code generation by setting the the configuration option `parakeet.config.backend` to one of the following values:

1. "*openmp*": This is the default backend, which generates translates Parakeet's optimized representation into C code and uses the OpenMP [DM98] to parallelize array operations across multiple cores.

2. "*cuda*": Uses NVIDIA's CUDA [Nvi11] framework to run parallel array operations on the GPU. This is not the default since (1) not all computers have CUDA-capable NVIDIA GPUs and (2) the compile times of NVIDIA's `nvcc` compiler are tediously long.

3. "*c*": Generate sequential C code, intended for machines with compilers which do not support the OpenMP API. This backend is also useful as a baseline for determining the degree of speedup achieved by parallelization of other backends.

4. "*interp*": Pure Python interpreter for Parakeet's intermediate representation, used primarily for internal debugging purposes, in general runs significantly slower than Python.

Rather than modifying a global configuration option, it is also possible to pass the name of the desired backend when calling into a Parakeet function with the keyword argument `_backend`.

2.5 Limitations

Parakeet is a runtime compiler for numerically intensive algorithms written in Python. In comparison with arbitrary Python code, Parakeet programs are very constrained. This is because Parakeet trades dynamicism and complexity for greater latitude in program transformation and ultimately the ability to generate efficient low-level code (we still have high-level constructs, they're just well behaved). The features which Parakeet does support are chosen to facilitate array-oriented numerical algorithms such as those found in machine learning, financial computing, and scientific simulation. The sections of code that Parakeet accelerates must obey the following constraints:

- The only data types which can be used within code compiled by Parakeet are tuples, scalars, slices, NumPy arrays, and the `None` object. Other data types, such as sets, dictionaries, generators, and user-defined objects are not supported.

- Due to Parakeet's use of a simplified program representation, only "structured" [BM94] control flow is allowed. This precludes the use of exceptions, as well as the `break` and `continue` statements.

- Since Parakeet must have access to a function's source, any compiled C extensions (or functions which call C extensions) cannot be used within Parakeet.

- Parakeet treats aggregate structures such as slices and arrays as immutable and the programmer cannot modify the values of their fields. The *data* contained within an array, however, can be modified. Similarly, the value of a local variable can be updated. So, though Parakeet does restrict where mutability can happen, Parakeet does *not* require the programmer to adhere to a purely functional programming style (unlike Copperhead [CGK11]).

- To compile Python into native code we must assign types to each expression. We are still able to retain some of Python's polymorphism by specializing different typed versions of a function for each distinct set of argument types. However, expressions whose types depend on dynamic values are disallowed (e.g. `42 if bool_val else (1,2,3)`).

- A Parakeet function cannot call any other function which violates these restrictions or one which is not implemented in Python. These restrictions recursively apply down the call chain, so that the use of an invalid construct by a function also taints any other functions which call it.

- To enable the use of NumPy library functions Parakeet must provide equivalent functions written in Python. However, some NumPy functions are not yet implemented and others use language constructs which make their inclusion in Parakeet unlikely to ever occur. For a full listing of which NumPy functions are supported, the programmer can look in the `parakeet.mapping` module.

These restrictions would be onerous if applied to an entire program but Parakeet is only intended to accelerate the computational core of an algorithm. All other code is executed as usual by the Python interpreter.

2.6 Differences from Python

In addition to restrictions on what sort of source code will even be accepted by Parakeet's frontend, there are also some semantic differences which may result in Parakeet computing a different value from Python.

- Since Parakeet does not implement Python lists, any list literals are treated as arrays. Similarly, list comprehensions are reinterpreted as (parallel) array comprehensions.

- Some expressions which seem to have conditionally dependent types *will* successfully be translated into Parakeet, but only by casting both values of a conditional into a single unifying type. For example, the Python expression `3 if bool_val else 2.0` will selectively return either an integer or a floating point value but Parakeet will upcast the `3` into a `3.0`.

- Parakeet's reimplementation of NumPy library functions does not currently support non-essential parameters such as `output`.

- Parakeet's parallel backends (OpenMP and CUDA) may change the order of a floating-point reductions (such as summing the elements of an array), resulting in small differences relative to Python due to the slight non-commutativity of floating point math.

- Parakeet implements NumPy's *array broadcasting* using a translation scheme dependent on the types of arguments. For example, adding a matrix and a vector together will be correctly translated into a **Map** which adds the vector to each column of the matrix. The full semantics of broadcasting, however, also allows for array elements to be replicated due to the values of an array's shape, which are not evident in the type signature. For example, adding a *200x1* matrix with a *1x300* matrix in NumPy will yield a *200x300* result. Parakeet, however, does not implement this correctly and will give an erroneous result.

2.7 Detailed Compilation Pipeline

In this section, we follow a simple function on its journey through the Parakeet JIT compiler to elucidate how Parakeet translates high level code into high performance, native versions. The function we will compile is the `count` function shown in Figure 2.6, which sums up the number of elements in an array less than a given threshold. We use a loop in this example rather than an adverb for now, as our focus is on the Parakeet compilation pipeline.

```python
@parakeet.jit
def count(values, thresh):
  n = 0
  for elt in values:
    n += elt < thresh
  return n
```

Listing 2.6: count: Python source

This function is simple, but it's not an entirely contrived computation. For example, it forms the core of a decision tree learning algorithm. Before breaking down how Parakeet compiles `count`, let's first look at how the original gets executed within the standard CPython interpreter.

The first thing that happens to a function on its way to being executed is parsing. The source of `count` gets read as a string of characters, tokenized, and then turned into a structured syntax tree as shown in Figure 2.7.

```
FunctionDef(
  name='count',
  args=arguments(args=[Name(id='values'), Name(id='thresh')],
                 vararg=None, kwarg=None, defaults=[]),
  body=[
    Assign(targets=[Name(id='n')], value=Num(n=0)),
    For(target=Name(id='elt'), iter=Name(id='values'), body=[
      AugAssign(target=Name(id='n'), op=Add(),
        value=Compare(left=Name(id='elt'), ops=[Lt()],
                      comparators=[Name(id='thresh')]))]),
    Return(value=Name(id='n', ctx=Load()))])
```

Listing 2.7: count: Python AST

A naive interpreter would then execute the syntax tree directly. Python achieves a minor performance boost by instead compiling to a more compact bytecode, as shown in Figure 2.8.

The inefficiency of tree-walking interpreters (which evaluate syntax trees) compared with bytecode execution is one of the reasons that Ruby has generally been slower than Python. Though an improvement over a naive interpreter, trying to execute this bytecode directly still results in terrible performance. If you inspect the behavior of the above instructions, you'll discover that they involve repetitive un-boxing and re-boxing of numeric values in and out of their PyObject wrappers, wasteful stack manipulation, and a lot of other very wasteful computations. If we're going to significantly speed up the numerical performance of Python code, it's going to have run somewhere other than the CPython bytecode interpreter.

```
0    LOAD_CONST               1 (0)
3    STORE_FAST               2 (n)

6    SETUP_LOOP              30 (to 39)
9    LOAD_FAST                0 (values)
12   GET_ITER
13   FOR_ITER                22 (to 38)
16   STORE_FAST               3 (elt)

19   LOAD_FAST                2 (n)
22   LOAD_FAST                3 (elt)
25   LOAD_FAST                1 (thresh)
28   COMPARE_OP               0 (<)
31   INPLACE_ADD
32   STORE_FAST               2 (n)
35   JUMP_ABSOLUTE           13
38   POP_BLOCK

39   LOAD_FAST                2 (n)
42   RETURN_VALUE
```

Listing 2.8: count: Python bytecode

2.7.1 From Python into Parakeet

When trying to extract an executable representation of a Python function, we face a choice between using a Python syntax tree or the lower-level bytecode. There are legitimate reasons to favor the bytecode – the syntax tree isn't saved anywhere and must instead be regenerated from source. However, the bytecode is littered with distracting stack manipulation and doesn't preserve some of the higher-level language constructs. Though it's a better starting point than the bytecode, an ordinary syntax tree is still somewhat clunky for program analysis and transformation. So, Parakeet starts with a Python AST and quickly slips into something a little more domain specific.

2.7.2 Untyped Representation

In Figure 2.9, we show the `count` function's internal representation in Parakeet. Notice that the loop counter n has been split apart into three distinct names: `n`, `n2`, and `n_loop`. This is because we translate the program into Static Single Assignment form. SSA is often used in compiler IRs because it allows for simplified analyses and optimizations. The most important things to know about SSA are:

- Every distinct assignment to a variable in the original programs becomes the creation of distinct variable. This is similar in style to functional programming.
- At a point in the program where control flow could have come from multiple places (such as the top of a loop), we explicitly denote the possible sources of a variable's value using a ϕ-node.

```
def count(values, thresh):
    n = 0
    for i in range(0, len(values), 1):
      (header)
        n_loop <- phi(n, n2)
      (body)
        elt = values[i]
        n2 = n_loop + (elt < thresh)
    return n_loop
```

Listing 2.9: count Untyped Intermediate Representation

Another difference from Python is that Parakeet's representation treats many array operations as first-class constructs. For example, in ordinary Python `len` is a library function, whereas in Parakeet it's actually part of the language syntax and thus can be analyzed with higher-level knowledge of its behavior. This is particular useful for inferring the shapes of intermediate array values.

2.7.3 Type-specialized Representation

When you call an untyped Parakeet function, it gets cloned for each distinct set of input types. The types of the other (non-input) variables are then inferred and the body of the function is rewritten to insert casts wherever necessary.

```
def count(values :: array1(float64), thresh :: float64) =>
   int64:
    n :: int64 = 0 :: int64
    shape_tuple :: tuple(int64) = values.shape
    for i in range(0, shape_tuple[0], 1):
      (header)
        n_loop <- phi(0 :: int64, n2)
      (body)
        elt :: float64 = values[i]
        less_tmp :: bool = elt < thresh
        n2 :: int64 = n_loop + cast<int64>(less_tmp)
    return n_loop
```

Listing 2.10: count: Typed Parakeet IR

A type-specialized version of `count` is given in Figure 2.10. Observe that the function has been specialized for input types `array1(float64)`, `float64` and that its return type is known to be `int64`. Furthermore, the boolean intermediate value produced by checking whether an element is less than the threshold is cast to `int64` before getting added to `n2`.

If you use a variable in a way that defeats type inference (for example, by treating it sometimes as an array and other times as a scalar), then Parakeet treats this as an error.

2.7.4 Optimization

Type specialization already gives us a big performance boost by enabling the use of an unboxed representation for numbers. Adding two floats stored in registers is orders of magnitude faster than calling Python's `__add__` operation on two `PyFloatObjects`.

However, if all Parakeet did was specialize your code it would still be significantly slower than programming in a lower-level language. Parakeet includes many standard compiler optimizations, such as constant propagation, common sub-expression elimination, and loop invariant code motion. Furthermore, to mitigate the abstraction cost of array expressions such as `0.5*array1 + 0.5*array2` Parakeet fuses array operators, which then exposes further opportunities for optimization. In this case, however, the computation is simple enough that only a few optimizations can meaningfully change it, as shown in Figure 2.11.

```
def count(values :: array1(float64), thresh :: float64) =>
   int64:
  shape_tuple :: struct(int64) = values.shape
  data :: ptr(float64) = values.data
  base_offset :: int64 = values.offset
  for i in range(0, shape_tuple.elt0, 1):
    (header)
      n_loop <- phi(0 :: int64, n2)
    (body)
      offset :: int64 = offset + i
      elt :: float64 = data[offset]
      less_tmp :: bool = elt < thresh
      n2 :: int64 = n_loop + cast<int64>(less_tmp)
  return n_loop
```

Listing 2.11: count: Optimized Parakeet IR

In addition to rewriting code for performance gain, Parakeet also "lowers" higher-level constructs such as tuples and arrays into more primitive concepts. Notice that the code in Figure 2.11 does not directly index into n-dimensional arrays, but rather explicitly computes offsets and indexes directly into an array's data pointer. Lowering complex language constructs simplifies the next stage of program transformation: translating from Parakeet into C.

2.7.5 Generated C code

The translation from Parakeet's lowered intermediate representation into C is largely mechanical. Structured data types such as array descriptors and tuples become C structs, which due to immutability assumptions can be safely passed by value. So that this code can be used by the Python interpreter, the entry point into a compiled C function must extract all the input values from the Python interpreter's `PyObject` representation. Similarly, the result must be packaged up as a value the Python interpreter understands. In between, however, the Python C API is not used. The C code for the `count` function is shown in Listing 2.12. In this instance, it does not matter whether the C, OpenMP, or CUDA backend was used, due to the lack of parallel operators in the original program, all the backends generate largely identical output.

2.7.6 Generated x86 Assembly

Once we pass the torch to the C compiler, Parakeet's job is mostly done. The external compiler performs its own host of optimization passes on our code and, at last, we arrive at native code. Shown in Figure 2.13 is the assembly generated for the main loop from the code above.

Notice that we end up with the same number of machine instructions as we originally had Python bytecodes. It's safe to suspect that the performance might have somewhat improved.

2.7.7 Execution Times

In addition to benchmarking against the Python interpreter (an unfair comparison with a predictable outcome), let's also see Parakeet stacks up against an equivalent function

```
#include <Python.h>
#include <numpy/arrayobject.h>
#include <numpy/arrayscalars.h>
#include <stdint.h>
#include <math.h>

typedef struct float64_ptr_type {
  double* raw_ptr;
  PyObject* base;
} float64_ptr_type;

typedef struct array_type {
  float64_ptr_type data;
  npy_intp shape[1];
  npy_intp strides[1];
  int64_t offset;
  int64_t size;
} array_type;

PyObject* count (PyObject* dummy, PyObject* args) {
  PyObject* values = PyTuple_GetItem(args, 0);
  npy_intp* shape_ptr = PyArray_DIMS((PyArrayObject*)values);
  npy_intp* strides_bytes = PyArray_STRIDES( (PyArrayObject*)
      values);
  array_type unboxed_array;
  float64_ptr_type data = {(double*) PyArray_DATA(((
      PyArrayObject*) values)), values};
  unboxed_array.data = data;
  unboxed_array.strides[0] = strides_bytes[0] / 8;
  unboxed_array.shape[0] = shape_ptr[0];
  unboxed_array.offset = 0;
  unboxed_array.size = PyArray_Size(values);
  PyObject* thresh = PyTuple_GetItem(args, 1);
  double thresh_2;
  if (PyFloat_Check(thresh)) { thresh_2 = PyFloat_AsDouble(
      thresh); }
  else { PyArray_ScalarAsCtype(thresh, &thresh_2); }
  npy_intp* shape = unboxed_array.shape;
  int64_t len_result = shape[0];
  int64_t n_loop = 0;
  int64_t idx;
  for (idx = 0; idx < len_result; idx += 1) {
    double elt = unboxed_array.data.raw_ptr[idx];
    int64_t temp = ((int64_t) (elt < thresh_2));
    int64_t n_loop = n_loop + temp;
  }
  return (PyObject*) PyArray_Scalar(&n_loop,
      PyArray_DescrFromType(NPY_INT64), NULL);
}
```

Listing 2.12: count: Generated C code

```
;; loop entry
  movq  8(%rdi), %rax
  movq  (%rax), %r8
  xorl  %eax, %eax
  testq %r8, %r8
  jle .LBB0_3
;; %loop_body.preheader
  movq  24(%rdi), %rax
  movq  (%rdi), %rdx
  leaq  (%rdx,%rax,8), %rdx
  xorl  %eax, %eax
  .align  16, 0x90
;; %loop_body
.LBB0_2:
  ucomisd (%rdx), %xmm0
  seta  %cl
  movzbl  %cl, %esi
  addq  %rsi, %rax
  addq  $8, %rdx
  decq  %r8
  jne .LBB0_2
.LBB0_3:
  ret
```

Listing 2.13: count: generated x86 assembly

implemented using NumPy primitives, shown in Figure 2.14.

In order to allow Parakeet to compile programs that look more like the NumPy version of `count`, we add adverbs to the Parakeet language. In fact, adverb use is encouraged not only because it allows the programmer to write code in a high-level array-oriented style. In addition, Parakeet is able to optimize and parallelize adverbs in ways that it cannot do on loops.

```
def numpy_count(values, thresh):
  return np.sum(values < thresh)
```

Listing 2.14: count: NumPy

On 10 million randomly generated inputs, best average the time taken for 5 runs each of the Python, NumPy, and two Parakeet versions is given in Table 2.1.

CPython loops	NumPy	Parakeet loops	Parakeet high level
33.6942s	0.0325	0.0127s	0.0063s

Table 2.1: Execution Time of different versions of count

Compared with CPython, both of the Parakeet implementations are several thousand times faster. Perhaps surprisingly, even on such simple code Parakeet is about 3x-5x faster than the pre-compiled NumPy library functions. The primary reason for this is that NumPy can't help create an array temporary for the expression `x < thresh`, whereas Parakeet optimizes this temporary away. Also, the NumPy version only uses one core, whereas the higher-level Parakeet implementation compiles to OpenMP and uses multiple cores simultaneously. In fact, if this problem were less memory-bound and exhibited greater computational density then Parakeet's high-level code would be twice faster than NumPy.

Chapter 3

History and Related Work

In the beginning, all computing was scientific computing. Early general purpose computers such as the ENIAC [GG46] and MESM [FBB06] were commissioned to compute artillery trajectories, simulate nuclear detonations, and crack cryptographic codes. These machines required a tremendous expenditure of human effort to correctly construct and encode their programs.

In the decade that followed, computers became widely available and better abstractions for programming them were eagerly sought after. As a variety of programming languages were devised, a tension became apparent between attaining efficiency through low-level control and the expressiveness of high-level abstractions. At the broadest scope, it is possible to distinguish two lineages of philosophy in the design of programming languages: (1) those whose features are carefully chosen to allow for efficient compilation and (2) abstract models of computation which optimize for programmer efficiency and leave efficient implementation as a secondary goal.

Programmers who adopted languages such as Fortran [BBB$^+$57], CPL [BBH$^+$63], and Algol [NBB$^+$63], demanded low-level control over data layout and program execution, hesitantly ceding a modicum of control over instruction selection to their compilers. Most relevant to this thesis is Fortran (“Formula Translator”), which remains in use today and has influenced nearly all the languages for scientific computing which followed. The first version of Fortran provided abstractions such as loops and array indexing, which could be compiled to a variety of hardware platforms. Great care was given, however, to ensure that all the language’s abstractions could be efficiently compiled to match the performance characteristics of hand-written assembly. Despite its helpful abstractions, Fortran was constrained to be *efficient for the machine*.

By contrast, programmers using Lisp [McC61], APL [Ive62], and the various languages that they inspired, placed their faith in an abstract model of computation far removed from the actual hardware upon which it ran. APL gave the programmer uniformly typed n-dimensional arrays and collective operators over them. Lisp, on the hand, focused the programmer on using non-uniform linked lists and functions as first-class language values. These abstractions afforded significant gains in brevity, clarity of thought, and consequently in productivity. The cost of abstraction, however, was a significant slowdown incurred from the pervasive use of data structures and operations unfamiliar to the hardware.

Most of the raw ingredients needed for a productive and efficient scientific programming language were in the air in the first few decades of modern computing, but only in piecemeal and scattered between different languages. Much of the later history of languages leading to Python, NumPy, and Parakeet is the story of combining abstractions and seeking more efficient implementations for them.

3.1 Array Programming

I adopted the matrix algebra used in my thesis work, the systematic use of matrices and higher-dimensional arrays (almost) learned in a course in Tensor Analysis rashly taken in my third year at Queen's, and (eventually) the notion of Operators in the sense introduced by Heaviside in his treatment of Maxwell's equations.

— Kenneth E. Iverson

Array Programming is a paradigm wherein the primary data structure is an n-dimensional array and algorithms are implemented chiefly through the creation, transformation, and contraction of array values. These semantics were inspired by matrix and tensor notation, but extended to collections of numbers without any particular linear algebraic properties. The array programming paradigm was first introduced by Kenneth Iverson's APL [Ive62], which he conceived of as a "mathematical notation" as much as a programming language.

APL uses an extremely terse syntax which is rich with different array operators. For example, summing the numbers from 0 to 99 can be expressed in APL as $+/\iota 100$. Here the first-order range operator "ι" is used to create a range of values and the higher-order reduction operator $/$ is combined with the first order addition function $+$ to express summation. Programming in APL makes extensive use of higher-order array operators, or in the language's native nomenclature, "adverbs".

The basic programming paradigm embodied by APL inspired a lineage of many other "pure" array languages, such as Nial [JGM86], Q [Bor08], and J [HI98]. The primary implementations for array languages have all tended to be simple interpreters, relying on efficient precompiled primitives for performance. The preference for interpreter implementations arises from the difficulty of assigning precise types to APL's sophisticated notion of subtyping between arrays and scalars [Tha90]. Nonetheless, driven by the desire for better performance, a creative variety of APL compilers were created [Bud83, Bud84, Chi86, Ber97].

The earliest known effort to compile APL [Abr70] was an extremely ambitious design for an APL virtual machine that, rather than evaluating expressions immediately, instead construct lazy expression trees and compiles them on demand. This turns out [Ayc03] to have been one of the first attempts at constructing a just-in-time compiler.

The actual community of programmers using APL (and its descendants) never grew very large, and the influence of array programming is felt primarily through its influence on other languages. APL directly inspired Speakeasy [CP79], which was "an extendable computer language based, like APL, upon the concept of arrays and matrices as natural entities but without the terseness of APL notation". Speakeasy, in turn, inspired the creation of Matlab [Mol80], which was meant to provide a high-level interface to performance critical Fortran libraries. Fortran itself eventually borrowed from the APL heritage, when Fortran90 [Met91] added "array expressions" to the core language, including element-wise intrinsic functions "often based on APL operators". NumPy was created to bring the functionality of Matlab to Python, and thus, through many layers, can trace its origins back to APL.

Collective operations over n-dimensional data structures have now become a ubiquitous feature of high level numerical programming and the origin of this feature is APL has been largely obscured. APL's innovative use of higher-order array operators is not as common as its first-order operators, since many languages with array-oriented features are interpreted and cannot efficiently implement higher-order usage of user-defined functions.

Aside from the adoption of n-dimensional arrays into dynamic languages, there were also several attempts to close the performance gap often perceived between dynamic array languages and Fortran by crafting statically compiled array languages. These include SISAL [CF90], ZPL [LS94], and Single Assignment C [Sch03].

3.2 Data Parallel Programming

A closely related paradigm is data parallel programming, which allows programmers to express algorithms through the declarative creation and transformation of uniform collections. For example, whereas an imperative language would require an explicit loop to sum the elements of an array, a data parallel language would instead implement summation via some form of high level reduction operator.

APL can be viewed as the first data parallel language, since it emphasized the collective transformation of n-dimensional arrays. Though the first implementation of APL was a sequential interpreter, the eminent parallelizability of the language's core operators was quickly recognized and parallel implementations were eventually developed. As computers with massively parallel hardware became more common in the 1980s many languages such as C [KRE88], Fortran [BBB+57], and Lisp [McC61], were retrofitted with data parallel extensions (Paralation Lisp [Sab88], HPF [BCF+93]). More recently, data parallel constructs have appeared repeatedly as core primitives for high level DSLs which synthesize distributed data processing algorithms (DryadLinq [YIF+08]), graphics card programs (Accelerator [TPO06], Copperhead [CGK11])).

The enduring appeal of data parallel constructs lies in the flexibility of their semantics. A data parallel transformation only specifies what the output *should be*, not how it is computed. This makes data parallel programs amenable (as the name suggests) to parallelization, both in terms of coarse-grained data partitioning and fine-grained SIMD vectorization. In addition, the algebraic nature of data parallel operators allows for dramatic restructuring of the program in order to improve performance. Most commonly, data parallel operators can be fused by applying simple syntactic rewrite rules which results in the elimination of synchronization points, increased computational density, and significant gains in performance.

3.2.1 Collection-Oriented Languages

Another closely paradigm closely related to both *array programming* and *data parallel programming* is that of *collection-oriented* languages. By the mid-1980's, there was a widespread proliferation of high level languages which relied on some privileged collection type as a central mechanism by which programs could be expressed. These included array-oriented languages of the APL family, the set-oriented language SETL [SDSD86], as well as Lisp dialects which used either conventional lists or other more exotic data structures such as distributed mappings [SJH86]. What such languages have in common is at minimum, an apply-to-each operation which implicitly traverses a collection (as well as often some form of reductions and scans).

Their similarities were documented by Sipelstein and Blelloch in a review named "Collection Oriented Langauges" [SB91]. Blelloch focuses on the pervasive use of data parallel operators such as **Map** and **Reduce**, though sometimes named differently or entirely disguised behind the semantics of implicit elementwise function application. This work seems to have inspired Blelloch's nested data parallel language NESL [BCH$^+$94].

The data parallel programming model allows programmers to express algorithms through the declarative creation and transformation of uniform collections. For example, whereas an imperative language would require an explicit loop to sum the elements of an array, a data parallel language would instead implement summation via some form of high level reduction operator.

The notion of collection-oriented languages overlaps greatly with data parallelism but they are not identical. For example, a reduction (even when expressed in a high-level collection-oriented form), may not be parallelizable due to the inherent sequentiality of its operator.

3.3 Related Projects

Numexpr [CH] is the simplest domain-specific numerical compiler for Python. It takes expressions containing simple element-wise operations, such as `tanh(x**2 - 2 * y)<= 0`, and compiles them to an efficient numerical virtual machine. This virtual machine executes the expression using cache-friendly blocking and splits the work across multiple cores.

Theano [BBB+10] is a library which allows programmers to construct expression graphs for multi-dimensional array operations and then compiles those operations to CPU or GPU code. Theano is remarkable in its ability to automatically compute gradients of user-defined functions, and has thus become very popular for the implementation of neural networks. Theano is more general than Numexpr, but still supports a much smaller range of expressions than Parakeet and requires encoding of programs in its explicit syntax representation.

Copperhead [CGK11] is a Python accelerator which compiles a purely functional data parallel subset of the language to the GPU. Copperhead repurposes Python's list comprehensions as a data parallel mappings and synthesizes GPU kernel templates for operators such as *map* and *reduce*. Copperhead is limited by its inability to express local mutable variables and array expressions. Parakeet can be seen as an extension of Copperhead's ideas to a much larger swath of idiomatic programming constructs.

Pythran [GBA13] is a compiler for numerical Python which achieves impressive performance by generating multi-core programs using OpenMP. It differs from Parakeet in that it is a static compiler which requires user annotations.

Numba [Con] is the closest project to Parakeet in both its aims and internal machinery. Like Parakeet, Numba provides a function decorator which intercepts function calls and uses argument types as the basis for type specialization. Numba then compiles type specialized programs to LLVM. Numba relies on loops for performance sensitive code, and lacks any concept analogous to Parakeet's higher order data parallel operators.

Chapter 4

Parakeet's Intermediate Representation

Parakeet uses a tree-structured intermediate representation where every function's body is a sequence of statements. Statements corresponding to loops and branches may in turn contain their own nested sequences of statements. All control flow in Parakeet must be "structured", meaning it is impossible to express arbitrary jumps between disparate parts of a program. Each structured control statement tracks the data flow arising from branching using SSA [CFR$^+$89] Φ-nodes at merge points. This combination is reminiscent of Oberon's Structured SSA [BM94], which is easier to analyze and rewrite but makes it difficult to express constructs such as Python's `break` and `continue`.

When a Python function is first encountered by Parakeet, it is translated into Parakeet's intermediate representation but without any type annotations. This syntax representation acts as an untyped function template, giving rise to multiple distinct typed instantiations for each distinct set of input types. Later, when specializing this untyped template for particular input types, all syntax nodes are annotated with types and any source of type ambiguity or potential dynamicism is translated into explicit coercions.

The two major transformations which must be performed while converting from Python's syntrax into Parakeet's intermediate representation are:

- **SSA Conversion**. Each assignment to the same variable name is given a distinct name. At places where a variable might take its value from multiple renamed source (control flow merge arising from loops and branches) explicit Φ-nodes track the flow of values. In Parakeet, these nodes are represented as a collection of bindings such as $x_i = \phi(e_{\text{left}}, e_{\text{right}})$, where the value of x_i is selected depending on which branch was taken.

- **Lambda Lifting** [Joh85] & **Closure Conversion** [Rey72]. Rather than allow nested function definitions as statements, Parakeet gives each function a unique identity by *lifting* it to the top-level scope. Any values which a definition closed over are added as formal parameters and where the definition originally took place a closure object is constructed instead.

4.1 Simple Expressions

This section enumerates the Parakeet expressions which are unrelated to the construction, transformation, and inspection of array values. This includes creating scalar values, tuples, closures, and slices.

- **Const**$(value)$

  Constants can be booleans, signed or unsigned integers, floating point values, or the `None` object.

- **Var**$(name)$

  Variables in Parakeet's intermediate representation must originate from a local binding either as the input argument to a function, a single assignment statement, or a ϕ-node associated with some control flow statement.

- **PrimCall**$(prim, x_1, \ldots x_n)$

  Parakeet's primitives are basic math operators such as `add` or `divide`, logical operators such as `logical_and`, or transcendental math functions such as `exp` or `sin`.

- **Select**$(cond, trueValue, falseValue)$

  Returns *trueValue* when *cond* is true, and *falseValue* otherwise.

- **Tuple**$(x_1, \ldots, x_n)$

  Construct an n-element tuple object from the individual values $x_1, \ldots, x_n$.

- **TupleElt**(x, i)

  Extract the i^{th} element of *x*. The index i must be a fixed constant and not a dynamically varying expression. Can be denoted more compactly as x_i.

- **Closure**$(f, x_1, \ldots, x_n)$

  Partially apply the arguments $x_1, \ldots, x_n$ to the function f, which must have at least $n + 1$ inputs. The result is a closure object, which can be called like a function.

- **ClosureElt**$(clos, i)$

  Extract the i^{th} partially applied argument from the closure value *clos*. The index i must be a fixed constant and not a dynamically varying expression. Can be denoted more compactly as $clos_i$.

- **Call**$(f, x_1, \ldots, x_n)$

  Calls a function or closure value with the given input $x_1, \ldots, x_n$. Evaluation semantics in Parakeet are *call-by-value* and functions must always return a result.

- **Slice**$(start, stop, step)$

  Construct a slice object with the given *start*, *stop*, and *step*, used to implement the Python syntax `start:stop:step`.

4.2 Statements and Control Flow

Statements in Parakeet bind variable names to values, initiate effectful computations, and allow for branching and loops. Unlike many intermediate representations, Parakeet's intermediate language is *not* a control flow graph [Pro59, All70], and does not allow unstructured jumps (which are necessary for Python statements such as `break` and `continue`).

- **Assign**$(lhs, value)$

  Evaluate the right hand side $value$ and assign it to the binding pattern lhs, which can be the name of a variable, an array indexing expression, or a tuple of other binding patterns.

- **ExprStmt**$(expr)$

  Evaluate the given expression purely for its side-effects. The if the expression is actually pure, then this statement can be safely removed.

- **If**$(cond, trueBlock, falseBlock, merge)$

  Selectively execute either the statements of either $trueBlock$ or $falseBlock$ depending on the value of $cond$. Afterwards, for each SSA variable $x_i = \phi(v_{true}, v_{false})$ in $merge$, assign to x_i the appropriate value from the branch that was taken.

- **ForLoop**$(x, start, stop, step, body, merge)$

  Repeatedly run the block of statements $body$ for a range of values from $start$ to $stop$, incremented by $step$, each getting bound to the variable x. This is equivalent to the Python construct `for in x in xrange(start,stop,step)`. The SSA control-flow merge point, $merge$, is a collection of variables matched with values flowing from before the loop has executed and after.

- **WhileLoop**$(cond, body, merge)$

  Repeatedly evaluate the statements in $body$ until the condition $cond$ becomes false. The SSA Φ-nodes denoted by $merge$ are a collection of bindings $x_i = \phi(e_{\text{before}}, e_{\text{after}})$. Each variable x_i is initialized to e_{before} before the loop executes and then updated to the value of e_{after} after each loop iteration.

- **ParFor**$(f, bounds)$

  Evaluate the given function f for every index value between the starting tuple $(0, \ldots, 0)$ and the stopping values $(bounds_1, \ldots, bounds_n)$. Unlike an ordinary loop, this statement guarantees the lack of dependencies between loop iterations, thus every call to f can potentially be executed in parallel. Higher order array operators such as **Map** and **OuterMap** are are ultimately lowered into **ParFor** statements.

4.3 Array Properties

- **Shape**$(array)$

  Return the shape tuple of the given array.

- **Len**(seq)

  For an array argument, returns **Shape**$(seq)_0$, whereas for a tuple this returns the number of elements in the tuple.

- **Rank**(x)

  The number of dimensions in a scalar. For an array this value is **Len**(**Shape**(x)), whereas for all other types the rank if 0.

- **Strides**$(array)$

  Return a tuple of integer strides for the given array. These are used to compute the addresses of the array's elements. These differ from the strides used by NumPy in that they scale with the number of elements in each dimension, rather than the number of bytes. For example, the memory location accessed by the indexing expression `x[a,j,k]` is:

$$\mathrm{base}(x) + \mathrm{itemsize}(x) \cdot (\mathbf{Strides}(x)_0 \cdot i + \mathbf{Strides}(x)_1 \cdot j + \mathbf{Strides}(x)_2 \cdot k)$$

4.4 Simple Array Operators

These are the first-order array operators used to construct new arrays, views of existing arrays, and simple transformations. Many of these operators exist only at the earlier stages of compilation and are later lowered into some combination of computed array views, specific uses of higher order array operators, and loops.

- **AllocArray**$(shape, \tau_{\text{elt}})$

  Allocate an empty array with dimensions given by the tuple *shape* and whose elements are of type τ_{elt}. Equivalent to the NumPy function `empty` and used by many other array operations after they are lowered.

- **ConstArray**$(value, shape)$

  Construct an array of the give *shape*, all of whose values are the specified *value*. Used in the reimplementation of NumPy functions `np.zeros` and `np.ones`.

- **Reshape**$(array, shape)$

  Create an array view over the same underlying data but with a new shape. The number of elements (given by the product of the shape's dimensions) must be preserved.

- **Transpose**$(array)$

  Create an array view with the same underlying data as *array* but with reversed shape and strides tuples.

- **Ravel**$(array)$

  Linearize a multi-dimensional array into a one-dimensional vector. In the case that some dimension is of unit stride, then the returned value is a reshaped array view. However, if this is not possible, a linearized copy is made.

- **FromDiagonal**$(vec, shape = \text{None}, offset = 0)$

  Given a one-dimensional vector *vec*, construct a higher-dimensional array with zeroes for all its elements except for the diagonal which will be copied from *vec*. By default, the result array is a two-dimensional matrix, but this can be changed by supplying a *shape* argument. If the *offset* argument is positive, then the values of *vec* are copied into a diagonal above the main diagonal. If it's negative, then *vec* gets copied below the main diagonal.

- **ExtractDiagonal**$(array, \mathit{offset} = 0)$

  Given a multidimensional array, return a vector containing the values of its diagonal entries $array[i + \mathit{offset}, \ldots, i + \mathit{offset}]$.

- **Tile**$(array, reps)$

  Repeat the contents of the given *array* as many times along dimension i as given in the tuple element $reps_i$. If **Len**(*reps*) $<$ **Rank**(*array*) then extra 1s are used to extend *reps* from the left. Similarly, if **Len**(*reps*) $>$ **Rank**(*array*) then the shape of the array is extended with 1s.

4.5 Memory Allocation

These constructs are accessible to the programmer but are rather used by later stages of the compilation pipeline.

- **Alloc**(n, τ).

  Returns a pointer to a freshly allocated data buffer with *n* elements of type τ, whose size in bytes is itemsize$(\tau) \cdot n$.

- **Free**(*ptr*).

  Manually deallocate the data buffer at address *ptr*.

- **ArrayView**(*data*, τ, *shape*, *strides*, *offset*)

  Create an array view over the memory address range $(\mathit{data} + \mathit{offset}) : (\mathit{data} + \mathit{offset} + \mathrm{itemsize}(\tau) \cdot \Pi_i \mathit{shape}_i)$. Its shape is given by the tuple argument *shape* and the array strides by the tuple argument *strides*.

4.6 Higher Order Array Operators

Parakeet's representation of its data parallel operations differs from traditional presentations in several ways.

Diversity. The biggest distinction between Parakeet and other languages which use data parallel operators, is the dizzying menagerie of operators that Parakeet uses internally. Many languages include general *Map* and *Reduce* functions, and a few even have something resembling a higher-order prefix-scan. Parakeet, however, also uses many other operators such as *OuterMap*, *IndexMap*, *IndexReduce*, and *Filter*. None of these operators are *essential*, in the sense that you could reimplement them using some combination of first order array constructors and nestings of some smaller core set of higher order operators. However, in the absence of any other forms of abstract iteration, a minimalist approach results in missed opportunities for array optimizations and sub-par performance.

Multiple Array Arguments. Data parallel operators are often presented as consuming a single collection, which is often combined with first-order operators such as `zip` that combine the elements of multiple collections. Parakeet, on the other, allows array operators to consume multiple array arguments directly. Arguments can even differ in dimensionality, so that *Map(f, matrix, vector, axis = 0)* would call f with every row in *matrix* and every element in *vector*.

Axis of Traversal. Since the primary data type in Parakeet is an n-dimensional array (and not some sort of flat sequence or stream) it is necessary for data parallel operators know which dimension(s) they should traverse. This is captured with the *axis* property of every operator, which is either an integer constant indicating a single dimension or the wildcard `None`, means that the operator should apply to every scalar element.

Additional Function Parameters. It is typical for operators such as *Reduce* and *Scan* to be specified using only a binary operator. However, this approach breaks down with multiple array arguments. It is necessary to also specify a transformation which maps from elements of the input arrays to the accumulator type. Additionally, *Scan* is equipped with an *emit* function which transforms from the accumulator type to some potentially different output type.

4.6.1 Mapping Operations

- **Map**(f, $X_1, ..., X_n$, *axis* = None)

  Apply the function f to each element of the array arguments. By default, the value of *axis* argument is `None`, which means that every call to f is given scalar elements of the array arguments. If the *axis* keyword is set to some integer value, then the arguments to f will be slices along that dimension of all the arrays. This can be used, for example, for applying f to all columns or rows of a matrix.

- **OuterMap**(f, $X_1, \ldots, X_n$, *axis* $= 0$). Apply the function f to all combinations of elements between the input arrays.

- **Filter**(f, *pred*, $X_1, \ldots, X_n$, *axis* $= 0$).

  For the aligned input elements $X_1, \ldots, X_n$, construct an array from the results of $f(x_1, \ldots, x_n)$ when $pred(x_1, \ldots, x_n)$ is true and discard the values for which the predicate is false.

- **IndexMap**(f, *shape*).

Similar to a *Map* but evaluates the function f for every index tuple in the iteration space from $0, \ldots, 0$ to $\sigma_1, \ldots, \sigma_n$.

- **IndexFilter**$(f,$ *pred*, *shape*$)$. Similar to **IndexMap** but uses the predicate function *pred* to exclude index combinations.

4.6.2 Reductions

- **Reduce**$(f, \oplus, X_1, \ldots, X_n,$ *init* $=$ None, *axis* $=$ None$)$

  Combine all the elements of the array arguments using the $(n+1)$-ary commutative function f. The *init* keyword is an optional initial value for the reduction. Examples of reductions are the NumPy functions `sum` and `product`.

- **IndexReduce**$(f, \oplus,$ *shape*$)$ Performs a reduction across the index space from $(0, \ldots, 0)$ to the bounds $(\textit{shape}_1, \ldots, \textit{shape}_n)$. The function argument f converts from an input index $(i_1, \ldots, i_n)$ into a value compatible with combine operation $\oplus$.

- **FilterReduce**$(f, \oplus,$ *pred*, $X_1 \ldots X_n)$. Filter the elements of the input arrays using the function *pred*, transform the surviving elements with f and combine them into a single result with the binary function $\oplus$.

- **IndexFilterReduce**$(f, \oplus,$ *pred*, $X_1 \ldots X_n)$.

4.6.3 Scans

- **Scan**$(f, \oplus,$ *emit*, $X_1, \ldots, X_n,$ *init* $=$ None, *axis* $=$ None$)$

  Combine all the elements of the array arguments and return an array containing all cumulative intermediate values of the combination. Examples of scans are the NumPy functions `cumsum` and `cumprod`.

- **IndexScan**$(f, \oplus,$ *emit*, *shape*$)$. Perform an n-dimensional prefix scan over all indices between $(0, \ldots, 0)$ and the n-tuple *shape*. The function f convert from every index input $(i_1, \ldots, i_n)$ to an accumulator value compatible with the binary operator $\oplus$. The last function argument, *emit*, converts from intermediate accumulator values to the output type of the array operator.

4.7 Formal Syntax

To simplify the specification of type inference and various optimizing transformations, we summarize the syntactic elements of the preceding chapters into a more compact formal syntax in Figure 4.1.

$$
\begin{array}{lll}
\text{function} & f ::= & \lambda x_1, \ldots, x_n . s^+ \\
\\
\text{statement} & s ::= & x = e \mid x[e_{\text{idx}}] = e \\
 & & \mid \textbf{expr } e \\
 & & \mid \textbf{return } e \\
 & & \mid \textbf{if } e \textbf{ then } s^+ \textbf{ else } s^+ \textbf{ merge } \Phi^+ \\
 & & \mid \textbf{while } e_{\text{cond}} \textbf{ do } s^+ \textbf{ merge } \Phi^+ \\
 & & \mid \textbf{for } x \textbf{ in } (e_{\text{start}}, e_{\text{stop}}, e_{\text{step}}) \textbf{ do } s^+ \textbf{ merge } \Phi^+ \\
 & & \mid \textbf{parfor}(e_f, e_{\text{bounds}}) \\
\\
\text{SSA merge} & \Phi ::= & x = \phi(e_{\text{left}}, e_{\text{right}}) \\
\\
\text{expression} & e ::= & x \mid \text{const} \mid \text{prim}(e_1, \ldots, e_n) \\
 & & \mid e_1 \times \ldots \times e_n \mid \textbf{proj}(e_{\text{tuple}}, i) \\
 & & \mid [e_1, \ldots, e_n] \mid e_{\text{array}}[e_{\text{idx}}] \\
 & & \mid \textbf{none} \mid \textbf{slice}(e_{\text{start}}, e_{\text{stop}}, e_{\text{step}}) \\
 & & \mid \textbf{closure}(f, e_1, \ldots, e_k) \\
 & & \mid e_f(e_1, \ldots, e_k) \\
 & & \mid \textbf{Map}_\alpha(e_f, e_1, \ldots, e_n) \\
 & & \mid \textbf{OuterMap}_\alpha(e_f, e_1, \ldots, e_n) \\
 & & \mid \textbf{Filter}_\alpha(e_f, e_{\text{pred}}, e_{\text{result}}, e_1, \ldots, e_n) \\
 & & \mid \textbf{Reduce}_\alpha(e_f, e_\oplus, e_{\text{init}}?, e_1, \ldots, e_n) \\
 & & \mid \textbf{FilterReduce}_\alpha(e_f, e_\oplus, e_{\text{pred}}, e_{\text{init}}?, e_1, \ldots, e_n) \\
 & & \mid \textbf{Scan}_\alpha(e_f, e_\oplus, e_{\text{ emit}}, e_{\text{init}}?, e_1, \ldots, e_n) \\
 & & \mid \textbf{IndexMap}(e_f, e_\sigma) \\
 & & \mid \textbf{IndexReduce}(e_f, e_\oplus, e_{\text{init}}?, e_\sigma) \\
 & & \mid \textbf{IndexFilterReduce}(e_f, e_{\text{pred}}, e_\oplus, e_{\text{init}}?, e_\sigma) \\
 & & \mid \textbf{IndexScan}(e_f, e_\oplus, e_{\text{emit}}, e_{\text{init}}?, e_\sigma)
\end{array}
$$

Figure 4.1: Parakeet's Internal Syntax

Chapter 5

Type Inference and Specialization

Once a user-defined function has been translated into Parakeet's intermediate representation, no further actions can be taken until Parakeet encounters some particular arguments for that function. Parakeet uses the types of function arguments as the basis for *specializing* the function, meaning cloning the function body, determining a unique type for every variable, and inserting coercions where it is necessary to convert between types.

The process of type specialization interwines inference (performed via abstract interpretation) and program rewriting (used to disambiguate dynamic features). Every type rule in Parakeet's inference engine consumes an untyped syntactic construction (statement or expression) and returns an equivalent syntax element with type annotations. Inference rules for statements may also destructively modify the type environment.

5.1 Type System

Parakeet's type system (Figure 5.1) is limited to scalars, n-dimensional arrays of scalar elements, tuples of arbitrary values, slice values, a singleton `None` object, and type values (used to represent the `dtype` arguments of NumPy functions). This small collection of possible types is sufficient for a large subset of the NumPy library functions. Notably missing is support for complex numbers structured array elements, various array iterators and grids, all of which would be necessary for a full-featured reimplementation of NumPy.

$$\begin{array}{llll}
\text{scalar} & \mathfrak{s} & ::= & \text{int8} \mid \text{int16} \mid \text{int32} \mid \text{int64} \mid \\
 & & & \text{uint8} \mid \text{uint16} \mid \text{uint32} \mid \text{uint64} \mid \\
 & & & \text{float32} \mid \text{float64} \mid \text{bool} \\
 & & & \\
\text{type} & \tau & ::= & \mathfrak{s} \mid \text{array}(\mathfrak{s}, k) \mid \text{slice} \mid \text{none} \mid \tau_1 \times \ldots \times \tau_n \mid \\
 & & & \text{closure}(f, \tau_1 \times \ldots \times \tau_n) \mid \text{typeval}(\mathfrak{s})
\end{array}$$

Figure 5.1: Types

Parakeet's type system has a rich subtype structure, implementing both the scalar hierarchy of NumPy (Figure 5.2) and broadcasting/rank polymorphism [Sch03]. Subtyping is implemented by inserting coercions [BTCGS91] into the rewritten typed representation of a function. These coercions can be simple casts between scalar values or **Map** operators which eliminate array-oriented polymorphism.

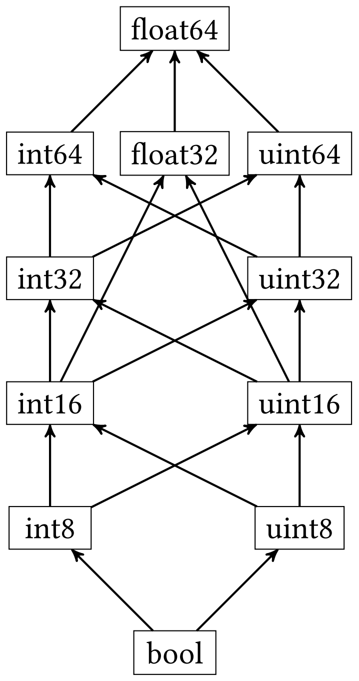

Figure 5.2: Scalar Subtype Hierarchy

5.2 Type Specialization Algorithm

Parakeet's type specializer is presented here in a stylized form as a set of mutually recursive rewrite rules. The entry point into specialization is $\mathcal{S}(f, \tau_1, \ldots, \tau_n)$, defined in Algorithm 5.1, which takes an untyped function and a sequence of input types, and returns a rewritten type-specialized version of the function. If the function argument to $\mathcal{S}$ is a closure rather than a bare function, then the underlying function is extracted and the types of the closed-over arguments are prepended to the sequence of input types.

The case-specific logic of specializing each kind of statement and expression in Parakeet's intermediate language is encompassed by the $[\![\cdot]\!]_{\mathcal{E}}$ operation. The subscripted $\mathcal{E}$ represents the accumulated type environment at the point where each syntactic construct is encountered. A variety of helper functions (shown in Figure 5.3) help in compactly expressing the specialization rules.

The specialization driver $\mathcal{S}$ also depends on the helper routine $\text{InsertCoercions}(\cdot, \mathcal{E})$, which is not listed here. The role of $\text{InsertCoercions}(\cdot, \mathcal{E})$ is to perform a second pass over the freshly annotated representation of a typed function and to insert coercions wherever the locally determined type annotation conflicts with the final types assigned to all variables and return values.

Algorithm 5.1 Specialize Function for Input Types

```
$\mathcal{S}(\lambda x_1, \ldots, x_n . s_1, \ldots s_m, \tau_1, \ldots, \tau_n) \rightsquigarrow$
  $\lambda x_1 : \tau_1, \ldots, x_n : \tau_n . s''_1, \ldots, s''_m$
*where*
  ▷ Fresh type environment containing input types
  $\mathcal{E} = \{x_i \to \tau_i\}$
  ▷ Use special symbol to unify types of exits from function,
  ▷ initialize to the unknown type $\bot$
  $\mathcal{E}[\text{return}] := \bot$
  ▷ Specialize each statement in the function's body
  $s'_i = [\![ s_i ]\!]_{\mathcal{E}}$
  ▷ Rewrite statements to insert casts incorporating
  ▷ global information about variable and return types
  $s''_i = \text{InsertCoercions}(s'_i, \mathcal{E})$
```

5.2.1 Specialization Rules for Statements

Algorithm 5.2 Specialize Expression Statement

$$[\![\textbf{expr } e]\!]_{\mathcal{E}} \rightsquigarrow \textbf{expr} [\![e]\!]_{\mathcal{E}}$$

Algorithm 5.3 Specialize Assignment Statement

```
$[\![x{=}e]\!]_{\mathcal{E}} \rightsquigarrow x : \tau'{=}\mathbf{coerce}(e, \tau')$
*where*
  $e : \tau = [\![e]\!]_{\mathcal{E}}$
  $\tau' = \begin{cases} \mathbf{combine}(\tau, \mathcal{E}[x]), & \text{if } x \in \mathcal{E} \\ \tau, & \text{otherwise} \end{cases}$
  $\mathcal{E}[x] := \tau'$
```

Algorithm 5.4 Specialize Return Statement

```
$[\![\mathbf{return}\ e]\!]_{\mathcal{E}} \rightsquigarrow \mathbf{return} e' : \tau'$
*where*
  $e' : \tau = [\![e]\!]_{\mathcal{E}}$
  $\tau' = \begin{cases} \mathbf{combine}(\tau, \mathcal{E}[\text{return}]), & \text{if return} \in \mathcal{E} \\ \tau, & \text{otherwise} \end{cases}$
  $\mathcal{E}[\text{return}] = \tau'$
```

Algorithm 5.5 Specialize If Statement

```
$[\![\textbf{if } e_{\text{cond}} \textbf{ then } T \textbf{ else } F \textbf{ merge } \{x_i = \phi(e_i^{\text{T}}, e_i^{\text{F}})\}]\!]_{\mathcal{E}} \rightsquigarrow$
    $\textbf{if } e''_{\text{cond}} : \text{bool } \textbf{then } T' \textbf{else } F' \textbf{ merge } \{x_i : \tau_i = \phi(t'_i, f'_i)\}$
where
    $e'_{\text{cond}} : \tau_{\text{cond}} = [\![e_{\text{cond}}]\!]_{\mathcal{E}}$
    ▷ Condition must be a boolean
    $e''_{\text{cond}} : \text{bool} = \textbf{coerce}(e'_{\text{cond}}, \text{bool})$
    ▷ Specialize the statements of the true branch
    $s_1^T, \ldots, s_m^T = T$
    $T' = [\![s_1^T]\!]_{\mathcal{E}} \ldots [\![s_m^T]\!]_{\mathcal{E}}$
    ▷ Specialize the statements of the false branch
    $s_1^F, \ldots, s_n^F = F$
    $F' = [\![s_1^F]\!]_{\mathcal{E}} \ldots [\![s_n^F]\!]_{\mathcal{E}}$
    ▷ Unify the types of values being merged by the SSA nodes
    $t_i : \tau_{ti} = [\![e_i^T]\!]_{\mathcal{E}}$
    $f_i : \tau_{fi} = [\![e_i^F]\!]_{\mathcal{E}}$
    $\tau_i = \begin{cases} \textbf{combine}(\tau_{ti}, \tau_{fi}, \mathcal{E}[x_i]), & \text{if } x \in \mathcal{E} \\ \textbf{combine}(\tau_{ti}, \tau_{fi}), & \text{otherwise} \end{cases}$
    $t'_i = \textbf{coerce}(t_i, \tau_i)$
    $f'_i = \textbf{coerce}(f_i, \tau_i)$
    ▷ Update environment for SSA variable types
    $\mathcal{E}[x_i] := \tau_i$
```

5.2.2 Specialization Rules for Higher Array Operators

To simplify the specification of type inference and specialization rules for array operators, we introduce several helper functions in 5.3. These are used to perform bookkeeping tasks such as increasing or decreasing the ranks of array types and constructing uniform tuples. Since the logic of extracting element values for array operators is complicated by the variety of possible axis arguments (an integer constant, a tuple of integers, or `None`), we also introduce a function specializer (Algorithm 5.6) which extracts element types of arrays depending on a given axis argument α.

Algorithm 5.6 Specialize Function for Elements of Array Arguments

$$\mathcal{S}_\alpha(f, X_1, \ldots, X_n) = \begin{cases} \mathcal{S}(f, \ldots \textbf{eltype}(\tau_i) \ldots), & \text{if } \alpha \text{ is None} \\ \mathcal{S}(f, \ldots \textbf{idxtype}(\tau_i, \alpha_i) \ldots), & \text{if } \alpha \text{ is tuple} \\ \mathcal{S}(f, \ldots \textbf{idxtype}(\tau_i, \alpha) \ldots), & \text{otherwise} \end{cases}$$

Result type

$$\textbf{restype}(f) \quad = \quad \text{result type of the function } f$$

Element type of array

$$\begin{aligned}
\textbf{eltype}(\text{array}(\mathfrak{s}, k)) &= \mathfrak{s} \\
\textbf{eltype}(\tau_1 \times \ldots \times \tau_n) &= \textbf{eltype}(\tau_1) \times \ldots \times \textbf{eltype}(\tau_n) \\
\textbf{eltype}(\tau) &= \tau, \text{ otherwise}
\end{aligned}$$

Array rank

$$\begin{aligned}
\textbf{rank}(\text{array}(\mathfrak{s}, k)) &= k \\
\textbf{rank}(\tau) &= 0, \text{ otherwise}
\end{aligned}$$

Slice type

$$\begin{aligned}
\textbf{idxtype}(\text{array}(\mathfrak{s}, k), \alpha) &= \text{array}(\mathfrak{s}, k - \alpha), \text{ if } \alpha < k \\
\textbf{idxtype}(\text{array}(\mathfrak{s}, k), \alpha) &= \mathfrak{s}, \text{ if } \alpha \geq k \\
\textbf{idxtype}(\tau_1 \times \ldots \times \tau_n, \alpha) &= \textbf{idxtype}(\tau_1) \times \ldots \times \textbf{idxtype}(\tau_n) \\
\textbf{idxtype}(\tau, \_) &= \tau, \text{ otherwise}
\end{aligned}$$

Increase array rank

$$\begin{aligned}
\textbf{uprank}(\text{array}(\mathfrak{s}, k), n) &= \text{array}(\mathfrak{s}, k + n) \\
\textbf{uprank}(\mathfrak{s}, n) &= \text{array}(\mathfrak{s}, n) \\
\textbf{uprank}(\tau_1, \ldots, \tau_n) &= \textbf{uprank}(\tau_1) \times \ldots \times \textbf{uprank}(\tau_n)
\end{aligned}$$

Maximum rank

$$\textbf{maxrank}(\tau_1, \ldots, \tau_n) = \max_i\{\textbf{rank}(\tau_i)\}$$

Repeat type

$$\tau^d = \underbrace{\tau \times \ldots \times \tau}_{\text{tuple of length } d}$$

Figure 5.3: Type Inference Helpers

Algorithm 5.7 Type Specialization For Map

```
$[\![\mathbf{Map}_\alpha(f, X_1, \ldots, X_n)]\!]_\mathcal{E} \rightsquigarrow$
    $\mathbf{Map}_\alpha(f_\tau, X'_1 : \tau_1, \ldots, X'_n : \tau_n) : \tau_{\text{result}}$
$\textit{where}$
    $\triangleright$ Locally infer types of array arguments
    $X'_1 : \tau_1, \ldots, X'_n : \tau_n = [\![X_1]\!]_\mathcal{E}, \ldots, [\![X_n]\!]_\mathcal{E}$
    $\triangleright$ Specialize $f$ for types of elements/slices
    $f_\tau = \mathcal{S}_\alpha([\![f]\!]_\mathcal{E}, \tau_1, \ldots, \tau_n)$
    $\tau_{\text{elt}} = \mathbf{restype}(f_\tau)$
    $\triangleright$ Number of outer dimensions
    $d = \begin{cases} \mathbf{maxrank}(\tau_1, \ldots, \tau_n)), & \text{if } \alpha \text{ is None} \\ 1, & \text{otherwise} \end{cases}$
    $\tau_{\text{result}} = \mathbf{uprank}(\tau_{\text{elt}}, d)$
```

Algorithm 5.8 Type Specialization For IndexMap

```
$[\![\mathbf{IndexMap}(f, \sigma)]\!]_\mathcal{E} \rightsquigarrow$
    $\mathbf{IndexMap}(f_\tau, \sigma' : \text{int64}^d) : \tau_{\text{result}}$
$\textit{where}$
    $\triangleright$ Iteration bounds are an d-dimensional tuple of integers
    $\sigma' : \text{int64}^d = [\![\sigma]\!]_\mathcal{E}$
    $\triangleright$ Index values are the same type as iteration bounds
    $f_\tau = \mathcal{S}([\![f]\!]_\mathcal{E}, \text{int64}^d)$
    $\tau_{\text{elt}} = \mathbf{restype}(f_\tau)$
    $\tau_{\text{result}} = \mathbf{uprank}(\tau_{\text{elt}}, d)$
```

Algorithm 5.9 Type Specialization For OuterMap

```
$[\![\textbf{OuterMap}_\alpha(f, X_1, \ldots, X_n)]\!]_\mathcal{E} \rightsquigarrow$
    $\textbf{OuterMap}_\alpha(f_\tau, X'_1 : \tau_1, \ldots, X'_n : \tau_n) : \tau_\text{result}$
*where*
    $X'_1 : \tau_1, \ldots, X'_n : \tau_n = [\![X_1]\!]_\mathcal{E}, \ldots, [\![X_n]\!]_\mathcal{E}$
    $f_\tau = \mathcal{S}_\alpha([\![f]\!]_\mathcal{E}, \tau_1, \ldots, \tau_n)$
    $\tau_\text{elt} = \textbf{restype}(f_\tau)$
    ▷ Number of outer dimensions
    $d = \begin{cases} \sum_i \textbf{rank}(\tau_i), & \text{if } \alpha \text{ is None} \\ \sum_i 1_{\textbf{rank}(\tau_i) > \alpha_i}, & \text{if } \alpha \text{ is tuple} \\ \sum_i 1_{\textbf{rank}(\tau_i) > \alpha}, & \text{otherwise} \end{cases}$
    $\tau_\text{result} = \textbf{uprank}(\tau_\text{elt}, d)$
```

Algorithm 5.10 Type Specialization For Filter

```
$[\![\textbf{Filter}_\alpha(f, pred, X_1, \ldots, X_n)]\!]_\mathcal{E} \rightsquigarrow$
    $\textbf{Filter}_\alpha(f_\tau, pred'_\tau, X'_1 : \tau_1, \ldots, X'_n : \tau_n) : \tau_\text{result}$
*where*
    $X'_1 : \tau_1, \ldots, X'_n : \tau_n = [\![X_1]\!]_\mathcal{E}, \ldots, [\![X_n]\!]_\mathcal{E}$
    $f_\tau = \mathcal{S}_\alpha([\![f]\!]_\mathcal{E}, \tau_1, \ldots, \tau_n)$
    $pred_\tau = \mathcal{S}_\alpha([\![pred]\!]_\mathcal{E}, \tau_1, \ldots, \tau_n)$
    $pred'_\tau = \lambda x.\textbf{coerce}(pred'_\tau(x), \text{bool})$
    $\tau_\text{elt} = \textbf{restype}(f_\tau)$
    $d = \begin{cases} \textbf{maxrank}(\tau_1, \ldots, \tau_n), & \text{if } \alpha \text{ is None} \\ 1, & \text{otherwise} \end{cases}$
    $\tau_\text{result} = \textbf{uprank}(\tau_\text{elt}, d)$
```

Algorithm 5.11 Type Specialization For IndexFilter

```
$[\![\textbf{IndexFilter}(f, \mathit{pred}, \sigma)]\!]_{\mathcal{E}} \rightsquigarrow$
    $\textbf{IndexFilter}(f_\tau, \mathit{pred}'_\tau, \sigma' : \text{int64}^d) : \tau_{\text{result}}$
$\textit{where}$
    $\sigma' : \text{int64}^d = [\![\sigma]\!]_{\mathcal{E}}$
    $f_\tau = \mathcal{S}([\![f]\!]_{\mathcal{E}}, \text{int64}^d)$
    $\mathit{pred}_\tau = \mathcal{S}([\![\mathit{pred}]\!]_{\mathcal{E}}, \text{int64}^d)$
    $\mathit{pred}'_\tau = \lambda i.\textbf{coerce}(\mathit{pred}'(i), \text{bool}))$
    $\tau_{\text{elt}} = \textbf{restype}(f_\tau)$
    $\tau_{\text{result}} = \textbf{uprank}(\tau_{\text{elt}}, d)$
```

Algorithm 5.12 Type Specialization for Reduce With Initial Value

```
$[\![\textbf{Reduce}_\alpha(f, \oplus, \mathit{init}, X_1, \ldots, X_n)]\!]_{\mathcal{E}} \rightsquigarrow$
    $\textbf{Reduce}_\alpha(f_\tau, \oplus_\tau, \mathit{init}'' : \tau_{\text{acc}}, X'_1 : \tau_1, \ldots, X'_n : \tau_n) : \tau_{\text{acc}}$
$\textit{where}$
    $X'_1 : \tau_1, \ldots, X'_n : \tau_n = [\![X_1]\!]_{\mathcal{E}}, \ldots, [\![X_n]\!]_{\mathcal{E}}$
    ▷ Specialize $f$ for the element types of array arguments
    $f_\tau = \mathcal{S}_\alpha([\![f]\!]_{\mathcal{E}}, \tau_1, \ldots, \tau_n)$
    $\tau_{\text{elt}} = \textbf{restype}(f_\tau)$
    ▷ Infer the type of the user-supplied initial value
    $\mathit{init}' : \tau_{\text{init}} = [\![\mathit{init}]\!]_{\mathcal{E}}$
    ▷ The accumulator is the least upper bound $\tau_{\text{init}}$ and $\tau_{\text{elt}}$
    $\tau_{\text{acc}} = \textbf{combine}(\tau_{\text{init}}, \tau_{\text{elt}})$
    $\oplus_\tau : \tau_{\text{acc}} = \mathcal{S}([\![\oplus]\!]_{\mathcal{E}}, \tau_{\text{acc}}, \tau_{\text{acc}})$
    ▷ Coerce the initial value to be compatible with accumulator type
    $\mathit{init}'' : \tau_{\text{acc}} : \textbf{coerce}(\mathit{init}', \tau_{\text{acc}})$
```

Algorithm 5.13 Type Specialization For Reduce Without Initial Value

```
$[\![\textbf{Reduce}_\alpha(f, \oplus, X_1, \ldots, X_n)]\!]_\mathcal{E} \rightsquigarrow$
    $\textbf{Reduce}_\alpha(f'_\tau, \oplus''_\tau, X'_1 : \tau_1, \ldots, X'_n : \tau_n) : \tau_{\text{acc}}$
*where*
    $X'_1 : \tau_1, \ldots, X'_n : \tau_n = [\![X_1]\!]_\mathcal{E}, \ldots, [\![X_n]\!]_\mathcal{E}$
    $f_\tau = \mathcal{S}_\alpha([\![f]\!]_\mathcal{E}, \textbf{idxtype}(\tau_1, \alpha), \ldots, \textbf{idxtype}(\tau_n, \alpha))$
    $\tau_{\text{elt}} = \textbf{restype}(f_\tau)$
    $\oplus_\tau = \mathcal{S}([\![\oplus]\!]_\mathcal{E}, \tau_{\text{elt}}, \tau_{\text{elt}})$
    $\tau_{\text{acc}} = \textbf{restype}(\oplus_\tau)$
    ▷ Coerce the result of f so it's output is the accumulator type
    $f'_\tau = \lambda x_1, \ldots x_n.\textbf{coerce}(f_\tau(x_1, \ldots, x_n), \tau_{\text{acc}})$
    ▷ Re-specialize the combiner to have correct input types
    $\oplus'_\tau = \mathcal{S}([\![\oplus]\!]_\mathcal{E}, \tau_{\text{acc}}, \tau_{\text{acc}})$
    ▷ Coerce the result of the combiner to match accumulator
    $\oplus''_\tau = \lambda x, y.\textbf{coerce}(\oplus'_\tau(x, y), \tau_{\text{acc}})$
```

Algorithm 5.14 Type Specialization for Scan With Initial Value

```
$[\![\mathbf{Scan}_\alpha(f, \oplus, emit, init, X_1, \ldots, X_n)]\!]_\mathcal{E} \rightsquigarrow$
  $\mathbf{Scan}_\alpha(f_\tau, \oplus_\tau, init'' : \tau_\text{acc}, X'_1 : \tau_1, \ldots, X'_n : \tau_n) : \tau_\text{result}$
*where*
  $X'_1 : \tau_1, \ldots, X'_n : \tau_n = [\![X_1]\!]_\mathcal{E}, \ldots, [\![X_n]\!]_\mathcal{E}$
  ▷ Specialize $f$ for the element types of array arguments
  $f_\tau = \mathcal{S}_\alpha([\![f]\!]_\mathcal{E}, \tau_1, \ldots, \tau_n)$
  $\tau_\text{elt} = \mathbf{restype}(f_\tau)$
  ▷ Infer the type of the user-supplied initial value
  $init' : \tau_\text{init} = [\![init]\!]_\mathcal{E}$
  ▷ The accumulator is the least upper bound $\tau_\text{init}$ and $\tau_\text{elt}$
  $\tau_\text{acc} = \mathbf{combine}(\tau_\text{init}, \tau_\text{elt})$
  $\oplus_\tau : \tau_\text{acc} = \mathcal{S}([\![\oplus]\!]_\mathcal{E}, \tau_\text{acc}, \tau_\text{acc})$
  ▷ Coerce the initial value to be compatible with accumulator type
  $init'' : \tau_\text{acc} = \mathbf{coerce}(init', \tau_\text{acc})$
  $emit_\tau = \mathcal{S}([\![emit]\!]_\mathcal{E}, \tau_\text{acc})$
  $\tau_\text{emit} = \mathbf{restype}(emit_\tau)$
  $d = \begin{cases} \mathbf{maxrank}(\tau_1, \ldots, \tau_n)), & \text{if } \alpha \text{ is None} \\ 1, & \text{otherwise} \end{cases}$
  $\tau_\text{result} = \mathbf{uprank}(\tau_\text{emit}, d)$
```

Algorithm 5.15 Type Specialization For Scan Without Initial Value

```
$[\![\mathbf{Scan}_\alpha(f, \oplus, emit, X_1, \ldots, X_n)]\!]_\mathcal{E} \rightsquigarrow$
    $\mathbf{Scan}_\alpha(f'_\tau, \oplus''_\tau, X'_1 : \tau_1, \ldots, X'_n : \tau_n) : \tau_\text{result}$
*where*
    $X'_1 : \tau_1, \ldots, X'_n : \tau_n = [\![X_1]\!]_\mathcal{E}, \ldots, [\![X_n]\!]_\mathcal{E}$
    $f_\tau = \mathcal{S}_\alpha([\![f]\!]_\mathcal{E}, \mathbf{idxtype}(\tau_1, \alpha), \ldots, \mathbf{idxtype}(\tau_n, \alpha))$
    $\tau_\text{elt} = \mathbf{restype}(f_\tau)$
    $\oplus_\tau = \mathcal{S}(\oplus, \tau_\text{elt}, \tau_\text{elt})$
    $\tau_\text{acc} = \mathbf{restype}(\oplus_\tau)$
    ▷ Coerce the result of f so it's output is the accumulator type
    $f'_\tau = \lambda x_1, \ldots x_n.\mathbf{coerce}(f_\tau(x_1, \ldots, x_n), \tau_\text{acc})$
    ▷ Re-specialize the combiner to have correct input types
    $\oplus'_\tau = \mathcal{S}(\oplus, \tau_\text{acc}, \tau_\text{acc})$
    ▷ Coerce the result of the combiner to match accumulator
    $\oplus''_\tau = \lambda x, y.\mathbf{coerce}(\oplus'_\tau(x, y), \tau_\text{acc})$
    $emit_\tau = \mathcal{S}([\![emit]\!]_\mathcal{E}, \tau_\text{acc})$
    $\tau_\text{emit} = \mathbf{restype}(emit_\tau)$
    $d = \begin{cases} \mathbf{maxrank}(\tau_1, \ldots, \tau_n)), & \text{if } \alpha \text{ is None} \\ 1, & \text{otherwise} \end{cases}$
    $\tau_\text{result} = \mathbf{uprank}(\tau_\text{emit}, d)$
```

Chapter 6

Optimizations

Parakeet aggressively optimizes its intermediate representation before attempting to generate native code. Parakeet's intermediate representation encodes higher-level semantic properties of a user's program which are hard to recover when translated to a backend representation such as C, CUDA, or LLVM. These properties includes the immutability of tuples, algebraic properties of array operators, and iteration independence of parallel constructs such as **ParFor**. Optimizations which take advantage of these properties can often have dramatic impact on overall performance. Additionally, it is important for Parakeet to also perform more traditional compiler optimizations, in order to better reveal opportunities for high-level restructuring.

The overall structure of Parakeet's optimization pipeline is:

1. *High Level Optimizations.* Combine array operators, lower first-order operators such as **Range** into an explicit **Map** or **IndexMap**, lift array operations out of loops (through Loop Invariant Code Motion), specialize functions on arguments with known scalar values. None of these optimizations should significantly decrease the level of abstraction evident in the code.

2. *Indexify.* Normalize parallel array traversals into a smaller set of constructs by converting mapping operations such as **Map**, **OuterMap**, and **IndexMap** into **ParFor** statements. Similarly, convert **Reduce** into **IndexReduce** and **Scan** into **IndexScan**. This simplifies later stages of optimization and allows the backends to consider a smaller set of parallel primitives.

3. *Sequentialize.* This optional stage, used only by the sequential C backend, eliminates all parallel constructs and replaces them with sequential loops.

4. *Lower Indexing.* Transform abstract indexing expressions such as `x[i,j]` into explicit dereferencing of a particular memory location determined by the data pointer and the stride values of the array `x`.

5. *Low-level Optimizations.* At this point, allocation should happen explicitly, indexing transformed into explicit memory accesses and the code has been reduced to either a canonical set of parallel operators or parallelism has been removed altogether. Now perform more classical loop optimizations such as a second round of Loop Invariant Code Motion and Scalar Replacement.

6. *Runtime Value Specialization.* Even after a function is fully optimized, Parakeet will attempt to partially evaluate it for "small" inputs, meaning scalars values equal to 0 or 1, or fields of structures which contain those values. This can sometimes dramatically improve performance by pruning branches of control flow or removing costly indirection.

Parakeet implements this optimization pipeline as a directed acyclic graph of *passes*, each of which can be individually deactivated and is associated with metadata such as:

- Is the optimization idempotent or is it unsafe to cache its results?

- Should a fresh copy of a function be created before the pass modifies it?
- Should the pass be run recursively on all referenced functions?

Once all the optimization passes have executed, the resulting code should have only explicit allocations (no operators which implicitly produce arrays), all indexing should occur through data pointers, and the only optimizations left to perform are those better suited to a compiler for a low level target language.

6.1 Standard Compiler Optimizations

6.1.1 Simplification

Parakeet's simplifier combines Common Subexpression Elimination [Coc70] and Sparse Conditional Constant Propagation [WZ91]. This optimization requires a data-flow analysis to determine which expressions can be considered constants, as well as an online approximation for which expressions have already been computed at each point in the program. Along with Dead Code Elimination, the simplifier is typically run as a clean-up step after every other optimization in the pipeline.

6.1.2 Dead Code Elimination

If an assignment produces a variable which is never used and the computation producing that value is known to lack side-effects, then the assignment can be removed. Similarly, if an SSA Φ-node generates a variable which is never used, then it too can be removed. In Listing 6.1, the first assignment in function `f` can be safely removed, but the second cannot, since the function call to `unsafe` has an observable side-effect.

```
def unsafe(x):
  x[0] = 1
  return x[1]

def f(x):
  a = x[0] + 1
  b = unsafe(x)
```

Listing 6.1: Dead Code Elimination Example

One advantage of using an intermediate representation with first-class array expressions is that Dead Code Elimination can be significantly more powerful than usual. For example, a simple array expression such as **Range** clearly has no observable side-effects and is safe to delete if its result is unused. On the other hand, removing an equivalent operation in C requires analyzing the result's allocation separately from the loop which fills it, and determining that both are safe to delete.

6.1.3 Loop Invariant Code Motion

Often a loop may contain computations which evaluate to the same result for every iteration. For example, in Listing 6.2, both the variable assignments for `a` and `b` can be safely hoisted outside the loop.

```
for i in xrange(n):
  a = np.sqrt(3)
  b = X[0] / a
  c += b
```

Listing 6.2: LICM example

Determining which expressions can be safely moved outside a loop requires first performing a data flow analysis which builds a set of "volatile" values which depend on loop indices (or, recursively, on other volatile values). If an assignment is known to not be volatile, it can safely move to the top of whatever loop it occurs in. This process repeats until each expression is either at the top-level scope of a function or considered volatile.

6.1.4 Scalar Replacement

Scalar replacement [CCK90] is an optimization whereby an array index (e.g. `x[i]`) which is read from and written to repeatedly within a loop is replaced by a scalar variable. This enhances performance both by allowing that a value to stay in a register for longer and also enables other optimizations which only work on scalars. This optimization requires alias analysis to ensure safety, since two seemingly distinct values `x[i]` and `y[i]` may in fact refer to the same location.

6.2 Fusion

In addition to the more traditional compiler optimizations described above, Parakeet performs higher-level restructuring "fusion" optimizations that can dramatically improve performance by eliminating array traversals and the creation of intermediate array values.

In Listing6.3, we show an example of a vector distance function before the application of array operator fusion. Notice the repeated traversals by **Map** operations which create the array temporaries `diff` and `squared`. After the application of fusion, as shown in Listing 6.4, the two **Map** operators have been fused into the **Reduce** operator, using an input transformation function `mapfn`. Three array traversals and two temporaries have been fused into one traversal and no temporaries.

In a loop-oriented imperative language such as Fortran or C, the equivalent to these optimizations would be loop fusion [Yer66]. Since Parakeet provides higher-

```
def dist(X, Y):
  diff = parakeet.map(f = lambda xi,yi: xi - yi, X, Y)
  squared = parakeet.map(f = lambda di: di**2, diff)
  total = parakeet.reduce(combine = add, squared)
  return sqrt(total)
```

Listing 6.3: Distance before Fusion

```
def mapfn(xi, yi):
  return (xi-yi) ** 2

def dist(X, Y):
  total = parakeet.reduce(f = mapfn, combine = add, X, Y)
  return sqrt(total)
```

Listing 6.4: Distance after Fusion

level array-traversing operations, such optimizations can be performed algebraically using fusion rules [JTH01]. Figure 6.1 shows Parakeet's fusion rules in a simplified form, for the special case where array operators only consume a single value. Parakeet's rewrite engine actually generalizes these rules to accept functions of arbitrary input arities. Similarly, we elide any complications arising from array operator axis arguments and optional initial values (for reductions), all of which must also be checked for compatibility.

These fusion transformations are safe if the array created by the source operation is not in any way modified before the operation it is to be fused with. This must also be true for any aliases and slices of this array (necessitating the use of *alias analysis* [GS97] to determine safety). For example, the following statements would not be safe for Fusion:

```
  y = map(f, x)
  y[0] = 0
  z = map(g, y)
```

Listing 6.5: Unsafe for Fusion

Parakeet's fusion optimizations are applied greedily wherever the above safety prerequisites are satisfied. A large body of previous work [AGLP01] has demonstrated that fusion can be applied more effictively when choosing which statements to fuse more with cost-directed strategies. Still, despite the simplicity of Parakeet's approach, fusion has proven both effective and absolutely critical for extracting performant compiled code out of high-level array-oriented user programs. Fusion enables us to boost the computational density of parallel sections, avoid the generation of unnecessary array temporaries, and, for the GPU backend, minimize the number of kernel launches.

6.2.1 Nested Fusion

In addition to "vertical" fusion, where values flow from one statement to another in the same function, it is also desirable to combined *nested* operators. For example, if the parameterizing function of a **Map** also performs a (nested) **Map**, then it might be possible to flatten these two array operators into a single **OuterMap**. A sketch of the nested fusion rules is shown in Figure 6.2. The actual implementation, however, is greatly complicated by the axis parameters, closure arguments, multiple array arguments, and the need to sometimes permute the order arrays to make them compatible with a particular function.

6.2.2 Horizontal Fusion

The previously discussed fusion optimizations have all involved "vertical" data flow from one operator into another. Typically the result of the first operator will no longer be needed after the fusion optimization is applied. In certain cases, it is possible to take two unrelated computations which traverse the same data or indices and combine them into a single array operator. This is done by "tupling" the function parameters of

Map Fusion

$\textbf{Map}(g, \textbf{Map}(f, X)) \rightsquigarrow \textbf{Map}(g \circ f, X)$

IndexMap Fusion

$\textbf{Map}(g, \textbf{IndexMap}(f, \sigma)) \rightsquigarrow \textbf{IndexMap}(g \circ f, \sigma)$
where σ is a shape tuple indicating the index space

OuterMap Fusion

$\textbf{OuterMap}(g, \textbf{Map}(f, X_1), X_2, \ldots, X_n) \rightsquigarrow$
$\quad\textbf{OuterMap}(\lambda x_1, \ldots, x_n.g(f(x_1), x_2, \ldots, x_n), X_1, \ldots, X_n)$

Reduce-Map Fusion

$\textbf{Reduce}(g, \oplus, \textbf{Map}(f, X)) \rightsquigarrow \textbf{Reduce}(g \circ f, \oplus, X)$

Reduce-IndexMap Fusion

$\textbf{Reduce}(g, \oplus, \textbf{Map}(f, \sigma)) \rightsquigarrow \textbf{IndexReduce}(g \circ f, \oplus, \sigma)$

Map-Filter Fusion

$\textbf{Map}(g, \textbf{Filter}(f, \textit{pred}, X)) \rightsquigarrow \textbf{Filter}(g \circ f, \textit{pred}, X)$

Filter-Map Fusion

$\textbf{Filter}(g, \textit{pred}, \textbf{Map}(f, x)) \rightsquigarrow \textbf{Filter}(g \circ f, \textit{pred} \circ f, X)$

Filter-Filter Fusion

$\textbf{Filter}(g, p_2, \textbf{Filter}(f, p_1, X)) \rightsquigarrow \textbf{Filter}(g \circ f, (p_2 \circ f) \wedge p_1, X)$
where $\wedge$ *combines predicates by returning the conjunction of their results*

Reduce-Filter Fusion

$\textbf{Reduce}(g, \oplus, \textbf{Filter}(f, p, X)) \rightsquigarrow \textbf{FilterReduce}(g \circ f, \oplus, p, X)$

Map-Scan Fusion

$\textbf{Map}(g, \textbf{Scan}(f, \oplus, \epsilon, X)) \rightsquigarrow \textbf{Scan}(f, \oplus, g \circ \epsilon, X)$
where ϵ *is the function that generates elements of the result*

Scan-Map Fusion

$\textbf{Scan}(g, \oplus, \epsilon, \textbf{Map}(f, X)) \rightsquigarrow \textbf{Scan}(g \circ f, \oplus, \epsilon, X)$

Figure 6.1: Fusion Rules

reductions, allowing us to construction a single reduction which returns two or more results.

As an example, let's look at Listing 6.6, where two reductions both traverse the same input array.

```
a = parakeet.reduce(f, add, X)
b = parakeet.reduce(g, multiply, Y)
```

Listing 6.6: Before Horizontal Fusion

After Horizontal Fusion (Listing 6.7), we are left with only a single reduction which performs the work of both original operators.

```
def fused_map(xi, yi):
  return f(xi), g(yi)
def fused_combine((xacc, yacc), (x, y)):
  return (xacc + x), (yacc * y)
a,b = parakeet.reduce(fused_map, fused_combine, X, Y)
```

Listing 6.7: After Horizontal Fusion

6.3 Symbolic Execution and Shape Inference

Parakeet's shape inference is an abstract interpretation which computes symbolic relationships between local variables and the scalar components of a function's inputs. In addition to constructing symbolic expressions for the shape of an array operator's outputs, this information can also be used for algebraic simplifications and determining symbolic equality/inequality.

Combine Nested Maps Over Same Array

$\textbf{Map}(\lambda x_i.\textbf{Map}(f, x_i), x) \rightsquigarrow \textbf{Map}(f, x)$

Combine Nested Maps Over Different Arrays

$\textbf{Map}(\lambda x_i.\textbf{Map}(f, y), x) \rightsquigarrow \textbf{OuterMap}(f, x, y)$

Combine Nested IndexMaps

$\textbf{IndexMap}(\lambda i.\textbf{IndexMap}(f, \sigma_2), \sigma_1) \rightsquigarrow \textbf{IndexMap}(f, \sigma_1 +\!\!+ \sigma_2)$

Figure 6.2: Nested Fusion Rules

Horizontal Reduce Fusion

$$\begin{aligned}
&a = \textbf{Reduce}(f, \oplus_1, \textit{init}_1, X)\\
&b = \textbf{Reduce}(g, \oplus_2, \textit{init}_2, X)\\
&\rightsquigarrow\\
&a, b = \textbf{Reduce}(f \times g, \oplus_1 \times \oplus_2, (\textit{init}_1, \textit{init}_2), X)\\
&\textit{where}\\
&\quad f \times g = \lambda x.(f(x), g(x))\\
&\quad \oplus_1 \times \oplus_2 = \lambda((a, b), (c, d)).((a \oplus_1 c), (b \oplus_2 d))
\end{aligned}$$

Horizontal IndexReduce Fusion

$$\begin{aligned}
&a = \textbf{IndexReduce}(f, \oplus_1, \textit{init}_1, \sigma)\\
&b = \textbf{IndexReduce}(g, \oplus_2, \textit{init}_2, \sigma)\\
&\rightsquigarrow\\
&a, b = \textbf{IndexReduce}(f \times g, \oplus_1 \times \oplus_2, (\textit{init}_1, \textit{init}_2), \sigma)\\
&\textit{where}\\
&\quad f \times g = \lambda i.(f(i), g(i))\\
&\quad \oplus_1 \times \oplus_2 = \lambda((a, b), (c, d)).((a \oplus_1 c), (b \oplus_2 d))
\end{aligned}$$

Figure 6.3: Horizontal Fusion Rules

6.4 Value Specialization

A potential source of inefficiency for programs which traverse NumPy arrays is that, by default, all indexing must make use of an auxiliary "strides" array in order to compute the position of elements in the data array. Since indexing is fundamental to all array reads and writes, it is desirable to eliminate the extra arithmetic and indirection associated with strided indexing. It is possible to alleviate some of this inefficiency by partially evaluating [CD93] a function on the stride values of input arrays. More generally, it can be beneficial to perform *runtime value specialization* [BLR10, CASQP13], for scalar inputs, elements of tuples, and array strides.

Parakeet performs this optimization by propagating scalar values from the inputs throughout the source of a function, determining which local variables have statically determinable values and then constructing a cloned copy of the function with these variables replaced by constants. Specialized values are propagated recursively through higher-order array operators, function calls, and **ParFor** statements. Since partially evaluating a function on all possible scalar inputs can result in a wasteful process of never-ending recompilation, Parakeet restricts this optimization to only propagate input values in the set $\{0, 1, 0.0, 1.0, \text{False}, \text{True}\}$, and structures which contain these values. This optimization can be viewed as analogous to creating singleton type for a subset of input values as done in *type directed partial evaluation* [Dan96]

Chapter 7

Evaluation

In this section, we will examine the degree to which Parakeet accelerates a variety of computationally intensive algorithms. The source of each benchmark is given, along with its performance under the standard Python interpreter CPython, as well as performance under all three of Parakeet's backends: the single core C backend, the multi-core OpenMP backend, and the GPU CUDA backend. In addition, we may also compare with any easily-accesible Python library functions which implement the same algorithm.

The degree to which Parakeet accelerates code depends most of all on whether that code spends most of its time in the Python interpreter. In the worst case for Python, when execution time is dominated by some highly optimized low-level routine, then Parakeet may in fact run slower than Python. On the other hand, if a large number of data elements are being inspected within "pure" Python code, then Parakeet may run the same code several thousand times faster.

A secondary consideration when determining to what degree one should expect a speed-up from Parakeet is how much parallelism is available. If an algorithm consists largely of array operations, NumPy library functions, or comprehensions, then Parakeet will be able to extract from all of those data parallel operators. which in term can be compiled into GPU kernels or multi-core saturating OpenMP declarations.

Lastly, we consider the question of Parakeet's backends. By default, Parakeet uses its OpenMP backend. It is possible, on certain examples, to gain a significant performance boost from the GPU. In other examples, however, the GPU can be a performance liability, due to either inefficient CUDA code generation or the overhead of transferring data to and from device memory. Furthermore, some algorithms will simply not run on the GPU at all, particularly those which use more system memory than is available on the device or which require nested memory allocation within a GPU kernel.

All benchmarks presented here were executed on a machine with 4 Xeon W3520 cores running at 2.67GHz, 12GB of memory, and an NVIDIA GTX 580 video card. The software used was Ubuntu Linux 12.10, Python 2.7.6, NumPy 1.8.0, CUDA 5.5, and the development branch of Parakeet 0.25.

7.1 Growcut

Growcut [VK05] is an automata-based image segmentation algorithm. It takes an image and a user-specified mask and then repeatedly applies the automata evolution rule below to derive a final segmentation.

Table 7.1 shows the execution times of this Growcut implementation for 500x500 float64 image inputs. This benchmark represents a best-case scenario for Parakeet's speed-up over Python. In the absence of runtime compilation, all of the computation occurs inside the Python interpreter, resulting in hopelessly slow performance.

```
@jit
def growcut_python(image, state, window_radius):
  height = image.shape[0]
  width = image.shape[1]
  def attack(i,j):
    pixel = image[i, j, :]
    winning_colony = state[i, j, 0]
    defense_strength = state[i, j, 1]
    for jj in xrange(max(j-window_radius,0), min(j+
        window_radius+1, width)):
      for ii in xrange(max(i-window_radius, 0), min(i+
          window_radius+1, height)):
        if ii != i or jj != j:
          d = np.sum((pixel - image[ii, jj, :])**2)
          gval = 1.0 - np.sqrt(d) / np.sqrt(3)
          attack_strength = gval * state[ii, jj, 1]
          if attack_strength > defense_strength:
            defense_strength = attack_strength
            winning_colony = state[ii, jj, 0]
    return np.array([winning_colony, defense_strength])
  return np.array([[attack(i,j)
                for i in xrange(height)]
                for j in xrange(width)])
```

Listing 7.1: Growcut: Automata Evolution Rule

	Execution Time	Speed-up Relative to Python	Compile Time
Python	1268.745s	–	
Parakeet (C)	0.478s	2654	0.714s
Parakeet (OpenMP)	0.134s	9468	0.889s
Parakeet (CUDA)	0.029s	43749	2.428s

Table 7.1: Growcut Performance

7.2 Matrix Multiplication

Matrix multiplication is one of the most heavily optimized algorithms in the whole of numerical computing. The one-line Python implementation show in Listing 7.2 captures the essence of matrix multiplication, unmarred by performance tweaks or optimizations. This style of programming, in fact, is ideal for Parakeet since it reveals parallelizable structure without any potentially confound explicit loops. A well tuned low-level implementation [WD98, GVDG08], on the other hand can consist of thousands of lines of Fortran, sometimes stitching together machine-generated "codelets" through auto-tuning.

```
def matmult(X,Y):
  return np.array([[np.dot(x,y) for y in Y.T] for x in X])
```

Listing 7.2: Matrix Multiplication

Shown in Table 7.2 are the execution times for *1200x1200* input matrices with *float32* element types. The performance under Python is actually surprisingly good, largely since the dot product operation is being performed by NumPy using an efficient implementation. For comparison, we also show the performance of a "purer" Python implementation which defines the dot product in terms Python's builtin `sum` function. We also compare with the performance of a highly tuned multi-core CPU library, ATLAS [WD98] and its GPU equivalent, CUBLAS [BCI$^+$08].

Unsurprisingly, ATLAS's tightly optimized multi-core matrix multiplication algorithm significantly outperform Parakeet's C and OpenMP backends. Similarly, the CUBLAS library outperforms Parakeet's naively generated CUDA code. A significant factor in this performance gap is Parakeet's lack of *cache tiling* [WL91]. We have previously investigated *tiled data parallel operators* [HRS13] which group array inputs into

	Execution Time	Speed-up over Python	Compile Time
Python	17.40s	–	–
Parakeet (C)	12.19s	1.4	0.34s
Parakeet (OpenMP)	3.85s	4.5	0.29s
Parakeet (CUDA)	0.11s	158.2	1.95s
ATLAS	0.40s	43.5	–
CUBLAS	0.008s	2175	–

Table 7.2: Matrix Multiplication Performance

small working sets but such operators are not currently part of Parakeet.

7.3 Rosenbrock Gradient

Howard H. Rosenbrock's "parabolic valley" [Ros60] is commonly used a test function to evaluate the efficacy of non-linear optimization methods. Since many of these optimization techniques rely on repeatedly evaluating the gradient of their objective, it can be useful to accelerate the gradient of the Rosenbrock function. An implementation of the gradient in terms of NumPy array operations is shown in Listing 7.3. Since the actual array arithmetic occurs through compiled primitives, we don't expect Parakeet to outperform Python as dramatically as when core computations occur as loops executed by the Python interpreter.

```
def rosenbrock_gradient(x):
  der = np.empty_like(x)
  der[1:-1] = (+ 200 * (x[1:-1] - x[:-2] ** 2)
               - 400 * (x[2:] - x[1:-1] ** 2) * x[1:-1]
               - 2 * (1 - x[1:-1]))
  der[0] = -400 * x[0] * (x[1] - x[0] ** 2) - 2 * (1 - x[0])
  der[-1] = 200 * (x[-1] - x[-2] ** 2)
  return der
```

Listing 7.3: Gradient of Rosenbrock Function

Table 7.3 shows the execution times for this code on an input array with 10 million float64 elements.

	Execution Time	Speed-up Relative to Python	Compile Time
Python	0.4361s	–	–
Parakeet (C)	0.0631s	6.9	0.277s
Parakeet (OpenMP)	0.0195s	22.4	0.261s
Parakeet (CUDA)	0.008s	54.5	2.249s

Table 7.3: Rosenbrock Derivative Performance

7.4 Image Convolution

Spatial convolution can be used to implement operations such as blurring or sharpening, or for feature extract in machine learning algorithms [LB95]. Listing 7.4 implements the application of a simple $3x3$ convolutional filter to every pixel location in an image.

	Execution Time	Speed-up over Python	Compile Time
Python	10.972s	–	–
Parakeet (C)	0.017s	645	0.761s
Parakeet (OpenMP)	0.008s	1371	0.873s
Parakeet (CUDA)	0.003s	3657	2.323s

Table 7.4: Image Convolution Performance

```
def conv_3x3(image, weights):
  def pixel_result(i,j):
    total = 0
    for ii in xrange(3):
      for jj in xrange(3):
        total += image[i-ii+1, j-jj+1] * weights[ii, jj]
  return np.array([[pixel_result(i,j)
                    for j in xrange(image.shape[1]-1)]
                    for i in xrange(image.shape[0]-1)])
```

Listing 7.4: Nested loops implementation of 3x3 window convolution

7.5 Univariate Regression

Ordinary least squares regression with a single explanatory variable has a particularly elegant closed form solution as the ratio of (1) the covariance of the outputs and inputs with (2) the variance of the inputs. Implemented using NumPy, as in Listing 7.5, this code spends almost no time in the Python interpreter. Nonetheless, as shown in Table 7.5, it is still accelerated by Parakeet due to fusion of array operations and parallel execution.

```python
def covariance(x,y):
  return ((x-x.mean()) * (y-y.mean())).mean()

def fit_simple_regression(x,y):
  slope = covariance(x,y) / covariance(x,x)
  offset = y.mean() - slope * x.mean()
  return slope, offset
```

Listing 7.5: Univariate regression using NumPy operations

	Execution Time	Speed-up over Python	Compile Time
Python	0.380s	–	–
Parakeet (C)	0.056s	6.8	0.353s
Parakeet (OpenMP)	0.014s	27.1	0.401s
Parakeet (CUDA)	0.029s	13.1	2.396s

Table 7.5: Univariate Regression Performance

7.6 Tensor Rotation

The ladder of loops shown in Listing 7.6 implements a rank 4 tensorial rotation. Also shown is a more compact alternative in Listing 7.7 which uses NumPy operations that aren't supported by Parakeet. The performance of the original code under the Python interpreter and Parakeet are compared with the shorter NumPy code in Table 7.6.

```
def rotate(T, g):
  def compute_elt(i,j,k,l):
    total = 0
    for ii in range(n):
      for jj in range(n):
        for kk in range(n):
          for ll in range(n):
            gij = g[ii, i] * g[jj, j]
            gkl = g[kk, k] * g[ll, l]
            total += gij * gkl * T[ii,jj,kk,ll]
    return total
 return np.array([[[[compute_elt(i,j,k,l)
                     for l in xrange(n)]
                     for k in xrange(n)]
                     for j in xrange(n)]
                     for i in xrange(n)])
```

Listing 7.6: Tensor Rotation

```
def rotate(T, g):
  gg = np.outer(g, g)
  gggg = np.outer(gg, gg).reshape(4 * g.shape)
  axes = ((0, 2, 4, 6), (0, 1, 2, 3))
  return np.tensordot(gggg, T, axes)
```

Listing 7.7: Alternative NumPy Tensor Rotation

	Execution Time	Speed-up Relative to Python	Compile Time
Python	108.72s	–	–
Parakeet (C)	0.057s	1907	0.449s
Parakeet (OpenMP)	0.015s	7248	0.578s
Parakeet (CUDA)	0.006s	18120	2.581s
NumPy	0.562s	193	–

Table 7.6: Tensor Rotation Performance

7.7 Harris Corner Detector

The Harris corner detector [HS88] is used as an early step in many image processing algorithms to identify “interesting” regions within an image. It does this by associating with every pixel a *2x2* matrix of products of directional derivatives in pixel space, and deriving a score from the eigenvalues of these matrices. Like other examples where computational cost is dominated by time spent in NumPy libraries, we expect Parakeet to extract modest gains over Python by merging array operations and parallel execution.

```
def harris(I):
  m,n = I.shape
  dx = (I[1:, :] - I[:m-1, :])[:, 1:]
  dy = (I[:, 1:] - I[:, :n-1])[1:, :]
  #
  #   At each point we build a matrix
  #   of derivative products
  #   M =
  #   | A = dx^2     C = dx * dy |
  #   | C = dy * dx  B = dy * dy |
  #
  #   and the score at that point is:
  #      det(M) - k*trace(M)^2
  #
  A = dx * dx
  B = dy * dy
  C = dx * dy
  tr = A + B
  det = A * B - C * C
  k = 0.05
  return det - k * tr * tr
```

Listing 7.8: Harris Corner Detector

Table 7.7 shows the performance of the above code on a *2400x2400* float32 input array. There is a roughly two-fold gain over NumPy simply from fusing array traversals and eliminating array temporaries.

	Execution Time	Speed-up Relative to Python	Compile Time
Python	0.173s	–	–
Parakeet (C)	0.066s	2.6	0.397s
Parakeet (OpenMP)	0.021s	8.2	0.325s
Parakeet (CUDA)	0.004s	43.2	2.23s

Table 7.7: Harris Corner Performance

7.8 Julia Fractal

The Julia Fractal is a visualization of the level sets of an iterated complex mapping. Since Parakeet does not directly support complex numbers, the complex magnitude operation must be explicitly expanded. Since the pixels of the rendered fractal are independent, this algorithm is significantly accelerated both across multiple cores and on the GPU.

```python
def kernel(zr, zi, cr, ci, lim, cutoff):
  count = 0
  while ((zr*zr + zi*zi) < (lim*lim)) and count < cutoff:
      zr, zi = zr * zr - zi * zi + cr, 2 * zr * zi + ci
      count += 1
  return count

def julia(cr, ci, N, bound=1.5, lim=1000., cutoff=1e6):
  grid = np.linspace(-bound, bound, N)
  return np.array([[kernel(x, y, cr, ci, lim, cutoff=cutoff)]
                   for x in grid]
                   for y in grid])
```

Listing 7.9: Julia Fractal

	Execution Time		Speed-up Relative to Python Compile Time
Python	243.841s	–	–
Parakeet (C)	0.122s	1924	0.761s
Parakeet (OpenMP)	0.055s	4269	0.873s
Parakeet (CUDA)	0.012s	19, 570	2.323s

Table 7.8: Julia Fractal Performance

7.9 Smoothed Particle Hydrodynamics

Smoothed Particle Hydrodynamics was originally formulated to simulate the behavior of fluids [MCG03]. Listing 7.10 shows an SPH liquid renderer adapted from a Numba implementation by Rok Roskar. Table 7.9 shows the execution times for rendering *120x120* images with 1600 particles. Unfortunately, this implementation is not amenable to parallel execution without significant restructuring, thus the CUDA and OpenMP execution times are not shown .

	Execution Time	Speed-up Relative to Python	Compile Time
Python	11022.29s	–	–
Parakeet (C)	1.04s	10,598	0.737s

Table 7.9: SPH Renderer Performance

```
def kernel_func(d, h) :
    if d < 1: f = 1.-(3./2)*d**2 + (3./4.)*d**3
    elif d<2: f = 0.25*(2.-d)**3
    else: f = 0
    return f/(np.pi*h**3)

def distance(x,y,z):
    return np.sqrt(x**2+y**2+z**2)

def physical_to_pixel(xpos,xmin,dx):
    return int32((xpos-xmin)/dx)

def pixel_to_physical(xpix,x_start,dx):
    return dx*xpix+x_start

def render_image(xs, ys, zs, hs, qts, mass, rhos, nx, ny, xmin, xmax, ymin,
    ymax):
    MAX_D_OVER_H = 2.0
    image = np.zeros((nx,ny))
    dx = (xmax-xmin)/nx
    dy = (ymax-ymin)/ny
    x_start = xmin+dx/2
    y_start = ymin+dy/2
    kernel_samples = np.arange(0, 2.01, .01)
    kernel_vals = np.array([kernel_func(x,1.0) for x in kernel_samples])
    qts2 = qts * mass / rhos
    for i, (x,y,z,h,qt) in enumerate(zip(xs,ys,zs,hs,qts2)):
      if ((x > xmin-2*h) and (x < xmax+2*h) and
          (y > ymin-2*h) and (y < ymax+2*h) and
          (np.abs(z) < 2*h)):
          if (MAX_D_OVER_H*h/dx < 1 ) and (MAX_D_OVER_H*h/dy < 1):
            xpos = physical_to_pixel(x,xmin,dx)
            ypos = physical_to_pixel(y,ymin,dy)
            xpixel = pixel_to_physical(xpos,x_start,dx)
            ypixel = pixel_to_physical(ypos,y_start,dy)
            dxpix, dypix, dzpix = [x-xpixel,y-ypixel,z]
            d = distance(dxpix,dypix,dzpix)
            if (xpos>0) and (xpos<nx) and (ypos>0) and (ypos<ny) and (d/h<2):
              kernel_val = kernel_vals[int(d/(.01*h))]/(h*h*h)
              image[xpos,ypos] += qt*kernel_val
          else :
              x_pix_start = int32(physical_to_pixel(x-MAX_D_OVER_H*h,xmin,dx))
              x_pix_stop  = int32(physical_to_pixel(x+MAX_D_OVER_H*h,xmin,dx))
              y_pix_start = int32(physical_to_pixel(y-MAX_D_OVER_H*h,ymin,dy))
              y_pix_stop  = int32(physical_to_pixel(y+MAX_D_OVER_H*h,ymin,dy))
              if(x_pix_start < 0):  x_pix_start = 0
              if(x_pix_stop  > nx): x_pix_stop  = int32(nx-1)
              if(y_pix_start < 0):  y_pix_start = 0
              if(y_pix_stop  > ny): y_pix_stop  = int32(ny-1)
              for xpix in range(x_pix_start, x_pix_stop):
                for ypix in range(y_pix_start, y_pix_stop):
                  xpixel = pixel_to_physical(xpix,x_start,dx)
                  ypixel = pixel_to_physical(ypix,y_start,dy)
                  dxpix, dypix, dzpix = [x-xpixel,y-ypixel,z]
                  d = distance(dxpix,dypix,dzpix)
                  if (d/h < 2):
                    kernel_val = kernel_vals[int(d/(.01*h))]/(h*h*h)
                    image[xpix,ypix] += qt*kernel_val
    return image
```

Listing 7.10: SPH Renderer

Chapter 8

Conclusion

This thesis set out to save numerically-inclined programmers from having to write in low-level languages in order to speed up their programs. The existing Parakeet library isn't the final word in this endeavor but rather a guidepost toward the right direction. We have demonstrated that compact high-level descriptions of numerical algorithms can be successfully compiled into efficient code which runs in parallel across multiple cores or on a GPU. Furthermore, we have shown the central role played by higher order data parallel array operators in attaining this goal. The basic contours of Parakeet's design, its variety of data parallel operators, typed intermediate representation, and restructuring optimizations, should be transferrable to other languages beyond Python.

There are several directions along which Parakeet can (and should) be extended:

- **Graph-Based Control Flow Representations**. While the current structured SSA representation was useful for quickly prototyping analyses and optimizations, it prevents the implementation of idiomatic control flow constructs. To support a broader subset of Python, it is important to migrate Parakeet to a proper control flow graph and reimplement all of Parakeet's data flow analyses to work with this more flexible representation.

- **Structured Array Elements**. Parakeet is currently limited to arrays with scalar elements, which forces it to reject many NumPy programs which use structured element types. Parakeet should be able to support not only to support these kinds of arrays, but to efficiently reorganize their layout via the standard *array-of-structs* to *struct-of-arrays* transformation.

- **User-defined Objects**. Parakeet cannot currently accelerate methods or use any sort of user-defined objects within its intermediate representation. It should be possible to partially lift these restrictions by treating objects as immutable records. A more ambitious attempt to fully support the semantics of Python objects would require a dramatic rethinking both of Parakeet's type system and of its assumptions regarding mutability.

- **Data-Dependent Broadcasting**. Parakeet implements a restriction of NumPy's notion of broadcasting. When two values of differing ranks are combined (a matrix and scalar, or a matrix and vector), Parakeet inserts an explicit **Map** operation to implement the logic of broadcasting. However, Parakeet is not able to support broadcasting which occurs due to unit dimensions, such adding a *200x1* matrix with a *1x300* matrix to get a *200x300* result. It should be possible to add dynamic broadcasting as part of the semantics of higher order array operators, though this may incur some additional overhead.

- **Full NumPy Support**. If Parakeet were expanded and improvement in all the ways mentioned above, then it might be possible to fully wrap the NumPy library and support any unmodified code which restricts itself to array operations.

- **Shorter Compile Times for GPU Backend**. Despite the impressive performance attained by the CUDA backend, it is still not ready to be used as the default compilation pathway in Parakeet. One of the major reasons for this are the extremely long compile times incurred through the use of NVIDIA's `nvcc` compiler. In the future, it should be possible to dramatically decrease these compile times by targeting the GPU pseudo-assembly language PTX directly, bypassing the central overhead of CUDA compilation.

- **Better utilizations of shared memory on GPU**. The current compilation template for GPU kernels in Parakeet is very naive and makes no use of the GPU's software managed cache (called *shared memory*). Utilization of this cache is often critical for attaining good performance on the GPU and thus it is strongly desirable for compiled code to do so. One possible avenue by which at least some programs can benefit from shared memory is to identify nested reductions within a parallel operator and to have a distinct compilation template for this case which pre-loads inputs into shared memory and then performs a "tree-structured" reduction.

- **Heterogeneous Execution**. The easiest way to facilitate the use of multiple GPUs, along with multi-core CPUs, is to identify bulk operations which can be concurrently executed and to assign each operation to a particular device. It may also be possible to achieve a more fine-grained heterogeneity, wherein a single operation is split between GPU(s) and CPU cores. This would require a more nuanced notion of where data lives, since portions of an array could be split across multiple memory spaces.

Despite its limitations, however, Parakeet already supports a wide range of numerical Python programs and enables their efficient execution on parallel hardware. The future of scientific programming lies with high-level collection-oriented languages. Runtime compilation techniques, like those found in Parakeet, will be essential for eliminating the cost of abstraction and freeing scientific programmers to simply write what they mean.

Bibliography

[Abr70] Philip Samuel Abrams. *An APL machine.* PhD thesis, Stanford, CA, USA, 1970.

[AGLP01] Marco Aldinucci, Sergei Gorlatch, Christian Lengauer, and Susanna Pelagatti. Towards parallel programming by transformation: the FAN skeleton framework. *Parallel Algorithms Appl.*, 16(2-3):87–121, 2001.

[All70] Frances E. Allen. Control flow analysis. In *Proceedings of a Symposium on Compiler Optimization*, pages 1–19, New York, NY, USA, 1970. ACM.

[AS79] Walid Abdul-Karim Abu-Sufah. *Improving the performance of virtual memory computers.* PhD thesis, Champaign, IL, USA, 1979.

[Ayc03] J. Aycock. A brief history of just-in-time. *ACM Computing Surveys (CSUR)*, 35(2):97–113, 2003.

[BBB+57] John W Backus, Robert J Beeber, Sheldon Best, Richard Goldberg, L Mitchell Haibt, Harlan L Herrick, Robert A Nelson, David Sayre, Peter B Sheridan, H Stern, et al. The FORTRAN automatic coding system. In *Papers presented at the February 26-28, 1957, western joint computer conference: Techniques for reliability*, pages 188–198. ACM, 1957.

[BBB$^+$10] James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. Theano: a cpu and gpu math expression compiler. In *Proceedings of the Python for Scientific Computing Conference (SciPy)*, volume 4, 2010.

[BBC$^+$11] Stefan Behnel, Robert Bradshaw, Craig Citro, Lisandro Dalcin, Dag Sverre Seljebotn, and Kurt Smith. Cython: The best of both worlds. *Computing in Science & Engineering*, 13(2):31–39, 2011.

[BBH$^+$63] David W Barron, John N Buxton, DF Hartley, E Nixon, and Christopher Strachey. The main features of cpl. *The Computer Journal*, 6(2):134–143, 1963.

[BCF$^+$93] Zeki Bozkus, Alok Choudhary, Geoffrey Fox, Tomasz Haupt, and Sanjay Ranka. Fortran 90d/hpf compiler for distributed memory mimd computers: Design, implementation, and performance results. In *Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, pages 351–360. ACM, 1993.

[BCH$^+$94] Guy E. Blelloch, S. Chatterjee, J. C. Hardwick, J. Sipelstein, and M. Zagha. Implementation of a portable nested data-parallel language. *Journal of Parallel and Distributed Computing*, 21(1):4–14, Apr. 1994.

[BCI$^+$08] Sergio Barrachina, Maribel Castillo, Francisco D Igual, Rafael Mayo, and Enrique S Quintana-Orti. Evaluation and tuning of the level 3 cublas for graphics processors. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–8. IEEE, 2008.

[Ber97] Robert Bernecky. Apex: The apl parallel executor. 1997.

[Blo03] Joshua Bloom. Python as super glue for the modern scientific workflow. `http://www.youtube.com/watch?v=mLuIB8aW2KA`, 2003. Accessed: 2013-12-03.

[BLR10] Carl Friedrich Bolz, Michael Leuschel, and Armin Rigo. Towards just-in-time partial evaluation of prolog. In *Logic-Based Program Synthesis and Transformation*, pages 158–172. Springer, 2010.

[BM94] Marc M. Brandis and Hanspeter Mössenböck. Single-pass generation of static single-assignment form for structured languages. *ACM Trans. Program. Lang. Syst.*, 16(6):1684–1698, November 1994.

[Bor08] Jeffrey A. Borror. *Q For Mortals: A Tutorial In Q Programming*. CreateSpace, 2008.

[BTCGS91] Val Breazu-Tannen, Thierry Coquand, Carl A Gunter, and Andre Scedrov. Inheritance as implicit coercion. *Information and computation*, 93(1):172–221, 1991.

[Bud83] Timothy A Budd. An apl compiler for the unix timesharing system. In *ACM SIGAPL APL Quote Quad*, volume 13, pages 205–209. ACM, 1983.

[Bud84] Timothy A. Budd. An APL compiler for a vector processor. *ACM Trans. Program. Lang. Syst.*, 6(3):297–313, July 1984.

[CASQP13] Igor Costa, Péricles Alves, Henrique Nazaré Santos, and Fernando Magno Quintao Pereira. Just-in-time value specialization. In *Code Generation and*

Optimization (CGO), 2013 IEEE/ACM International Symposium on, pages 1–11. IEEE, 2013.

[CCK90] David Callahan, Steve Carr, and Ken Kennedy. Improving register allocation for subscripted variables. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, PLDI '90, pages 53–65, New York, NY, USA, 1990. ACM.

[CD93] Charles Consel and Olivier Danvy. Tutorial notes on partial evaluation. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 493–501. ACM, 1993.

[CF90] David Cann and John Feo. Sisal versus fortran: A comparison using the livermore loops. In *Supercomputing'90. Proceedings of*, pages 626–636. IEEE, 1990.

[CFR$^+$89] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. An efficient method of computing static single assignment form. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 25–35. ACM, 1989.

[CFR$^+$91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13:451–490, October 1991.

[CGK11] Bryan Catanzaro, Michael Garland, and Kurt Keutzer. Copperhead: Compiling an embedded data parallel language. In *Proceedings of the 16th ACM*

Symposium on Principles and Practice of Parallel Programming (PPoPP), pages 47–56, 2011.

[CH] David Cooke and Timothy Hochberg. Numexpr. fast evaluation of array expressions by using a vector-based virtual machine.

[Chi86] Wai-Mee Ching. Program analysis and code generation in an apl/370 compiler. *IBM Journal of Research and Development*, 30(6):594–602, 1986.

[Coc70] John Cocke. Global common subexpression elimination. In *Proceedings of a Symposium on Compiler Optimization*, pages 20–24, New York, NY, USA, 1970. ACM.

[Con] Continuum Analytics. Numba. `http://numba.pydata.org`.

[CP79] S. Cohen and S. Piper. Speakeasy iii reference manual. Speakeasy Computing Corp., Chicago, Ill., 1979.

[Dan96] Olivier Danvy. *Pragmatics of type-directed partial evaluation*. Springer, 1996.

[DM98] Leonardo Dagum and Ramesh Menon. Openmp: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

[FBB06] A. Fitzpatrick, S. Berkovich, and S. Berkovich. Mesm and the beginning of the computer era in the soviet union. *Annals of the History of Computing, IEEE*, 28(3):4–16, 2006.

[FJ05] Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93:216–231, 2005. Special Issue on "Program Generation, Optimization, and Adaptation".

[GBA13] Serge Guelton, Pierrick Brunet, and Mehdi Amini. Compiling Python modules to native parallel modules using Pythran and OpenMP annotations. In *Python for High Performance and Scientific Computing 2013*, 2013.

[GG46] Herman H Goldstine and Adele Goldstine. The electronic numerical integrator and computer (eniac). *Mathematical Tables and Other Aids to Computation*, 2(15):97–110, 1946.

[Gre07] P. Greenfield. Reaching for the stars with python. *Computing in Science Engineering*, 9(3):38–40, 2007.

[GS97] K Gopinath and R Seshadri. Alias analysis for fortran90 array slices. In *Parallel Processing Symposium, 1997. Proceedings., 11th International*, pages 628–632. IEEE, 1997.

[GVDG08] Kazushige Goto and Robert Van De Geijn. High-performance implementation of the level-3 BLAS. *ACM Transactions on Mathematical Software*, 35(1):4:1–4:14, July 2008.

[HI98] Roger KW Hui and Kenneth E Iverson. *J: Dictionary*. Iverson Software, 1998.

[HRS13] Eric Hielscher, Alex Rubinsteyn, and Dennis Shasha. Locality optimization for data parallel programs. *arXiv preprint arXiv:1304.1835*, 2013.

[HS88] Chris Harris and Mike Stephens. A combined corner and edge detector. In *Alvey vision conference*, volume 15, page 50. Manchester, UK, 1988.

[Hun07] John D Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, pages 90–95, 2007.

[IG96] Ross Ihaka and Robert Gentleman. R: A language for data analysis and graphics. *Journal of computational and graphical statistics*, 5(3):299–314, 1996.

[Ive62] Kenneth E. Iverson. A programming language. In *Proceedings of the May 1-3, 1962, spring joint computer conference*, AIEE-IRE ’62 (Spring), pages 345–351, 1962.

[JGM86] Michael A. Jenkins, Janice I. Glasgow, and Carl D McCrosky. Programming styles in nial. *Software, IEEE*, 3(1):46–55, 1986.

[Joh85] Thomas Johnsson. Lambda lifting: Transforming programs to recursive equations. In *Functional programming languages and computer architecture*, pages 190–203. Springer, 1985.

[JOP01] Eric Jones, Travis Oliphant, and Pearu Peterson. Scipy: Open source scientific tools for python. *http://www. scipy. org/*, 2001.

[JTH01] Simon Peyton Jones, Andrew Tolmach, and Tony Hoare. Playing by the rules: rewriting as a practical optimisation technique in GHC. In *Proceedings of the 2004 Haskell Workshop*, 2001.

[Kin12] Volodymyr Kindratenko. Scientific computing with gpus. *Computing in Science & Engineering*, 14(3):8–9, 2012.

[KM93] Ken Kennedy and Kathryn S. Mckinley. Typed fusion with applications to parallel and sequential code generation. Technical report, 1993.

[KRE88] Brian W Kernighan, Dennis M Ritchie, and Per Ejeklint. *The C programming language*, volume 2. prentice-Hall Englewood Cliffs, 1988.

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoff Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems 25*, pages 1106–1114, 2012.

[KY13] Matthew G Knepley and David A Yuen. Why do scientists and engineers need gpu's today? In *GPU Solutions to Multi-scale Problems in Science and Engineering*, pages 3–11. Springer, 2013.

[LB95] Yann LeCun and Yoshua Bengio. Convolutional networks for images, speech, and time series. *The handbook of brain theory and neural networks*, 3361, 1995.

[LS94] Calvin Lin and Lawrence Snyder. Zpl: An array sublanguage. In *Languages and Compilers for Parallel Computing*, pages 96–114. Springer, 1994.

[MAT10] MATLAB. *version 7.10.0 (R2010a)*. The MathWorks Inc., Natick, Massachusetts, 2010.

[McC61] John McCarthy. A basis for a mathematical theory of computation, preliminary report. In *Papers presented at the May 9-11, 1961, western joint IRE-AIEE-ACM computer conference*, pages 225–238. ACM, 1961.

[MCG03] Matthias Müller, David Charypar, and Markus Gross. Particle-based fluid simulation for interactive applications. In *Proceedings of the 2003 ACM*

SIGGRAPH/Eurographics symposium on Computer animation, pages 154–159. Eurographics Association, 2003.

[McK11] Wes McKinney. pandas : powerful python data analysis toolkit, 2011.

[Met91] Michael Metcalf. Why fortran 90? Technical report, CM-P00065587, 1991.

[MGS07] Christopher R Myers, Ryan N Gutenkunst, and James P Sethna. Python unleashed on systems biology. *Computing in Science & Engineering*, 9(3):34–37, 2007.

[Mol80] Cleve B. Moler. MATLAB — an interactive matrix laboratory. Technical Report 369, University of New Mexico. Dept. of Computer Science, 1980.

[NBB$^+$63] Peter Naur, John W Backus, Friedrich L Bauer, Julien Green, Charles Katz, John McCarthy, Alan J Perlis, Heinz Rutishauser, Klaus Samelson, Bernard Vauquois, et al. Revised report on the algorithmic language algol 60. *Communications of the ACM*, 6(1):1–17, 1963.

[NBF96] Bradford Nichols, Dick Buttlar, and Jacqueline Farrell. *Pthreads programming: A POSIX standard for better multiprocessing*. O'Reilly Media, Inc., 1996.

[Nvi11] CUDA Nvidia. Nvidia cuda programming guide, 2011.

[OHL$^+$08] John D Owens, Mike Houston, David Luebke, Simon Green, John E Stone, and James C Phillips. Gpu computing. *Proceedings of the IEEE*, 96(5):879–899, 2008.

[Oli06] Travis E Oliphant. *A Guide to NumPy*, volume 1. Trelgol Publishing USA, 2006.

[PGH11] F. Perez, B.E. Granger, and J.D. Hunter. Python: An ecosystem for scientific computing. *Computing in Science Engineering*, 13(2):13–21, 2011.

[PJR$^+$11] Prakash Prabhu, Thomas B Jablin, Arun Raman, Yun Zhang, Jialu Huang, Hanjun Kim, Nick P Johnson, Feng Liu, Soumyadeep Ghosh, Stephen Beard, et al. A survey of the practice of computational science. In *State of the Practice Reports*, page 19. ACM, 2011.

[Pre03] L. Prechelt. Are scripting languages any good? a validation of Perl, Python, Rexx, and Tcl against C, C++, and Java. *Advances in Computers*, 57:205–270, 2003.

[Pro59] Reese T. Prosser. Applications of boolean matrices to the analysis of flow diagrams. In *Papers Presented at the December 1-3, 1959, Eastern Joint IRE-AIEE-ACM Computer Conference*, IRE-AIEE-ACM '59 (Eastern), pages 133–138, New York, NY, USA, 1959. ACM.

[PVG$^+$11] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in Python. *The Journal of Machine Learning Research*, 12:2825–2830, 2011.

[Rey72] John C Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM annual conference-Volume 2*, pages 717–740. ACM, 1972.

[RHWS12] Alex Rubinsteyn, Eric Hielscher, Nathaniel Weinman, and Dennis Shasha. Parakeet: A just-in-time parallel accelerator for Python. In *Proceedings of the 4th USENIX Conference on Hot Topics in Parallelism (HotPar)*, 2012.

[Ros60] HoHo Rosenbrock. An automatic method for finding the greatest or least value of a function. *The Computer Journal*, 3(3):175–184, 1960.

[RP06] Armin Rigo and Samuele Pedroni. PyPy's approach to virtual machine construction. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 944–953, New York, NY, USA, 2006. ACM.

[Sab88] Gary Sabot. *Introduction to Paralation Lisp*. Thinking Machines Corporation, 1988.

[SB91] Jay Sipelstein and Guy E. Blelloch. Collection-Oriented languages. In *Proceedings of the IEEE*, pages 504–523, 1991.

[Sch03] Sven-Bodo Scholz. Single assignment c: efficient support for high-level array operations in a functional setting. *Journal of Functional Programming*, 13(06):1005–1059, 2003.

[SDSD86] Jacob T Schwartz, Robert B Dewar, Edmond Schonberg, and Ed Dubinsky. *Programming with sets; an introduction to SETL*. Springer-Verlag New York, Inc., 1986.

[SGS10] John E Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(3):66, 2010.

[SJH86] G.L. Steele Jr and W.D. Hillis. Connection machine lisp: Fine-grained parallel symbolic processing. In *Proceedings of the 1986 ACM conference on LISP and functional programming*, pages 279–297. ACM, 1986.

[Tha90] Satish Thatte. Type inference and implicit scaling. In Neil Jones, editor, *ESOP '90*, volume 432 of *Lecture Notes in Computer Science*, pages 406–420. Springer Berlin / Heidelberg, 1990.

[TPO06] David Tarditi, Sidd Puri, and Jose Oglesby. Accelerator: Using data parallelism to program GPUs for general-purpose uses. In *ASPLOS '06: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, November 2006.

[VK05] Vladimir Vezhnevets and Vadim Konouchine. Growcut: Interactive multi-label ND image segmentation by cellular automata. In *Proc. of Graphicon*, pages 150–156, 2005.

[VRDJ95] Guido Van Rossum and Fred L Drake Jr. *Python reference manual.* Centrum voor Wiskunde en Informatica Amsterdam, 1995.

[WD98] R Clint Whaley and Jack J Dongarra. Automatically tuned linear algebra software. In *Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1–27. IEEE Computer Society, 1998.

[WL91] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. In *PLDI '91: Proceedings of the 1991 ACM SIGPLAN Conference on Programming Languages Design and Implementation (PLDI)*, pages 30–44, New York, NY, USA, 1991. ACM.

[Wol99] Stephen Wolfram. *The MATHEMATICA® Book, Version 4.* Cambridge university press, 1999.

[WPD01] Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27:3–35, Jan 2001.

[WZ91] Mark N Wegman and F Kenneth Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(2):181–210, 1991.

[Yer66] A. P. Yershov. ALPHA: An automatic programming system of high efficiency. *J. ACM*, 13(1):17–24, January 1966.

[YIF+08] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. DryadLINQ: a system for general-purpose distributed data-parallel computing using a high-level language. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, pages 1–14, 2008.