# Locality Optimization For Data Parallel Programs

by

Eric Hielscher

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

Courant Institute of Mathematical Sciences

New York University

February 2013

---

Professor Dennis Shasha

© 2013 Eric Hielscher

All Rights Reserved

# Acknowledgements

This thesis would not exist but for the hard work, dedication, and patience of my project partner in crime Alex Rubinsteyn. It's been a long journey together, with much headache and boneheadedness and honest mistakes and successes along the way. This work is very much a joint effort, and it doesn't quite seem fair that we aren't graduating together.

I would also like to thank my adviser Dennis for his patience, kindness, and encouragement. He took me in when I was a second-year student looking for an adviser, and stuck with me even though I turned out to be an easily-distracted amateur musician. I never cease to be amazed at the breadth of topics he has his intelligent hands in.

My beautiful wife Galia also played a key role in making this happen. She's always a source of brightness in my life, and wouldn't let me give up even in the hardest moments.

I would also like to thank my parents for letting me run off from my tiny Minnesota hometown and do all the crazy things I've decided to do. Their total lack of judgment of my actions so long as I am happy has served as wonderful encouragement to try the many beautiful things our world has to offer.

Leslie Cerve, the department secretary, is an absolutely amazing force for good. She fights the good fights on all of our behalves in ways we don't even realize.

Lastly, I'd like to thank all of the other smart and interesting people I've met in my 5+ years at NYU who have helped me along the way and taught me all sorts of cool things. Special thanks go out to the brilliant Nguyen Tran and my good friend Arthur Meacham.

# Abstract

Productivity languages such as NumPy and Matlab make it much easier to implement data-intensive numerical algorithms than it is to implement them in efficiency languages such as C++. This is important as many programmers (1) aren't expert programmers; or (2) don't have time to tune their software for performance, as their main job focus is not programming per se. The tradeoff is typically one of execution time versus programming time, as unless there are specialized library functions or precompiled primitives for your particular task a productivity language is likely to be orders of magnitude slower than an efficiency language.

In this thesis, we present Parakeet, an array-oriented language embedded within Python, a widely-used productivity language. The Parakeet just-in-time compiler dynamically translates whole user functions to high performance multi-threaded native code. This thesis focuses in particular on our use of data parallel operators as a basis for locality enhancing program optimizations. We transform Parakeet programs written with the classic data parallel operators (Map, Reduce, and Scan; in Parakeet these are called *adverbs*) to process small local pieces (called tiles) of data at a time. To express this locality we introduce three new adverbs: TiledMap, TiledReduce, and TiledScan. These tiled adverbs are not exposed to the programmer but rather are automatically generated by a tiling transformation.

We use this tiling algorithm to bring two classic locality optimizations to a data parallel setting: cache tiling, and register tiling. We set register tile sizes statically at compile time, but use an online autotuning search to find good cache tile sizes at runtime. We evaluate Parakeet and these optimizations on a suite of benchmark programs, and exhibit excellent performance even compared to typical C implementations.

# Contents

Acknowledgements . . . . . . . . . . . . . . . . . . . . . . . . . . . . iv

Abstract . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . v

List of Figures . . . . . . . . . . . . . . . . . . . . . . . . . . . . x

List of Tables . . . . . . . . . . . . . . . . . . . . . . . . . . . . . xi

**1 Introduction** **1**

**2 The Parakeet Language and Compiler** **6**

2.1 Using Python for High Performance Computing . . . . . . . . . . . . . 6

2.2 The Parakeet Language . . . . . . . . . . . . . . . . . . . . . . . . 8

2.3 Journey of a Function Through the Parakeet JIT . . . . . . . . . . . . 10

2.4 Adverbs . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 21

**3 Tiling** **28**

3.1 Cache Tiling . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29

3.2 Register Tiling . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32

3.3 Tiled Adverbs . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35

**4 Algorithmic Tiling of Adverbs** **39**

4.1 Tiling Simple Adverb Nestings . . . . . . . . . . . . . . . . . . . . . 39

4.2 The Full Algorithm: Tiling Nested Scalar Statements . . . . . . . . . . . 59

**5 Selecting Tile Sizes 69**

5.1 Register Tile Sizes . . . . . . . . . . . . . . . . . . . . . . . . . . . . 69

5.2 Cache Tile Sizes . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 70

**6 Experimental Evaluation 74**

6.1 Matrix Multiplication . . . . . . . . . . . . . . . . . . . . . . . . . . 75

6.2 K-Means Clustering . . . . . . . . . . . . . . . . . . . . . . . . . . . 82

6.3 Gaussian Blur . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 83

6.4 Compilation Time . . . . . . . . . . . . . . . . . . . . . . . . . . . . 85

**7 Related Work 87**

**8 Future Work 93**

**9 Conclusion 95**

# List of Figures

2.1 Simple Parakeet Function . . . . . . . . . . 9

2.2 Add 1 To Every Element of an Array . . . . . . . . . . 9

2.3 Loopy Version of Parakeet Code to Sum The Number of Elements In An Array Less Than a Given Threshold . . . . . . . . . . 10

2.4 count_thresh abstract syntax tree . . . . . . . . . . 11

2.5 count_thresh Python bytecode . . . . . . . . . . 12

2.6 count_thresh Untyped Intermediate Representation . . . . . . . . . . 14

2.7 Parakeet's Internal Syntax . . . . . . . . . . 15

2.8 count_thresh Type-Specialized Representation . . . . . . . . . . 16

2.9 count_thresh Optimized Parakeet Version . . . . . . . . . . 17

2.10 count_thresh LLVM Assembly Version . . . . . . . . . . 18

2.11 count_thresh x86 Assembly Version . . . . . . . . . . 19

2.12 count_thresh NumPy Assembly Version . . . . . . . . . . 20

2.13 Adverbs in Python . . . . . . . . . . 21

2.14 Type semantics of array element operations . . . . . . . . . . 23

2.15 Internal Representation of Map . . . . . . . . . . 24

2.16 Representation and Semantics of Reduce . . . . . . . . . . 25

2.17 Internal Representation of Scan . . . . . . . . . . 27

3.1 Untiled Matrix Multiply . . . . . . . . . . . . . . . . . . . . . . . . . 29
3.2 Untiled Matrix Multiply, Data Accessed Per Row of Left Matrix . . . . 30
3.3 Tiled Matrix Multiply, Data Accessed Per Row of Left Matrix Tile . . . 31
3.4 Tiled C Matrix Multiply with Constant Tile Size . . . . . . . . . . . . 31
3.5 Register Tiled C Matrix Multiply . . . . . . . . . . . . . . . . . . . . . 33
3.6 Decomposition of an Array $X$ Into $n$ $k$-length Tiles Plus An $s$-length Straggler Tile . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
3.7 Visual Semantics of Tiled Adverbs. For TiledMap, partial results of tiles are concatenated with the $+\!\!+$ operator. For TiledReduce, partial results of tiles are combined with the $\oplus$ function. The semantics presented for TiledScan are inherently sequential as given, but a parallel algorithm is used internally. . . . . . . . . . . . . . . . . . . . . . . . . 36

4.1 Rows Sums Performance with Two Different Memory Layouts . . . . . 41
4.2 Iteration Order of Untiled (left) and Tiled (right) 2D Row Sums . . . . . 42
4.3 Sum Each Row Of A 2D Array . . . . . . . . . . . . . . . . . . . . . . 42
4.4 Type-Annotated Pseudocode for Tiled Sum Each Row of a 2D Array . . 43
4.5 Simplified Tiling Transformation Algorithm, Part 1 . . . . . . . . . . . 44
4.6 Simplified Tiling Transformation Algorithm, Part 2 . . . . . . . . . . . 45
4.7 Two Nested Reductions . . . . . . . . . . . . . . . . . . . . . . . . . . 58
4.8 Full Tiling Transformation Algorithm, Part 1 . . . . . . . . . . . . . . 60
4.9 Full Tiling Transformation Algorithm, Part 2 . . . . . . . . . . . . . . 61
4.10 Full Tiling Transformation Algorithm, Part 3 . . . . . . . . . . . . . . 62
4.11 Indexed Sum of Rows of a 2D Array . . . . . . . . . . . . . . . . . . . 63
4.12 Type-Annotated Pseudocode for Tiled Indexed Sum Each Row of a 2D Array . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 64

5.1 Matrix Multiply Performance vs Tile Size . . . . . . . . . . . . . . . . . 72

6.1 Parakeet Matrix Multiply . . . . . . . . . . . . . . . . . . . . . . . . . 75

6.2 Matrix Multiply Runtimes with Row-Major Layouts . . . . . . . . . . . 77

6.3 Matrix Multiply Runtimes with Cache-Friendly Layouts . . . . . . . . . 78

6.4 Matrix Multiply Performance Compared To C . . . . . . . . . . . . . . . 81

6.5 Gaussian Blur Parakeet Code . . . . . . . . . . . . . . . . . . . . . . . 83

6.6 Impact of Tiling on 11x11 Gaussian Blur Kernel . . . . . . . . . . . . . 85

# List of Tables


2.1 Execution Time of different versions of count_thresh . . . . . . . . . . . 20

6.1 Matrix Multiply Cache-Friendly Layout Cache and Register Tiling Performance . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 79

6.2 Matrix Multiply Row-Major Layout Cache and Register Tiling Performance . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 79

6.3 Matrix Multiply Cache-Friendly Layout Autotuner Performance . . . . 80

6.4 K-Means Performance with k = 1000, 500 features, and 10 iterations . . 82

6.5 Parakeet Compilation Times . . . . . . . . . . . . . . . . . . . . . . . . 86

# Chapter 1

# Introduction

Productivity languages such as NumPy [57] and Matlab [53] make it much easier to implement data-intensive numerical algorithms than it is to implement them in efficiency languages such as C++ [60]. This is important as many programmers (1) aren't expert programmers; or (2) don't have time to tune their software for performance, as their main job focus is not programming. The tradeoff is typically one of execution time versus programming time, as unless there are specialized library functions or precompiled primitives for your particular task a productivity language is likely to be orders of magnitude slower than an efficiency language.

A representative example from the NYU computer science department is a computer vision professor who uses a cluster of computers running Matlab for doing computer vision computations. These computations can run for days or weeks before producing a result, but the professor would prefer to wait this long for results rather than spend the large effort required to write his programs in an efficiency language such as C++ or a specialty language such as CUDA. He and his graduate students simply have better uses of their time such as thinking about computer vision. There are many professionals

who use computers as a tool in this way, and while execution times are indeed of great concern to them, they're not willing to divert their attention to tuning efficiency language versions of their programs.

In this thesis, we present our system Parakeet [66], a just-in-time compiler and runtime system for an array-oriented subset of Python, a widely used productivity language. Parakeet dynamically compiles whole user functions to high performance multi-threaded native code. In particular, this thesis focuses on our use of data parallel operators as a basis for locality enhancing program optimizations [44].

The data parallel model allows programmers to express algorithms by creating and transforming collections using high level constructs. For example, whereas an imperative language would require an explicit loop to sum the elements of an array, a data parallel language might instead implement summation via some form of reduction operator. 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 to parallelization (as the name suggests), both in terms of coarse-grained data partitioning and fine-grained SIMD vectorization.

The first language to feature data parallel abstractions was APL [46], whose central programming constructs involved high-level manipulation of n-dimensional arrays. The eminent parallelizability of the language's core operators inspired early research in vector processors [76] and parallelization [52]. As computers with massively parallel hardware became more common in the 1980s, many languages such as C [43], Fortran [37], and Lisp [73] were retrofitted with data parallel extensions. More recently, data parallel constructs have appeared repeatedly as core primitives for high level languages and libraries which compile to FPGA descriptions [41], GPU programs [20, 75],

and even the coordination of distributed computations [88].

In our system, we transform programs written with the classic data parallel operators (Map, Reduce, and Scan; in our system these are called *adverbs*) to process small local pieces of data at a time [44]. To express this locality we have introduced three new adverbs: TiledMap, TiledReduce, and TiledScan. These tiled adverbs are not exposed to the programmer but rather are automatically generated by a tiling transformation. For programs that exhibit spatial or temporal locality of memory access, applying this transformation can result in signficant performance gains.

We use this algorithm to bring two classic locality optimizations to a data parallel setting: cache tiling, and register tiling. Cache tiling is a technique to break up a computation into subcomputations that only operate on amounts of data at a time that can fit into caches [51, 82]. Our tiled adverbs break up the computations of regular adverbs into locality-friendly pieces called *tiles*.

Much work has gone into developing models that attempt to derive good cache tile sizes statically at compile time [71, 85, 86]. However, the state of the art for achieving top performance remains offline autotuning of programs to find good tile sizes [6, 81]. Offline autotuning searches across different tile sizes by timing the execution of different versions of a program on real hardware. This approach burdens programmers in at least one of two ways: (1) it requires users either to annotate their code manually with information about its tunable parameters, or to implement by hand the hooks which an auto-tuner can use to explore different parameter settings; or (2) it involves a time-consuming offline search that may not be worth the cost for code not reused extensively.

In addition, previous work has shown that a given parallel application – even one as simple as the embarrassingly parallel Black-Scholes option pricing – can have wildly varying performance across different input data sets [61]. Thus statically setting all tun-

ing parameters is not even desirable as performance may not be portable across different inputs.

For these reasons, and since it also fits the standard rapid prototyping workflow of productivity languages well, we adopt a strategy of online autotuning. Tile sizes are left unspecified during compilation. The runtime uses an analytical model similar to those presented in [71] to guess reasonable initial tile size settings. We then use a Gaussian sampling method to perform a search across different tile sizes. In contrast with most offline auto-tuners, every timing run involves useful work that progresses the overall computation. We evaluate the impact various aspects of this search has on performance, including e.g. percentage of the runtime time spent searching and the sampling interval.

In addition to cache tiling, we apply our program transformation a second time to enable another locality optimization called register tiling. The high level concept of register tiling is similar to that of cache tiling – breaking up a computation into small pieces that operate on only so much data as can fit in a small fast memory. In this case, the small fast memory is the set of processor registers, which is typically on the order of a few dozen bytes' worth of space.

Register tiling differs from cache tiling in that once the program is broken up into tiles, a series of further compile-time optimizations are necessary to get the full performance boost. First, the small loops over the data in the tiles need to be completely unrolled. Loop unrolling is a compiler optimization that replaces a loop with a series of repeated statements [33]. Second, an optimization called scalar replacement is applied, whose purpose is to remove direct accesses to memory in these unrolled loop statements [18]. Instead, reads to memory are performed once at the top of the series of statements, and partial results are accumulated directly in registers. Only at the end of the loop are results written back to memory.

Since register tiling involves additional compile-time optimizations after the locality transformation, we would need to compile new version of programs with different tile sizes at runtime. We thus use a heuristic that takes into account the number of registers on the processor to set the register tile sizes once at compile time.

We evaluate our system on a range of input programs written in Parakeet, and show that we can achieve excellent performance as compared with auto-tuned C versions of the same programs. Our tiling optimizations result in large performance gains in different benchmarks. In addition, Parakeet programs are concise and easy to write, abstracting away tedious details and enabling programmers to focus on high level concepts while enjoying excellent performance.

To summarize, the contributions of this thesis are as follows:

1. A novel high-level code transformation that tiles data parallel programs to improve both cache and register usage.

2. An online autotuning search to select tile sizes.

3. Implementation of the above in a just-in-time optimizing, parallelizing compiler for an array-oriented DSL embedded in Python, which can readily outperform naive C implementations on a variety of tasks.

# Chapter 2

# The Parakeet Language and Compiler

In this chapter, we discuss the Parakeet language and compiler. We first discuss the current landscape of using Python for high performance computing in order to provide motivation for our design of Parakeet.

## 2.1 Using Python for High Performance Computing

Python is an extremely popular language for number crunching and data analysis. This may come as a surprise, since Python is actually orders of magnitude slower at simple numerical operations than most lower level languages. If you need to do 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 small set of native machine instructions. The Python "interpreter" (actually a stack-based virtual machine), however, uses a very bulky object representation. Furthermore, Python's dynamicism introduces a lot of indirection around simple operations like getting the value of a field or multiplying two numbers. Every time you execute simple code

such as `x[i] = math.sqrt(y[i] * z.imag)`, a large number of dictionary look-ups, allocations, and other wasteful computations are performed.

The trick, then, to getting good numerical performance from Python is to avoid really doing your work in Python. Instead, you should use Python's remarkable capacity as a glue language to coordinate calls between highly optimized lower-level numerical libraries. NumPy plays an extremely important role in enabling this style of programming by providing a high-level Pythonic interface to an unboxed array that can be passed easily into precompiled C and Fortran libraries.

In order to benefit from NumPy and its vast ecosystem, your algorithm must spend most of its time performing some common operation for which someone has already written an efficient library – for example, an optimized BLAS such as ATLAS for matrix multiplication [81], or FFTW for Fourier transforms [38]. If, however, no one has yet written a library that does the heavy lifting for your particular algorith, the standard solutions all boil down to "implement the bottleneck in C." Even then, as we'll see in later chapters, naively written C can still greatly underutilize a computer's resources. For example, SIMD extensions, multithreading, and general purpose GPU programming are all very difficult compared to Python programming and only expert programmers with a lot of time are able to use them well.

Thus, two main goals of the Parakeet project are thus:

- Find a way to accelerate a meaningfully expressive subset of Python, such that it's possible to continue using convenient abstractions but without a large runtime cost. This generally implies a just-in-time compiler of some sort.

- Exploit the high level representation to enable automatic parallelization and other optimizations.

To be clear, Parakeet's goal isn't to speed up all of Python, though some very excellent work has been done in that area already [15]. Rather, the great thing about many numerically intensive algorithms is that they are remarkably simple. You might get away with using some subset of Python for implementing the core of your computation, and still feel like you are coding at a high level of abstraction (so long as the boundary between the numerical language subset and the rest of Python is mostly seamless).

## 2.2 The Parakeet Language

In this section, we introduce the Parakeet language. We keep the discussion here brief; the purpose is to situate the reader with the rough idea of what programming in Parakeet looks like. In the next section we walk through a detailed example of how a program is compiled.

The idea for Parakeet was originally to provide a JIT compiler for all of NumPy. This proved to be overly ambitious, as NumPy wraps a very large collection of libraries and Parakeet would have to support all of them. We thus stepped back and carved out a general-purpose subset of NumPy that is both likely to be useful in high performance computing and manageable to support. A key aspect of NumPy that we use in parakeet is its array data type. NumPy arrays form the main data type used throughout Parakeet

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

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

Figure 2.1: Simple Parakeet Function

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

Figure 2.2: Add 1 To Every Element of an Array

specialized for the types of the given arguments and then compiled and executed. For example, consider the simple function shown in Figure 2.13.

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, which in Parakeet we call *adverbs* (borrowing terminology from the Q programming language [17]). 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 2.2.

In addition to its core adverbs Map, Reduce, and Scan, Parakeet also supports derived adverbs. For example, the generalized outer product AllPairs is ultimately translated into to a nested pair of maps, but can be more covenient to use.

We use adverbs to provide a convenient place to parallelize and optimize user func-

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

Figure 2.3: Loopy Version of Parakeet Code to Sum The Number of Elements In An Array Less Than a Given Threshold

tions. We discuss these optimizations and adverbs in general in Section 2.4. The Parakeet source is available for download at `http://github.com/iskandr/parakeet`. We are working toward an official, publicized release in the next few months.

## 2.3 Journey of a Function Through the Parakeet JIT

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_thresh` function shown in Figure 2.3, 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. We discuss adverbs later on in Section 2.4.

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_thresh`, 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_thresh` gets read as a string of characters, tokenized, and then turned into a structured syntax tree as shown in Figure 2.4.

A naive interpreter would then execute the syntax tree directly. Python achieves a

```
FunctionDef(
  name='count_thresh',
  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()))])
```

Figure 2.4: count_thresh abstract syntax tree

minor performance boost by instead compiling to a more compact bytecode, as shown in Figure 2.5.

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.

### 2.3.1 What Does Parakeet Do?

Compared with the many other run-time compilation techniques that have been developed over the past decade, Parakeet is a relatively modest function-specializing compiler. If you want Parakeet to compile a particular function, then wrap that function with the `@jit` decorator as shown in Figure 2.3. The job of `@jit` is to intercept calls into the

```
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
```

Figure 2.5: count_thresh Python bytecode

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 any argument types which get passed in.

3. Aggressively optimize the typed code, and translate abstractions such as tuples and $n$-dimensional arrays into simple heap-allocated structures with low-level access code.

4. Translate the optimized and lowered code into LLVM, which we use to perform lower-level optimizations and to generate architecture-specific native code.

The `@jit` cannot be used on any arbitrary Python function to generate an efficient version, since Parakeet is not a general-purpose compiler for all of Python. Parakeet only

supports a handful of Python's data types (numbers, tuples, slices, and NumPy arrays – notably not dictionaries). To manipulate these values, Parakeet lets the programmer use any of the usual math and logic operators, along with some, but not all, of the built-in functions. Functions such as `range` are compiled to deviate from their usual behavior – in Python their result would be a list but in Parakeet such functions create NumPy arrays.

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, or alternatively you could outsource some of your functionality to native code via Cython.

Let's continue with the `count_thresh` compilation example.

### 2.3.2 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.

### 2.3.3 Untyped Representation

In Figure 2.6, we show the `count_thresh` function's internal representation in Parakeet. Notice that the loop counter n has been split apart into three distinct names: `n, n2`,

```
def count_thresh(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
```

Figure 2.6: count_thresh Untyped Intermediate Representation

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 `phi`-node.

In Figure 2.7, we show a formal version of the entire IR syntax of Parakeet.

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.

$$
\begin{array}{lll}
\text{statement} & ::= & \textbf{if } e \textbf{ then } \text{statement}^{+} \textbf{ else } \text{statement}^{+} \\
& & \mid \textbf{while } e_{\text{cond}} \textbf{ do } \text{statement}^{+} \\
& & \mid \textbf{for } x \textbf{ in range } (e_{\text{start}}, e_{\text{stop}}, e_{\text{step}}) \textbf{ do } \text{statement}^{+} \\
& & \mid \text{pattern} \textbf{ = } e \\
& & \mid \textbf{return } e \\
\\
\text{pattern} & ::= & x \mid x[e_{\text{idx}}] \mid \text{pattern}_1, \ldots, \text{pattern}_n \\
\\
\text{expression} & ::= & x \mid \text{const} \mid \text{prim}(e_1, \ldots, e_n) \mid \textbf{none} \\
& & \mid e_1 \times e_2 \mid \textbf{proj}(e_{\text{tuple}}, \text{elt}) \\
& & \mid [e_1, \ldots, e_n] \mid \textbf{index}(e_{\text{array}}, e_{\text{idx}}) \mid \textbf{slice}(e_{\text{start}}, e_{\text{stop}}, e_{\text{step}}) \\
& & \mid \lambda x_1, \ldots, x_n.\text{statement}^{+} \\
& & \mid \mathsf{Map}_{\alpha}(f_{\text{transform}}, e_1, \ldots, e_n) \\
& & \mid \mathsf{Reduce}_{\alpha}(f_{\text{transform}}, f_{\text{combine}}, e_{\text{init}}, e_1, \ldots, e_n) \\
& & \mid \mathsf{Scan}_{\alpha}(f_{\text{transform}}, f_{\text{combine}}, f_{\text{emit}}, e_{\text{init}}, e_1, \ldots, e_n)
\end{array}
$$

Figure 2.7: Parakeet's Internal Syntax

### 2.3.4 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.

A type-specialized version of `count_thresh` is given in Figure 2.8. 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.

```
def count_thresh(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
```

Figure 2.8: count_thresh Type-Specialized Representation

### 2.3.5 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 `PyFloatObject`s.

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 2.9.

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 2.9 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

```
def count_thresh(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
```

Figure 2.9: count_thresh Optimized Parakeet Version

from Parakeet into LLVM.

### 2.3.6 LLVM

LLVM is a well-engineered compiler toolkit which that comes with its a powerful arsenal of optimizations and generates native code for a variety of platforms. To get LLVM to finish the job of compiling `count_thresh`, we need to translate into LLVM's assembly language, shown in Figure 2.10. Once the Parakeet representation has been typed, optimized, and stripped clean of abstractions, the translation to LLVM turns out to be surprisingly easy.

### 2.3.7 Generated x86 Assembly

Once we pass the torch to LLVM, Parakeet's job is mostly done. LLVM performs its own array of optimization passes on the assembly version we give to it. Then LLVM uses a platform-specific back-end to translate from its assembly language into native instructions. And thus, at last, we arrive the native code shown in Figure 2.11.

```
%ArrayT = type { double*, %PyCStructType*, %PyCStructType*,
                 i64, i64 }
%PyCStructType = type { i64 }

define i64 @count_thresh(%ArrayT* nocapture %values.21,
                         double %thresh.22) nounwind {
entry:
  %shape_ptr = getelementptr %ArrayT* %values.21, i64 0, i32 1
  %shape_value = load %PyCStructType** %shape_ptr, align 8
  %data_ptr = getelementptr %ArrayT* %values.21, i64 0, i32 0
  %data_value = load double** %data_ptr, align 8
  %offset_ptr = getelementptr %ArrayT* %values.21, i64 0, i32 3
  %offset_value = load i64* %offset_ptr, align 8
  %elt0_ptr = getelementptr %PyCStructType* %shape_value,
              i64 0, i32 0
  %elt0_value = load i64* %elt0_ptr, align 8
  %enter_cond = icmp sgt i64 %elt0_value, 0
  br i1 %enter_cond, label %loop_body, label %after_loop

loop_body:                            ; preds = %entry, %loop_body
  %n_loop.2.0 = phi i64 [ %add_result5, %loop_body ], [ 0, %entry ]
  %i.2.0 = phi i64 [ %incr_loop_var, %loop_body ], [ 0, %entry ]
  %add_result = add i64 %i.2.0, %offset_value
  %elt_pointer = getelementptr double* %data_value, i64 %add_result
  %elt = load double* %elt_pointer, align 16
  %less_result = fcmp olt double %elt, %thresh.22
  %less.2_val.cast_int64 = zext i1 %less_result to i64
  %add_result5 = add i64 %less.2_val.cast_int64, %n_loop.2.0
  %incr_loop_var = add i64 %i.2.0, 1
  %exitcond = icmp eq i64 %incr_loop_var, %elt0_value
  br i1 %exitcond, label %after_loop, label %loop_body

after_loop:                           ; preds = %loop_body, %entry
  %n_loop.2.1 = phi i64 [ 0, %entry ], [ %add_result5, %loop_body ]
  ret i64 %n_loop.2.1
}
```

Figure 2.10: count_thresh LLVM Assembly Version

```
count_thresh:
;; %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
```

Figure 2.11: count_thresh x86 Assembly Version

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

Figure 2.12: count_thresh NumPy Assembly Version

| CPython | NumPy | Parakeet |
|---|---|---|
| 3.7205s | 0.0036s | 0.0025s |

Table 2.1: Execution Time of different versions of count_thresh

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.

### 2.3.8 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 implemented using NumPy primitives, shown in Figure 2.12.

On 1 million randomly generated inputs, the average the time over 5 runs each of the Python, NumPy, and Parakeet versions took is given in Table 2.1.

Parakeet is about about 1500 times faster than CPython and even manages to edge out NumPy by a safe margin. However, the NumPy code in Figure 2.12 is much more compact than the explicit loop we've been working with throughout this post.

In order to allow Parakeet to compile programs that look more like the NumPy version of `count_thresh`, 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.

## 2.4 Adverbs

The utility of data parallel language constructs has been demonstrated by their use in a long history of collection-oriented languages [72], as well as more recently in embedded domain-specific languages [20, 75] and distributed computing frameworks [32, 88]. Parakeet uses higher order data parallel array operators, more succinctly named *adverbs*, for several important purposes. First, adverbs simplify the expression of an array-oriented programming style by providing language-level constructs for structured array traversal. Second, they allow the Parakeet compiler to perform specialized optimizations on array expressions. Lastly, adverbs are the basic unit which enables parallel execution within Parakeet.

Parakeet's adverbs are exposed to the user as the following Python functions:

```
# transform elements of the input arrays to create a new array
map(transform, a, ..., z, axes = axes)

# combine the elements of the input array into a single value
reduce(combine, x, axes = axes, init = init)

# collect all the sub-results as you combine the elements of x
scan(combine, x,  axes = axes, init = init)

# transform all pairs of elements of the two inputs
allpairs(transform, x, y  axes = axes)
```

Figure 2.13: Adverbs in Python

Each adverb takes as its first argument a parameterizing function, some number of arrays to be transformed, and an optional *axes* argument. An adverb's *axes* indicate along which dimension the adverb should traverse each of its array arguments. For example, applying `parakeet.mean` to a matrix along axis $0$ would yield the average row, whereas the mean along axis $1$ is the average column. Internally, each adverb is represented with a fixed sequence of axes, thus the Python values used for the *axes* argument

must be statically/syntactically computable. The two combining adverbs, Reduce and Scan, both also take *init* arguments, which are the initial values before any array elements have been incorporated into the result.

A programmer uses these adverbs either directly as Python functions or indirecly either through Parakeet's library functions or due to the implicit insertion of Map operators as part of the representation of elementwise operations. For example, imagine a programmer were to implement the following function to compute the squared Euclidean norm of a vector.

```
@parakeet.jit
def sqr_norm(x):
  return parakeet.sum(x**2)
```

Listing 2.1: Definition `norm` using implicit adverbs

The library function `parakeet.sum` is implemented in terms of the Reduce adverb. Ignoring some details (such as the axis of iteration), the definition of `sum` would look like: `parakeet.reduce(lambda acc, xi: acc+xi, x)`. In addition, the elementwise exponentiation `x**2` will be transformed into a Map when the function is converted into Parakeet's typed representation. Thus, the same function could, for the sake of pedagogy, be made more explicit in its use of adverbs had it been written like:

```
@parakeet.jit
def sqr_norm(x):
  squares = parakeet.map(lambda xi: xi**2, x)
  return parakeet.reduce(lambda acc, xi: acc+xi, squares)
```

Listing 2.2: Definition `norm` using explicit adverbs

Though the meaning of adverbs is often intuitive, it is important to better formalize their semantics, especially to clarify the handling of edge cases such as arguments of different ranks. For example, since Parakeet treats array "broadcasting" as an implicit use of adverbs, we would like the expression `map(add, vector, 2.0)` to yield the same result as `vector + 2.0`. What does it mean to map a function over a vector and a scalar? To properly describe the behavior of adverbs, we will first introduce two useful type-level operations:

1. $[\![\tau]\!]_{\downarrow}$: Turns an $n$ dimensional array into an $n-1$ dimensional array, with scalars treated as if they were $0$ dimensional.

2. $[\![\tau]\!]^{\uparrow}$: Collect a set of lower rank $n$ dimensional values into an $n+1$ dimensional array.

A more formal definition of these operations is given in Figure 2.14.

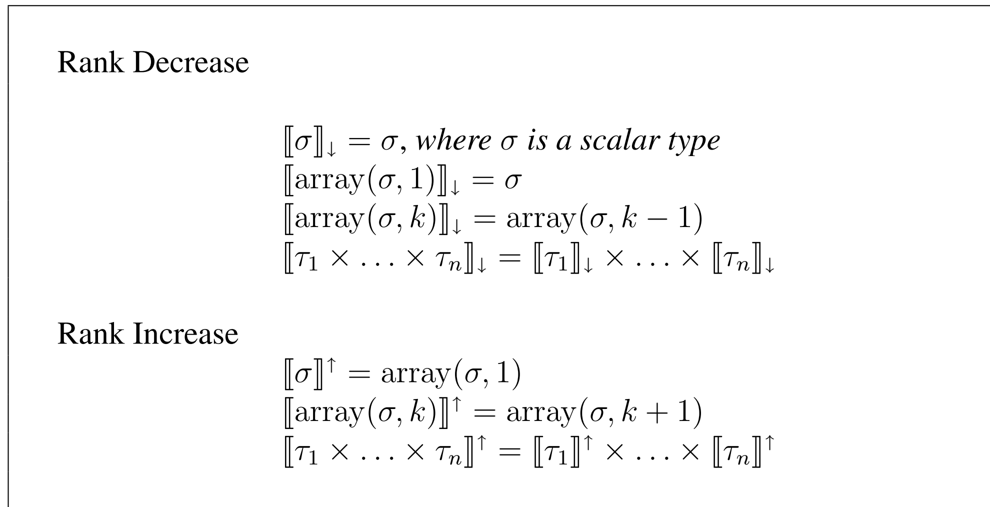



Figure 2.14: Type semantics of array element operations

Untyped representation

$$\mathsf{Map}_\alpha(f_{\text{transform}}, e_1, \ldots, e_n)$$

Simplified semantics

$$\mathsf{Map}(f, e_1, \ldots, e_n)[i] \equiv f(e_1[i], \ldots, e_n[i])$$

Typed representation

$$\mathsf{Map}(f_{\text{transform}}, e_1 : \tau_1 \ldots e_n : \tau_n) : [\![\tau_{out}]\!]^{\uparrow}$$
*where*
$$f_{\text{transform}} : [\![\tau_1]\!]_{\downarrow} \ldots [\![\tau_n]\!]_{\downarrow} \rightarrow \tau_{out}$$

Figure 2.15: Internal Representation of Map

### 2.4.1 Map

Map is the simplest and most common data parallel operator, and its internal representation is very similar to the external API exposed in Python. To Map a function $f_{\text{transform}}$ over $n$ array arguments means to extract their elements, call $f_{\text{transform}}$ with each $n$-tuple of elements, and to collect the results in an array.

The *axes* argument from Map's external interface is preserved as the $\alpha$ parameter of the internal representation. This parameter tells the Map which axis of each of the input arguments be iterated over. For example, imagine that a user wants to map across every 3D element along the second dimension of a single 4D array $X$. In this case, the user would set the `axes` parameter to be `[1]`, as dimensions are 0-indexed and there is only one array. Every argument to an adverb must have the same number of elements along its axis of iteration (unless it is a scalar).

### 2.4.2 Reduce

The most well-known Reduce is probably the `sum` operation, which adds up all the elements of an array. In general, a Reduce takes a function argument $\oplus$ and returns the results of applying $\oplus$ iteratively to an accumulated value and each element of the array. The accumulated value is initialized to the Reduce's argument $e_{\text{init}}$. A sum of a 1D array can be represented by a Reduce with addition as the $\oplus$ function, and 0 as the initial accumulator value. To enable pervasive parallelization we assume that the binary operator $\oplus$ is both commutative and associative. If a programmer wishes to expresses a Reduce-like computation which would be invalid if parallelized, they must do so using an explicit loop instead.

The internal representation of Reduce is slightly more complicated than its external interface. Whereas in Python Reduce takes only one function argument, internally it is represented as using both $f_{\text{transform}}$ (which is a mapping from $n$ arrays to a single accumulator type), as well as a binary combining operator $\oplus$.

Untyped representation

$$\mathsf{Reduce}_{\alpha}(f_{\text{transform}}, \oplus, e_{\text{init}}, e_1, \ldots, e_n)$$

Simplified semantics

$$\begin{aligned}&\mathsf{Reduce}(f, \oplus, e_{\text{init}}, e_1, \ldots, e_n) \equiv \\ &\quad e_{\text{init}} \oplus f(e_1[0], \ldots, e_n[0]) \oplus \cdots \oplus \\ &\quad f(e_1[m-1], \ldots, e_n[m-1])\end{aligned}$$

Typed representation

$$\begin{aligned}&\mathsf{Reduce}_{\alpha}\left(f_{\text{transform}}, \oplus, e_{\text{init}} : \tau_{\text{acc}}, e_1 : \tau_1 \ldots e_n : \tau_n\right) : \tau_{\text{acc}} \\ &\textit{where} \\ &\quad f_{\text{transform}} : [\![\tau_1]\!]_{\downarrow} \ldots [\![\tau_n]\!]_{\downarrow} \rightarrow \tau_{\text{acc}} \\ &\quad \oplus : (\tau_{\text{acc}}, \tau_{\text{acc}}) \rightarrow \tau_{\text{acc}}\end{aligned}$$

Figure 2.16: Representation and Semantics of Reduce

The extra higher order parameter $f_{\text{transform}}$ is used during type specialization for the insertion of any coercions necessary to convert inputs to the same type as the result of $\oplus$. Additionally, Parakeet use this extra function when combining adverbs that share iteration spaces to remove the overhead of having multiple loops [80]. The function parameter of a Map applied to the input of Reduce can be inlined directly into $f_{\text{transform}}$. For example, the inner product of a matrix multiplication can be represented by a single Reduce that has as its $f_{\text{transform}}$ function the multiplication of the rows' and columns' elements, and as its $\oplus$ function addition of these partial results.

### 2.4.3 Scan

A Scan is reminiscent of a Reduce, but rather than returning a single accumulated result, it instead collects every partial accumulator value into an array. While the Scan operator may at first seem like a strange choice for inclusion in Parakeet's core primitives, it has been shown to be very useful as a building block in parallel algorithms [13, 69]. It can, for example, be used to implement any cumulative operators (i.e. `cumsum` and `cumprod`), operations involving neighboring elements of data, and certain sorting algorithms.

In addition to Reduce's input transformer $f_{\text{transform}}$, Scan is also equipped with an output transformer $f_{\text{emit}}$. For example, when computing the cumulative `argmax` of an array, the accumulated value will be a pair containing both a maximum value and the index of that value. The final result of the computation, however, should be an array of indices. The job of $f_{\text{emit}}$ is to extract from the accumulator whatever information should be stored by Scan in an array. The reason Parakeet isolates the combining operator $\oplus$ from $f_{\text{transform}}$ and $f_{\text{emit}}$ is that it's important for $\oplus$ to have a uniform type signature. If

Untyped representation

$$\mathsf{Scan}_\alpha(f_{\text{transform}}, \oplus, f_{\text{emit}}, e_{\text{init}}, e_1, \ldots, e_n)$$

Simplified semantics

$$\mathsf{Scan}(f, \oplus, g, e_{\text{init}}, e_1, \ldots, e_n)[i] = \\ g(e_{\text{init}} \oplus \cdots \oplus f(e_1[i], \ldots, e_n[i])$$

Typed representation

$$\mathsf{Scan}_\alpha\,(f_{\text{transform}}, \oplus, f_{\text{emit}}, e_{\text{init}} : \tau_{\text{acc}}, e_1 : \tau_1 \ldots e_n : \tau_n) : [\![\tau_{\text{out}}]\!]^{\uparrow}$$

*where*

$$f_{\text{transform}} : [\![\tau_1]\!]_{\downarrow} \ldots [\![\tau_n]\!]_{\downarrow} \to \tau_{\text{acc}}$$
$$\oplus : (\tau_{\text{acc}}, \tau_{\text{acc}}) \to \tau_{\text{acc}}$$
$$f_{\text{emit}} : \tau_{\text{acc}} \to \tau_{\text{out}}$$

Figure 2.17: Internal Representation of Scan

the inputs and output of $\oplus$ weren't all the same type, it would not in general be possible to parallelize Scan and Reduce, which brings us to our next topic.

### 2.4.4 Parallelization of Adverbs

The most dramatic gain Parakeet gets from the pervasive use of adverbs is the ability to predictably parallelize code without any form of loop analysis. Since adverbs are defined as declarative data transformations, they don't specify the order in which their operations are to be performed. The implementation of an adverb is free to split up the iterations of an adverb among different threads of execution. Parakeet parallelizes the outermost adverb in any nesting of adverbs in this way, and sequentializes any inner adverbs into loops. This alone results in a very large performance gain over NumPy on general-purpose array code.

# Chapter 3

# Tiling

When the data access pattern of a program involves significant locality – temporal, spatial, or both – this enables a number of different performance optimizations. Temporal locality enables much better data cache behavior, as accesses to a data item after the first can result in relatively cheap cache hits as opposed to expensive reads from RAM. Spatial locality can also improve cache performance, as data is stored in caches in units called cache lines that are typically on the order of 16 words of memory. When data is accessed in a pattern that uses entire cache lines at a time, all but the first read to an item in the line is serviced by a cheap cache hit. Locality is also important for good use of processor registers. It is often very beneficial for performance when data is reused repeatedly in the inner loop of a computation to load a small amount of data into registers and then to perform the inner loop on the registers. This way, every access after the first involves using a register as opposed to a trip to slower levels of the memory hierarchy. Locality can also make SIMD vectorization possible [3, 89], and can enable better use of software-managed fast memories such as shared memory in GPUs [67] or local memory in Cell processors [35]. Chen et al. recently presented a modified MapReduce system

```
void mm(double *A, double *B, double *C, int m, int n, int rowLen) {
  int i, j, k;
  for (i = 0; i < m; ++i) {
    for (j = 0; j < n; ++j) {
      C[i*n + j] = 0.0;
      for (k = 0; k < rowLen; ++k) {
        C[i*n + j] += A[i*rowLen + k] * B[j*rowLen + k];
      }
    }
  }
}
```

Figure 3.1: Untiled Matrix Multiply

that uses tiling for better distributed computing performance [25].

In this chapter, we discuss in detail two such locality optimizations: cache tiling and register tiling. In addition, we describe *tiled adverbs*, a new set of adverbs we introduce to group adverb computations into locality-friendly pieces.

## 3.1 Cache Tiling

Cache tiling is a classic performance optimization that exploits temporal reuse of data to get maximal benefit from data caches [51, 82]. Many numerical computations, including e.g. dense matrix multiplication, involve significant data reuse.

For example, consider the C code for matrix multipication given in Figure 3.1. This is often referred to in the literature as the "naive C" version – i.e., the version one would write most naturally when not thinking explicitly of tuning for performance.

If the `A` and `B` matrices are large, then this code will have poor cache behavior and thus suboptimal performance. To understand why, consider the data access pattern. The columns of `B` are accessed repeatedly, and thus could benefit from being cached to lower their access times. However, between each repeated access to a specific element of `B`, the entire rest of `B` is accessed, in addition to an entire row of both `A` and the output matrix

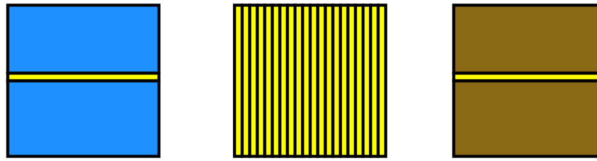


Figure 3.2: Untiled Matrix Multiply, Data Accessed Per Row of Left Matrix

`C`. This set of data is shown in yellow in Figure 3.2. Since per hypothesis `B` is large and doesn't fit entirely in cache, each column of `B` will be evicted from the cache between each use, and it will have to be read from a slower level of the memory hierarchy.

A better access pattern is one that accesses pieces (called tiles) of the input data at a time such that as much data as possible remains resident in cache between repeated uses. Consider the visualization of a tiled organization of matrix multiply in Figure 3.3. The rows of the `A` matrix are grouped into tiles, and each `A` tile is entirely consumed before moving onto the next one. An analagous grouping is done with `B`'s columns. Finally, the iteration through each group of rows and columns is broken up into tiles as well. Code for this tiled version is given in Figure 3.4, where we see that each loop in the original code is broken up into both an outer loop that iterates across tiles and an inner loop that iterates across elements of the current tile.

Consider the data access pattern in this version. Now rather than the entire matrix `B` being accessed between each access to a given element of `B`, only an entire *tile* of `B` is accessed. If the tile size is small enough that this tile fits in cache, repeated accesses to elements of `B` will be much faster in this version as the elements will stay resident in cache. The tile size that works best depends on the size of the cache, the size of the data, and the particular code being executed.

Cache tiling need not be for a single level of cache; tiling loops can be added for each level of cache in which to keep data resident between accesses. In addition, tiling can be used to keep data resident in other levels of the memory hierarchy such as registers.

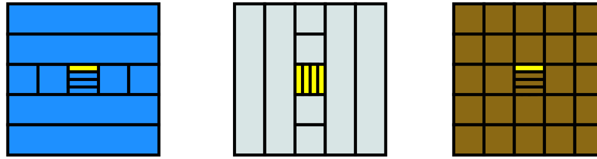


Figure 3.3: Tiled Matrix Multiply, Data Accessed Per Row of Left Matrix Tile

```
const int tileSize = 60;

void mm(double *A, double *B, double *C, int m, int n, int rowLen) {
  int it, jt, kt, i, j, k;
  for (it = 0; it < m; it += tileSize) {
    for (jt = 0; jt < n; jt += tileSize) {
      for (kt = 0; kt < rowLen; kt += tileSize) {
        for (i = it; i < it + tileSize; ++i) {
          for (j = jt; j < jt + tileSize; ++j) {
            C[i*n + j] = 0.0;
            for (k = kt; k < kt + tileSize; ++k) {
              C[i*n + j] += A[i*rowLen + k] * B[j*rowLen + k];
            }
          }
        }
      }
    }
  }
}
```

Figure 3.4: Tiled C Matrix Multiply with Constant Tile Size

## 3.2 Register Tiling

Register tiling is very similar in spirit to cache tiling, but instead of tiling to get better reuse of data in caches, we tile so as to get better reuse of data in processor registers [19, 55]. Even though L1 data caches have very low latencies compared to RAM, their latencies are still considerably worse than processor registers. Thus, for tight inner loops it can be very beneficial for performance to tile data for processor registers as well.

Tiling for registers involves similar steps to cache tiling, but then requires further compile-time optimizations to work. We still divide the input data into tiles, but we have to fix these tile sizes at compile time. A typical Intel processor has roughly 16 floating point registers, and our goal is to have the inner loop fill as many of these registers as possible without using any extras. This way, the compiler won't have to spill the registers onto the stack, which would defeat the purpose of the tiling procedure.

Reference code for a register tiled C implementation of matrix multiply is shown in Figure 3.5. In this example, the register tile sizes for the two outer loops are each 3, while the register tile size for the inner loop is set to 1. Thus the outer loops sweep across the rows of each of the `A` and `B` matrices 3 at a time, while the `k` loop that iterates through the rows has step 1.

Recall from the cache tiling example in Figure 3.4 that there were 3 outer loops for the tiles, and 3 inner loops to perform a miniature matrix multiply on the tiles. In register tiling, we perform an additional compile-time optimization called loop unrolling on the miniature matrix multiply's loops [33]. Loop unrolling takes some number of iterations of a loop and creates separate, explicit statements for each of these iterations rather than leaving them in the loop. There are many benefits of loop unrolling, including reduced

```
void register_tiled_mm(double *A, double *B, double *C,
                       int m, int n, int rowLen) {
  int i, j, k;
  for (i = 0; i < m - 2; i += 3) {
    double *Arow = A + i*rowLen;
    double *Orow = O + i*n;
    for (j = 0; j2 < n - 2; j += 3) {
      double *Brow = B + j*rowLen;
      double *Ocol = Orow + j;
      double c0, c1, c2, c3, c4, c5, c6, c7, c8;
      c0 = c1 = c2 = c3 = c4 = c5 = c6 = c7 = c8 = 0.0;
      for (k = 0; k < rowLen; ++k) {
        double a0 = Arow[k];
        double a1 = Arow[rowLen + k];
        double a2 = Arow[2*rowLen + k];
        double b0 = Brow[k];
        double b1 = Brow[rowLen + k];
        double b2 = Brow[2*rowLen + k];
        c0 = c0 + (a0 * b0);
        c1 = c1 + (a0 * b1);
        c2 = c2 + (a0 * b2);
        c3 = c3 + (a1 * b0);
        c4 = c4 + (a1 * b1);
        c5 = c5 + (a1 * b2);
        c6 = c6 + (a2 * b0);
        c7 = c7 + (a2 * b1);
        c8 = c8 + (a2 * b2);
      }
      Ocol[0*n + 0] = Ocol[0*n + 0] + c0;
      Ocol[0*n + 1] = Ocol[0*n + 1] + c1;
      Ocol[0*n + 2] = Ocol[0*n + 2] + c2;
      Ocol[1*n + 0] = Ocol[1*n + 0] + c3;
      Ocol[1*n + 1] = Ocol[1*n + 1] + c4;
      Ocol[1*n + 2] = Ocol[1*n + 2] + c5;
      Ocol[2*n + 0] = Ocol[2*n + 0] + c6;
      Ocol[2*n + 1] = Ocol[2*n + 1] + c7;
      Ocol[2*n + 2] = Ocol[2*n + 2] + c8;
    }
    /* Cleanup code for the j tile goes here. */
  }
  /* Cleanup code for the i tile goes here. */
}
```

Figure 3.5: Register Tiled C Matrix Multiply

loop overhead (updating the index variable and potential missed branch predictions) as well as allowing better pipelined use of the processor's functional units. All modern production compilers such as gcc include support for loop unrolling.

In this case, we unroll the loops to try to keep the data they access explicitly in registers. For example, in Figure 3.5, notice how we read the three values from each of the `A` and `B` arrays' tiles into the registers `a0, a1, a2, b0, b1,` and `b2`. Then these registers are used instead of direct reads to the `A` and `B` array elements in memory.

In reality, after unrolling each of the statements computing a partial inner product such as `c0 = c0 + (a0 * b0)` would still contain a direct read to memory after the unrolling step. This statement would look like `Ocol[0*n + 0] = Ocol[0*n + 0] + Arow[k] * Brow[k]` after the unrolling step alone. In order to push the memory accesses to before and after the inner loop as they are in the figure, we need to apply another compile-time optimization called scalar replacement. Scalar replacement scans a loop body for loop-invariant memory accesses and moves them outside the loop. It also replaces repeated reads to a single memory location by one read that stores the value in a register, and then repeated reads from the register. Scalar replacement succeds in moving the stores to the output array to after the loop and the reads from the `A` and `B` to be done only once.

The final version of the code as shown in Figure 3.5 is tiled perfectly for registers after these two steps. We initialize the 9 running partial sums in registers before the inner loop begins. We read each of the three values from the `A` and `B` arrays' tiles into 6 registers, and then we compute the partial results of the inner product using these registers. The running sum is kept entirely in registers for the duration of the loop. Only after the loop are the partial sums added to the output array.

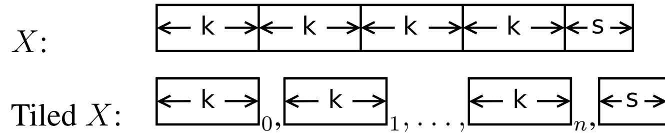



Figure 3.6: Decomposition of an Array $X$ Into $n$ $k$-length Tiles Plus An $s$-length Straggler Tile

## 3.3 Tiled Adverbs

In order to provide a convenient single abstraction for dividing adverb computations into locality-friendly pieces, we introduce *tiled adverbs*, one for each regular adverb. Tiled adverbs are a natural generalization of adverbs that, rather than applying their transform function to each element of their input arrays, instead break their input arrays up into groups of elements of bounded size called tiles and execute their transform functions on these tiles. Users never program directly with tiled adverbs – they are strictly internal syntax for use in locality optimizations. The Parakeet compiler automatically generates them from untiled code via a tiling transformation described later in Chapter 4.

Figure 3.6 shows the decomposition of an array $X$ into $n$ tiles each of length $k$, with an additional tile of length $s$ that contains the leftover elements of $X$ if its length isn't evenly divisible by $k$. Figure 3.7 gives a visualization of the semantics of tiled adverbs. Tiled adverbs can take multiple arguments and axes just like regular adverbs, but we omit them from this picture for clarity. As an example, a TiledMap decomposes its input arguments into tiles, executes its transform function on each tile (including the straggler tile), and then concatenates the results. The semantics of TiledReduces and TiledScans are similarly direct generalizations of their untiled counterparts.

Tiled adverbs also take two versions of their transform functions – one specialized for the specific tile size $k$ (denoted $f^k$), and one generic to array length (simply denoted

- TiledMap($f$, $f^k$, $X$):
  $f^k(\boxed{\leftarrow \mathsf{k} \rightarrow}_0) +\!\!+ \ldots +\!\!+ f^k(\boxed{\leftarrow \mathsf{k} \rightarrow}_n) +\!\!+ f(\boxed{\leftarrow \mathsf{s} \rightarrow})$
- TiledReduce($f$, $f^k$, $\oplus$, $X$):
  $f^k(\boxed{\leftarrow \mathsf{k} \rightarrow}_0) \oplus \cdots \oplus f^k(\boxed{\leftarrow \mathsf{k} \rightarrow}_n) \oplus f(\boxed{\leftarrow \mathsf{s} \rightarrow})$
- TiledScan($f$, $f^k$, $\oplus$, $X$):
  $result_0 := f^k(\boxed{\leftarrow \mathsf{k} \rightarrow}_0)$
  $result_{i=1..n} :=$ last el. of $result_{i-1} \oplus f^k(\boxed{\leftarrow \mathsf{k} \rightarrow}_i)$
  $result_{n+1} :=$ last el. of $result_n \oplus f(\boxed{\leftarrow \mathsf{s} \rightarrow})$
  **return** $result_0 +\!\!+ \ldots +\!\!+ result_{n+1}$

Figure 3.7: Visual Semantics of Tiled Adverbs. For TiledMap, partial results of tiles are concatenated with the $+\!\!+$ operator. For TiledReduce, partial results of tiles are combined with the $\oplus$ function. The semantics presented for TiledScan are inherently sequential as given, but a parallel algorithm is used internally.

$f$) which is used to process the straggler tile. Knowledge that the $f^k$ transform function will be called only on arrays of a fixed size allows Parakeet to optimize it in various ways, for example by removing boundary checks. In addition, register tiling optimizes the $f^k$ functions even further.

In the cases of TiledReduce and TiledScan, we make integral use of the fact that the transform functions and combine functions for adverbs are separated, as discussed in Section 2.4. The transform functions $f$ and $f^k$ are the ones used to perform the computation on each tile, while the combine functions $\oplus$ are used to combine the results of computing on each tile. Thus our internal representation for adverbs already lends itself naturally to locality considerations.

Decomposing adverbs in this way allows the Parakeet compiler to perform both cache tiling and register tiling via the same abstraction. The decomposition step alone is enough to enable cache tiling, provided that the values of $k$ are chosen properly such

that the working set of each function call fits into cache. When Parakeet tiles for cache locality, the values of $k$ are left undetermined until runtime when an online search is performed to choose good values for them.

By contrast, in order for Parakeet to use tiled adverbs to perform register tiling, we fix the $k$ values to small compile-time constants based on a heuristic that takes into account the number of registers on the target machine (discussed in more detail in Chapter 5). Parakeet then lowers $f^k$ into a loop and completely unrolls it, which is possible due to its fixed length. Afterward, scalar replacement is applied to remove as many direct memory accesses to the tile's elements as possible, instead keeping them in registers.

Implementing tiling via tiled adverbs was relatively simple and required only roughly 500 lines of Python code. Tiled adverbs have simple, high level semantics that allowed us to reason easily about the tiling process. By contrast, tiling methods that operate directly on loops, such as the polyhedral method, are typically much more cumbersome to implement as they involve a lot of complex machinery [9, 11, 16, 42, 59]. We prefer to take advantage of the high level of our intermediate representation to make optimization easier rather than first lower the adverbs to loops and then perform a loop tiling pass on them.

An additional benefit of tiling using tiled adverbs rather than tiling lowered loops is that it allows us to perform adverb fusion on the tiled versions of the code. Adverb fusion is a well known optimization technique to combine data parallel operators that share iteration spaces [80]. Our tiling algorithm (as described in the next chapater) can generate new adverbs for statements in a function that gets tiled, and adverb fusion can help optimize these generated adverbs to be more efficient. Again, we could perform such optimization on lowered loop versions of the adverbs. However, this would require a complicated dependence analysis pass. Our adverb fusion optimization required only

150 lines of Python to implement.

We envision more uses of tiled adverbs in the future, some of which we describe in Chapter 9.

# Chapter 4

# Algorithmic Tiling of Adverbs

In this chapter, we present our algorithm for automatically translating a Parakeet function with adverbs into a version with tiled adverbs. The basic idea should be intuitive: we wrap adverbs in tiled versions of themselves (say, a Map in a TiledMap), and in this way use the tiled adverb to break up the original adverb's computation into locality-friendly pieces. The original adverbs become the transform functions of the tiled adverbs, so that the original computation is performed on each tile.

Our approach will be first to present a subset of our algorithm that is able to tile only simple nestings of functions that contain a single adverb statement each. After this exposition, we expand our algorithm by adding machinery for tiling more general code.

## 4.1 Tiling Simple Adverb Nestings

In this section, we describe our algorithm for tiling simple nestings of functions with adverbs, a formal description of which is given in Figures 4.5 and 4.6. In presenting our algorithm for tiling simple adverbs nestings, we will use a program that sums the rows

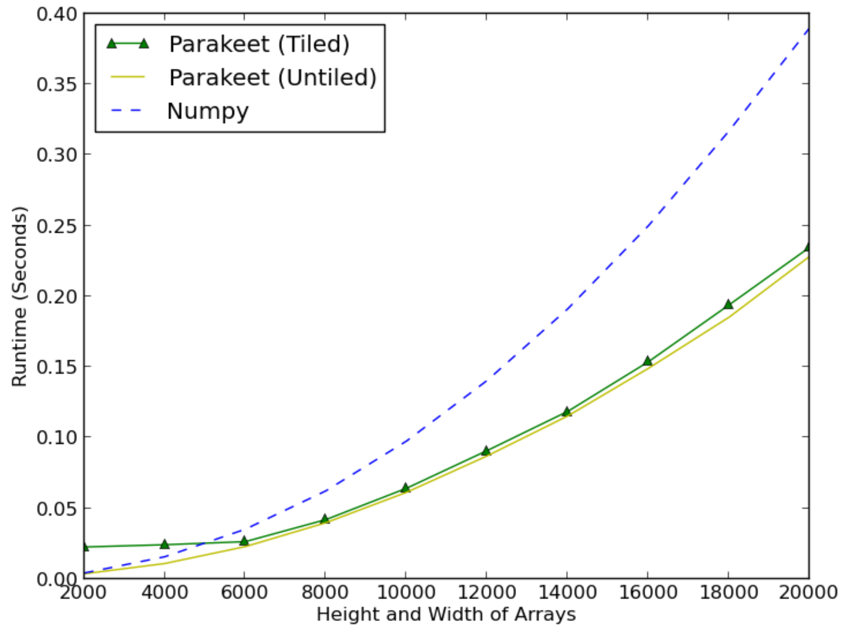
of a 2D matrix as a simple running example. Parakeet code for this is given in Figure 4.3. The iteration pattern through the elements of the array for this version is shown on the left side of Figure 4.2 – the Reduce iterates over each row in its entirety before the Map moves on to the next row.

Such a program has the potential to benefit from cache tiling due to the cache line effect discussed in Chapter 3. For example, imagine that the data of variable `Xs` is laid out such that adjacent elements of columns (rather than rows) are adjacent in memory. In this case, whenever an element of the array is read (and thus brought into cache), some number of neighboring elements in its column will also be brought into cache as they'll be in the same cache line. If the rows are at least roughly the size of the cache however, these neighboring elements will have been evicted by the time it is their turn to be read. Thus what could have been a cheap cache hit if the program were tiled instead becomes a costly trip to RAM.

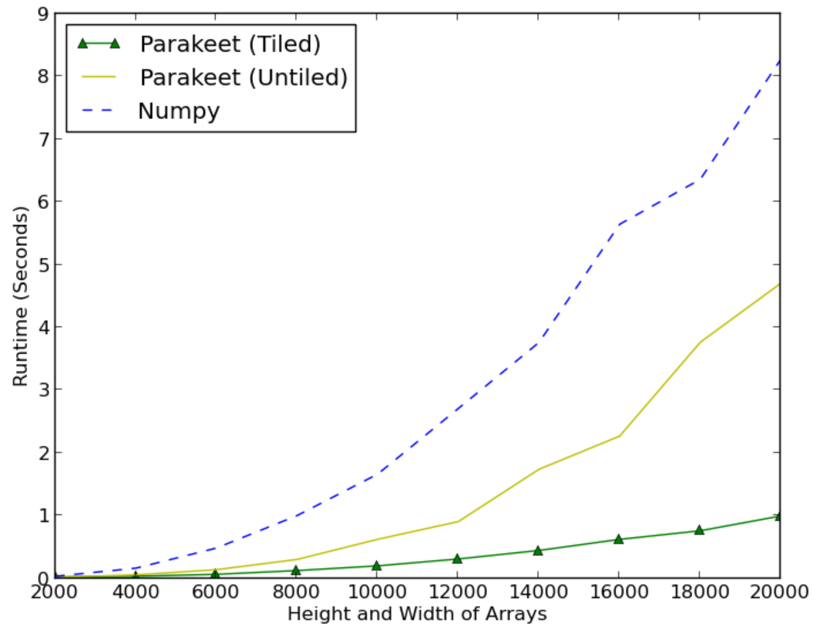
In Figure 4.1, we show the performance of the tiled Parakeet version of this program as compared to a NumPy version. The top graph shows the runtimes of the row sums program on data laid out in row-major order. This is the cache-friendly layout, as adjacent elements in the cache lines are also adjacent elements read by the program's Reduce. The runtimes for all array sizes shown are below 0.5 seconds. Tiling doesn't do much to alter the runtime of this version, and Parakeet outperforms NumPy by a factor of 2 on 20000x20000 arrays.

On the bottom of Figure 4.1, we show the performance of the program on data laid out in column-major order. Here, the tiled Parakeet version performs significantly better than the untiled version, with tiling speeding the computation up by 4.7X for a 20000x20000 array. The tiled version is 8.3X faster than the NumPy version.

Our strategy is to break up each adverb's computation into cache-friendly pieces by




(a) Runtimes with row-major layout




(b) Runtimes with column-major layout

Figure 4.1: Rows Sums Performance with Two Different Memory Layouts

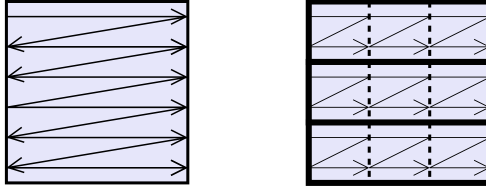


Figure 4.2: Iteration Order of Untiled (left) and Tiled (right) 2D Row Sums

```
def add2(a, b):
  return a+b

def sum_row(row):
  return parakeet.reduce(add2, row, init=0, axes=[0])

def sum_rows(Xs):
  return parakeet.map(sum_row, Xs, axes=[0])
```

Figure 4.3: Sum Each Row Of A 2D Array

adding tiled versions of them. To break up the Map, we wrap the entire computation in a TiledMap that groups the rows of the array into tiles. These row tiles are represented on the right side of Figure 4.2 by the bold boxes around groups of rows. To tile the Reduce, we then break these row tiles further via the use of a TiledReduce that divides the rows in each row tile into a sequence of partial rows. The divisions added by the TiledReduce are represented in the figure by the dashed lines. Thus the final tiles are 2D pieces of the original 2D array `Xs`. The iteration order through each of the final tiles is represented by the arrows on the right side of Figure 4.2. Note that this procedure lets us break every dimension of the input into pieces of bounded size. Thus regardless of the size of any of the dimensions of the input array, with properly chosen tile sizes we can ensure that the amount of data in the smallest tiles fits into whatever size cache is being targeted for optimization. Another way of viewing this is that the working set of the innermost computation fits entirely in cache, and thus its memory accesses to (1) elements in the same cache lines; and (2) elements that are accessed multiple times are

```
float -> float
def identity(x): return x

float -> float -> float
def add2(a, b):
  return a+b

array1<float> -> float
def sum_row(row):
  return reduce(identity, row, init=0, combine=add2, axes=[0])

array2<float> -> array1<float>
def sum_rows(XsRowTile):
  return map(sum_row, XsRowTile, axes=[0])

array1<float> -> array1<float> -> array1<float>
def tiledadd2(accumulatorTile, reduceTile):
  return map(add2, accumulatorTile, reduceTile, axes=[0,0])

array2<float> -> array1<float>
def tiledsum_row(XsTile):
  return tiledreduce(sum_rows, XsTile, init=0,
                     combine=tiledadd2, axes=[1])

array2<float> -> array1<float>
def tiledsum_rows(Xs):
  return tiledmap(tiledsum_row, Xs, axes=[0])
```

Figure 4.4: Type-Annotated Pseudocode for Tiled Sum Each Row of a 2D Array

much more efficient.

Pseudocode for the tiled version of this code is given below in Figure 4.4. Remember that tiled adverbs aren't exposed to users, and thus one need not (and indeed can not) program in this way directly. In addition, recall that in Parakeet's internal representation of Reduces and Scans, these adverbs have additional slots for the transform and emit functions. These default to the `identity` function, as shown in the figure.

$\alpha$ ::= Axes along which an operator slices each of its arguments

$\Delta$ ::= Maps variables to the list of axes remaining of the original variables of which they are a piece

$\epsilon$ ::= Maps variables to their number of expansions

$\epsilon[x]$ ::= 0 if $x \notin \epsilon$

$[\![f = \lambda x_1, \ldots, x_n.\text{body}]\!] \longmapsto f' = \lambda x_1, \ldots, x_n.\text{body}'$

where
- $\Delta[x_i] = \langle 0, 1, \ldots, \text{rank}(x_i) - 1 \rangle$
- $\epsilon = \{\}$
- $\text{body}' = \langle\rangle$
- for each statement $s \in$ body:
  - if $s$ contains an adverb $a$:
    - $s' = s$ with $a$ replaced by $[\![a]\!]^{\Delta,\epsilon,f}$
    - $body' = body' +\!\!+ \langle s' \rangle$
  - else:
    - $body' = body' +\!\!+ \langle s \rangle$

$[\![block]\!]^{\Delta,\epsilon,g} \longmapsto \langle s' \rangle$

where
- if $block$ contains a single statement $s$:
  - if $s$ contains an adverb $a$:
    - $s' = s$ with $a$ replaced by $[\![a]\!]^{\Delta,\epsilon,g}$
  - else:
    - $s' = s$
- else:
  - Abort tiling transformation and return original function $g$

Figure 4.5: Simplified Tiling Transformation Algorithm, Part 1

*Adverb Transformations*

$$\text{BuildTree}(l, x_1, \ldots, x_n, block) \quad ::= \quad \mathsf{Map}_{\langle 0,\ldots,0\rangle}(args, \lambda args.\ \mathsf{Map}_{\langle 0,\ldots,0\rangle}(args, \lambda args.\ \ldots\ \mathsf{Map}_{\langle 0,\ldots,0\rangle}(args, \lambda args.block) \ldots))$$

where
- $args = x_1, \ldots, x_n$
- number of $\mathsf{Map}$s = $l$

$$[\![\mathsf{Map}_\alpha(v_1, \ldots, v_n, f)]\!]^{\Delta,\epsilon,g} \longmapsto \mathsf{TiledMap}_{\alpha'}(v_1, \ldots, v_n, f^\uparrow)$$

where
- $x_1, \ldots, x_n = \text{args}(f)$
- $\epsilon'[x_i] = \epsilon[v_i] + 1$
- $\alpha'_i = \Delta[v_i][\alpha_i]$
- $\Delta'[x_i] = \Delta[v_i]$ with element $\Delta[v_i][\alpha_i]$ removed
- If $f$ contains adverbs, $f^\uparrow = \lambda x_1, \ldots, x_n.[\![\text{body}(f)]\!]^{\Delta',\epsilon',g}$
- Otherwise, $f^\uparrow = \text{g}$

$$[\![\mathsf{Reduce}_\alpha(v_1, \ldots, v_n, f, \oplus)]\!]^{\Delta,\epsilon,g} \longmapsto \mathsf{TiledReduce}_{\alpha'}(v_1, \ldots, v_n, g, \oplus^\uparrow)$$

where
- $\alpha'_i = \Delta[v_i][\alpha_i]$
- $l = \text{max}_i(\epsilon[v_i])$
- $\oplus^\uparrow = \text{BuildTree}(l, \text{args}(\oplus), \text{body}(\oplus))$

$$[\![\mathsf{Scan}_\alpha(v_1, \ldots, v_n, f, \oplus)]\!]^{\Delta,\epsilon,g} \longmapsto \mathsf{TiledScan}_{\alpha'}(v_1, \ldots, v_n, g, \oplus^\uparrow)$$

where
- $\alpha'_i = \Delta[v_i][\alpha_i]$
- $l = \text{max}_i(\epsilon[v_i])$
- $\oplus^\uparrow = \text{BuildTree}(l, \text{args}(\oplus), \text{body}(\oplus))$

Figure 4.6: Simplified Tiling Transformation Algorithm, Part 2

### 4.1.1 Algorithm Walkthrough using 2D Row Sums as an Example

In this section, we break down the formal algorithm description by describing all of its parts as applied to tiling the program in Figure 4.3. The purpose of this section is largely to provide clarity to the formal description given in the figures. A more informal discussion that includes justifications for the algorithm's design choices follows in Section 4.1.2. Our tiling algorithm is called with a top-level function as its input. Throughout this exposition, we refer to this initial function as the *outermost function*, and we refer to its arguments as the *outermost function arguments* or *outermost arguments*. In the current example, the outermost function is the `sum_rows` function in Figure 4.3.

- $\alpha$. We use the variable $\alpha$ to compactly represent the axes parameter of both regular and tiled adverbs. In our example in Figure 4.3, both the Map and the Reduce have `axes = [0]`, and so $\alpha = \langle 0 \rangle$ for both of them. Thus the Map iterates across the rows, and the Reduce iterates across the elements of each row.

- $\Delta$. The $\Delta$ variable stores a mapping from program variables to the dimensions remaining of the outermost arguments of which they are a part. For example, the `row` variable in the `sum_row` function is a 1D row that is a part of the outermost argument `Xs`. This 1D row has had the 0th dimension of `Xs` removed by the iteration of the Map over that dimension in `sum_rows`. Thus $\Delta$[`row`] is equal to the list $\langle 1 \rangle$, as only the 1st dimension of `Xs` remains out of the original dimensions $\langle 0, 1 \rangle$ (we use the notation $\langle \rangle$ to represent a list). We use this information to translate the axes parameters of the original adverbs into those used for the tiled adverbs. We expand on this when we discuss the steps of the algorithm that tile adverbs.

- $\epsilon$. The $\epsilon$ variable is another mapping, this time from program variables to the

number of times they were an argument to a tiled adverb. This includes the number of times that the variables of which they are a sub-part were arguments to tiled adverbs. We call this the number of times that the variable was *expanded*. To continue the example, we assign $\epsilon$[`row`] = 1 for the `row` variable in the `sum_row` function, as `Xs` was expanded once by the Map in the `sum_rows` function and `row` is a part (viz. a row) of `Xs`. This information is used in generating tiled combine functions for TiledReduces and TiledScans. We set $\epsilon[x] = 0$ when $x \notin \epsilon$.

- $[\![f = \lambda x_1, \ldots, x_n.\text{body}]\!]$. The $[\![]\!]$ operator represents a call to some step of the tiling algorithm on some piece of syntax. In the case of this step in the algorithm, the call is on an entire outermost function $f$, and in our example, this outermost function is the `sum_rows` function in Figure 4.3. The compiler calls the tiling algorithm via this entrypoint, after type-specializing `sum_rows` for its specific arguments (i.e a single 2D array of floats). The algorithm sets up the values of $\Delta[x_i]$ for each input argument $x_i$ to be the list $\langle 0, 1, \ldots,\ \text{rank}(x_i)\rangle$, where by rank we mean the variable's dimensionality. The reason is that each outermost argument has all of its dimensions remaining. In the example, $\Delta$[`Xs`] is set to $\langle 0, 1\rangle$, as `Xs` is 2D. We also set the $\epsilon$ map to be empty, as no variables have yet been tiled. Then, for each statement in `sum_rows`'s body, we tile any adverbs in the statement using these values of $\Delta$ and $\epsilon$. The only statement in `sum_rows` is the one that returns the call to the Map, and this Map is recursively tiled. If there are no adverbs in a statement, we keep the statement unchanged. We also pass along the whole outermost function `sum_rows` to subsequent parts of the algorithm, as we eventually need to splice `sum_rows` in as the innermost computation that consumes the smallest tiles. The tiled version of `sum_rows` with the TiledMap replacing the Map (shown in Figure 4.4 as the `tiledsum_rows` function) is then returned as the

final output of the tiling algorithm.

- $[\![block]\!]^{\Delta,\epsilon,g}$. This step processes an entire block of statements, and is called as a sub-step of the tiling procedures for the adverbs to perform the recursive tiling of adverbs in transform functions. It takes as input parameters the block, as well as the current values of $\Delta$ and $\epsilon$ and the outermost function $g$. Since our simplified algorithm can only handle tiling transform functions that contain a single statement it checks that the block meets this constraint; if not then the tiling procedure is aborted and the original untiled function $g$ is returned. If there is a single statement with an adverb, the procedure continues by calling a function to tile the adverb. If the single statement has no adverb, the single statement is returned unchanged. In our example, this step is called on the body of the `sum_row` function in order to allow its Reduce to be recursively tiled.

- BuildTree($l, x_1, \ldots, x_n, block$). BuildTree is a helper function that is used in generating a combine functions for TiledReduces and TiledScans. It wraps a block of code in $l$ Maps, each of which takes as input the same variables $x_1, \ldots, x_n$ and iterates over axis 0 on each such input variable. The purpose of BuildTree is to peel off extra dimensions added to partial results by tiling, and it is explained in more detail in the discussion of tiling Reduces.

- $[\![\mathsf{Map}_\alpha(v_1, \ldots, v_n, f)]\!]^{\Delta,\epsilon,g}$. This step tiles a Map expression, returning a TiledMap to be added to the tiled version of the program we generate. In the example, it takes the `map(sum_row, Xs, axes=[0])` expression in the `sum_rows` function, and returns the `tiledmap(tiledsum_row, Xs, axes=[0])` expression generated for the `tiledsum_rows` function shown in Figure 4.4. Let us walk through the steps of this function one by one as they pertain to tiling this Map.

- $x_1, \ldots, x_n$ = args($f$). This is simply a convenience definition - each $x_i$ is an input argument to the function $f$.
- $\epsilon'[x_i] = \epsilon[v_i] + 1$. Here, we add the arguments to the Map's transform function `sum_row` to a new version of the $\epsilon$ map that tracks the number of times each of these variables has been expanded. This new version $\epsilon'$ will be used in recursively tiling `sum_row`. For each argument $x_i$ to `sum_row`, we add 1 to the number of expansions $\epsilon[v_i]$ of the respective argument to the Map. Thus we set $\epsilon'$[`row`] = $\epsilon$[`Xs`] + 1 = 1. This way, the number of expansions for a variable is maintained as the total number of expansions of the outermost argument of which it is a part.
- $\alpha'_i = \Delta[v_i][\alpha_i]$. Here we generate the axes parameter $\alpha'$ for the TiledMap. To do this, for each index $i$ we pull out the list of remaining dimensions in the $\Delta$ map for the input variable $v_i$ to the Map. We then index into this list with the axis parameter $\alpha_i$ of the original Map. In our example, we look up $\Delta$[`Xs`] and find the list $\langle 0, 1\rangle$. Next, we index into this list at the value of the axes parameter for this variable, i.e. $\alpha_0 = 0$. Thus we set $\alpha'_0 =$ $\Delta$[`Xs`][$\alpha_0$] = 0. This translates the local axis variable of the Map to a global axis parameter suitable for the TiledMap. Since this is the only input variable, $\alpha' = \langle 0\rangle$, and we use $\alpha'$ as the axes parameter for the generated TiledMap.
- $\Delta'[x_i] = \Delta[v_i]$ with element $\Delta[v_i][\alpha_i]$ removed. Next we generate an updated $\Delta'$ that removes the dimensions from each variable $v_i$ that are sliced away by the Map. In our example, the Map in `sum_rows` slices away dimension 0 of the `Xs` variable, passing the rows one at a time into its transform function `sum_row`. We thus need to generate an updated $\Delta'$ that reflects these

missing dimensions for use in tiling `sum_row`. We thus remove $\Delta[\texttt{Xs}][\alpha_0]$ = 0 from $\Delta[\texttt{Xs}]$ = $\langle 0, 1\rangle$, and are left with $\langle 1\rangle$. We thus set $\Delta'[\texttt{row}]$ = $\langle 1\rangle$. As expected, we see that this value for `row` stores the information that `row` consists only of the 1st dimension of the variable of which it is a part, viz. `Xs`.

  - Finally, we check whether the transform function `sum_row` contains any adverbs. Since it does, we recursively tile its body via a call to the block tiling function, using the updated parameters $\Delta'$ and $\epsilon'$ and still passing the outermost function `sum_rows` through. The result of this tiling call on `sum_row`'s body becomes the body of the transform function of the generated TiledMap. If the transform function $f$ didn't contain any adverbs, we would be finished tiling the tree. In this case we would use the original function $g$ as the transform function of the TiledMap, and the algorithm would terminate.

- $[\![\mathsf{Reduce}_{\alpha}(v_1, \ldots, v_n, f, \oplus)]\!]^{\Delta,\epsilon,g}$. This step tiles a Reduce expression, returning a TiledReduce. In our example, it takes as input the

```
reduce(identity, row, init=0, combine=add2, axes=[0])
```

expression in the `sum_row` function and returns the tiled version

```
tiledreduce(sum_rows, XsTile, init=0, combine=tiledadd2, axes=[1])
```

used in the `tiledsum_row` function shown in Figure 4.4. Note that the tiling algorithm always terminates upon reaching a Reduce, splicing in the outermost function $g$ (`sum_rows` in our example) as the transform function of the TiledReduce.

The reason for this is explained in the next section. Again, let's walk through each of its substeps.

- $\alpha'_i = \Delta[v_i][\alpha_i]$. This step generates the axes parameter $\alpha'$ of the TiledReduce in exactly the same way as we did for the TiledMap as described in the previous step. In our example, we set $\alpha'_0 = \Delta[\texttt{row}][\alpha_0] = \langle 1 \rangle[0] = 1$. Thus, we set the axes parameter for the tiled version of `row` in the TiledReduce to be 1, not 0 as in the original reduce. The reason is that we need to translate the local axis of the original Reduce (which only operated on 1D rows at a time) into a global axis which picks the right dimension out of an entire 2D tile.
- $l$ = $\max_i(\epsilon[v_i])$. Here, we find the maximum number of expansions for any of the input variables to the Reduce. In this case, there is only one input variable `row`, and it was exanded once. Thus $l = 1$. We use this value of $l$ in the next step.
- $\oplus^{\uparrow}$ = BuildTree($l$, args($\oplus$), body($\oplus$)). In this step, we generate a tiled combine function $\oplus^{\uparrow}$ for the TiledReduce. In Figure 4.4, $\oplus^{\uparrow}$ = `tiled_add2`. We can't use the original combine function `add2` directly, because the partial results of the TiledReduce will be results of calling a transform function on entire tiles, and thus they will have larger rank than the arguments to `add2`. Specifically, they will have rank exactly $l$ larger. This is where we use the expansion information. Each expansion adds a rank to the tiles that form the inputs to the TiledReduce. Thus the partial results of the TiledReduce will be 1 rank larger than those of the regular Reduce, and we need to alter the combine function so as to accept arguments of this rank. We do this by

calling a helper function called BuildTree that wraps the combine function in $l$ Maps. Each such Map removes a rank from the partial results of the TiledReduce. In this case, $l = 1$, and so we wrap `add2` in 1 extra Map. These Maps each take as input all arguments to the combine function, thus peeling off a rank from each. Afterward, we can call the original combine function `add2` directly on each pair of elements of the tiled partial results, and notice that by design this nesting of Maps iterates over every such pair in tiles. Thus we call the combine function on exactly the same elements as the original program did.

- $[\![\mathsf{Scan}_\alpha(v_1, \ldots, v_n, f, \oplus)]\!]^{\Delta,\epsilon,g}$. This step tiles a Scan, returning a TiledScan. Its substeps are otherwise identical to those for tiling a Reduce, so we omit any further discussion of them.

Thus we have covered all of the parts of the algorithm that were used to generate the code given in Figure 4.4. The TiledMap groups rows into tiles, and the TiledReduce slices these tiles further into 2D groups of partial rows, following the iteration order shown on the right side of Figure 4.2. The original `sum_rows` function is called on each of these 2D tiles, which generates a 1D partial result consisting of the partial sum for each of the partial rows in the 2D tile. A tiled combine function `tiledadd2` is used to combine these 1D partial results in the proper way, such that the TiledReduce's output is a 1D tile of single sums, one for each of the rows in the TiledMap's tile. Thus, we see that the tiled computation produces the same results as the original computation. We are then free to set the sizes of the tiles to whatever is best for the cache for which we are optimizing.

### 4.1.2 Tiling Algorithm Design Breakdown

We now expand on this walkthrough of the algorithm with a high level discussion of the features of the algorithm in order to provide the intuition behind the design choices we made in developing our tiling algorithm.

#### 4.1.2.1 Outer Nesting of Tiled Adverbs

First, notice that the final nesting of adverbs in the tiled code is

TiledMap(TiledReduce(Map(Reduce(...)))).

Another option, rather than generating an outer nesting of tiled adverbs that matches the nesting of regular adverbs, would have been to wrap each adverb directly in a tiled version of itself. In this case, the final nesting of adverbs would have been

TiledMap(Map(TiledReduce(Reduce(...)))).

Much of the complexity of our algorithm – for example, the translation of the axes necessary for the tiled adverbs – revolves around producing a nesting that collects all of the tiled adverbs on the outside.

To see why we nonetheless opt for this, imagine the iteration order for the latter nesting. The TiledMap would group some number of rows of `Xs` into a tile. Next, the Map would iterate over each row one at a time. Thus the TiledReduce would execute on only one row at a time in a tiled fashion. This means that the final tiles would have an arbitrary size along the second dimension of `Xs`, but would be fixed at having only a size of 1 along the first dimension. This in turn means that the TiledMap isn't really serving

much function in slicing up the innermost computation's working set in this version of tiling – in the end, only the innermost tile size will matter much for locality. Hence, we opt for the approach discussed in the previous paragraph.

#### 4.1.2.2 Axes of Tiled Operators

The TiledReduce's iteration axis for the `XsMapTile` is 1, not 0 as in the original Reduce. To understand why, look again at the iteration order shown in Figure 4.2 and the type-annotated version of the tiled code in Figure 4.4. Notice that each `XsMapTile` is a two dimensional group of rows, whereas the original reduce iterated over single 1D rows. A key difference between tiled adverbs and regular adverbs is that tiled adverbs always operate on pieces of the outermost arguments (`Xs` being the only such outer argument in this case) that are the same rank as the entire outermost arguments themselves. In contrast, regular adverbs operate on pieces of the outermost arguments with progressively decreasing rank, as each regular adverb "slices away" one entire dimension by pulling out each single element of the variable at a time and passing these elements to the adverb's transform function. Thus the axes of iteration for regular adverbs have a local view on the outermost arguments – they see only the dimensions of the outermost arguments that remain after any previous adverbs have sliced some dimensions away. Therefore, even though the Reduce in the original code iterates over the rows that form axis 1 of the outermost 2D input argument `Xs`, its axis of iteration over its single 1D row is 0, as axis 0 is the only possible axis of iteration for a 1D array. What we need to do in order to retain the original semantics of the untiled program is to have the tiled versions of adverbs iterate over the same elements as their untiled counterparts. In this way, the tiled adverbs perform the desired function, viz. splitting up the iterations of their untiled analogues. For this, we need to translate the local views on axes that regular adverbs

have into global views that reflect the dimensions as situated in the entire tiles/outermost arguments.

In order perform this translation, we need to keep around some state for each variable we encounter as we walk the function tree. Specifically, for each array variable that is a piece of an outermost function argument, we store a list of the dimensions that remain of the original outermost argument. In the formal description of the algorithm, this mapping from variables to their remaining dimensions is denoted by $\Delta$. In addition, we initialize this list of remaining dimensions to be $\langle 0, ..., n-1 \rangle$ for $n$-dimensional outermost arguments. For example, when we tile the `sum_rows` function from Figure 4.3, we mark the `Xs` variable as having dimensions $\langle 0, 1 \rangle$ remaining, since no dimensions have yet been sliced off of it by adverbs. Next, when we traverse into the Map's transform function `sum_row`, we mark its argument `row` to have all of these dimensions *except the one sliced away* by the Map. Since the Map iterates over axis 0, we mark `row` as having only axes $\langle 1 \rangle$ remaining of the original input argument `Xs`.

Using these lists of axes, we can translate the local axes of the adverb back to the global axes of the entire outermost function argument. Since tiles of an outermost argument always have the same rank as that argument, this translation accomplishes what we're really after, which is translating the local axis into an axis that fits the tile that the tiled adverb we're generating takes as an input. We do this by looking up the value in the list of remaining dimensions at the index equal to the axis value for the regular adverb. This tells us which of the original dimension's elements are stored at the respective locally-viewed axis. To finish the example, we see that the original Reduce iterates over axis 0 on variable `row`. We look up which value is at index 0 in the list of remaining dimensions $\langle 1 \rangle$, and find the value 1. Thus we know that the Reduce iterates over dimension 1 of the original input argument, and we set the axis value of the

TiledReduce to be 1 for its tile of this variable.

#### 4.1.2.3 Tiled Combine Functions

Recall that the purpose of a combine function is to provide a way to combine two partial results of a reduction into a meaningful result, such that we are free to split the reduction into sub-reductions of any length. For a TiledReduce or a TiledScan, we need the combine function to combine two partial results of executing the transform function on *tiles*.

We create this tiled combine function by altering the original combine function to operate on tiles rather than arguments of the original ranks. To do this, we introduce the notion of variable *expansions*. Whenever we tile an adverb, we create a version of the adverb whose transform function takes arguments of one higher rank than the transform function of the original adverb. This is again because regular adverbs "slice away" one entire dimension from a variable. Tiled adverbs, on the other hand, don't slice away any dimensions, rather creating pieces of their inputs with equal rank to the inputs themselves. We thus say that these tiled counterparts of the transform function's input arguments have been *expanded* by the tiling. The number of expansions a variable has undergone is equal to the number of extra ranks that the tiled analogue of that variable has compared with the variable itself. We keep track of this number of expansions for each variable in another mapping, which in the formal algorithm is represented by $\epsilon$.

Let's walk through how we use expansions to generate the TiledReduce's combine function in the `tiledsum_row` function in Figure 4.4. We use the original combine function to perform the heart of the combine operation between two of the TiledReduce's partial results, but we need somehow to break apart these partial results into values that are of the rank that the original combine function can accept. We do this simply by

wrapping the combine function in a number of Maps equal to the greatest number of expansions that any input variable to the Reduce has undergone, which in turn is equal to the number of extra ranks that the TiledReduce's partial results have compared to those of the original Reduce. (In the formal presentation of the algorithm, the helper function BuildTree performs this wrapping in Maps.) Each input variable of the combine function is passed as an argument to each of these Maps. This procedure gives us exactly what we want, as each of these Maps "peels off" a rank from a variable. Once all of the extra ranks have been so peeled off, we are left with elements that have the ranks that the original combine function expects, and we can call that function to combine the elements. The nesting of Maps also by design traverses every element of the TiledReduce's expanded partial results, ensuring that we call the combine function on everything that the original code did.

To get back to our example, let's walk through maintaining the expansions map $\epsilon$ and how we use it to generate the combine function for the TiledReduce. When we tiled the Map in the `sum_rows` function, we marked the argument `row` to the `sum_row` function as having been expanded once. When we tile the Reduce, we thus have to wrap its combine function in a single Map in order to peel off the extra rank added by this expansion. Notice that the original Reduce's partial results are scalar `float`s, and its combine function `add2` thus combines two scalar `floats` into one via summation. The TiledReduce's partial results, on the other hand, are partial sums for some number of different rows, this number equal to the number of rows in each tile of the TiledMap. Thus the TiledReduce's combine function needs to add each of these partial sums together across two tiles. We see that a single Map does the job, mapping across each pair of partial sums, adding them together with the original combine function `add2`.

```
def add2_tuple(acc, (val, idx)):
  return (acc + val, idx)

def test_max((best_max, best_idx), (val, idx)):
  if val > best_max:
    return (val, idx)
  else:
    return (best_max, best_idx)

def max(xrow):
  return parakeet.reduce(test_max, xrow, init=(-1,-1))

def sum_maxes(Xs):
  return parakeet.reduce(add2_tuple, Xs, init=(0,0))[0]
```

Figure 4.7: Two Nested Reductions

### 4.1.3 Termination Upon Reaching a Reduce or a Scan

The algorithm as described in the formal description only recursively tiles the transform functions of Maps, and terminates the tiling transformation upon encountering a Reduce or a Scan. The reason for this is that, in general, it isn't safe for an outer computation to operate on a partial result of a reduction. Consider for example the code given in Figure 4.7. This somewhat contrived example takes as input a 2D array `Xs`, finds the maximum value of every row, and sums these maximum values. If we were to tile this code, we would generate two nested TiledReduces. The outer TiledReduce would group the rows of `Xs` into tiles, while the inner TiledReduce would take a split a group rows of `Xs` into a group of partial rows and it would call the original function `sum_maxes` on that tile. This `sum_maxes` would find the maximum value in each of these partial rows in the tile, and then it would compute the sum of these maxima. This one scalar value would form the partial result for the entire tile, and then the TiledReduce would move on to the next tile.

This is clearly incorrect. The final sum for the entire inner TiledReduce includes a sum of maxima for each of its tiles (a group of partial rows). However, these maxima

needn't have been the final maxima for the entire rows, which are the only things that should have been included in the final sum. In addition, we don't want to add up all of these partial maxima, thus contributing a value to the sum for each partial row, but rather want a single global maximum to be added to the sum for each entire row. A key reason why tiling these two nested Reduces doesn't work is that the partial maxima don't have any meaning outside the context of the inner Reduce. Using their partial values elsewhere in the computation in the hope that they will compose isn't valid. For this reason, we terminate the tiling algorithm upon reaching a Reduce or a Scan. Thus, the partial results of a TiledReduce or a TiledScan are consumed by that TiledReduce or TiledScan directly before passing up the chain to any surrounding TiledMaps.

## 4.2 The Full Algorithm: Tiling Nested Scalar Statements

Our simplified version of the tiling algorithm can handle tiling only simple nestings of adverbs where each adverb's transform function, if it contains a statement with an adverb, has only that statement as its entire body. If this constraint doesn't hold, we abort the tiling operation and return the original untiled version of the code. Our full algorithm includes extensions that allow it to tile functions that contain multiple statements, so long as only one of them contains an adverb. We discuss the changes necessary to support this in this section. A formalized description of our entire algorithm is given in Figures 4.8, 4.9, and 4.10.

Let us consider an example. In Figure 4.11, we present a modified version of the program that sums rows of a 2D array. In this version, everything is identical to that in the previous section, except that the `sum_row` function slices off the first element of each row before summing it. This may seem contrived, but similar things are often done in

| | | |
|---|---|---|
| $\alpha$ | ::= | Axes along which an adverb slices each of its arguments |
| $\sigma$ | ::= | Ordered sequence of visited adverbs |
| $\Delta$ | ::= | Maps variables to the list of axes remaining of the original variables of which they are a piece |
| $\epsilon$ | ::= | Maps variables to the nesting depths at which they were tiled |
| $\epsilon[x]$ | ::= | $\langle\rangle$ if $x \notin \epsilon$ or if $x$ is a constant |
| FV$(e)$ | ::= | The set of free variables in expression $e$ |

BuildTree$(\sigma, \epsilon, x_1, \ldots, x_n,$ $block)$ ::=

$$\sigma_0(\text{vars}_0, \lambda\text{vars}_0.\ \sigma_1(\text{vars}_1, \lambda\text{vars}_1.\ \ldots\ \sigma_d(\text{vars}_d, \lambda\text{vars}_d.block) \ldots))$$

where

$d = |\sigma|$

$\text{vars}_i = \{x_j | i \in \epsilon[x_j]\}$

$[\![f = \lambda x_1, \ldots, x_n.\text{body}]\!] \longmapsto f' = \lambda x_1, \ldots, x_n.\text{body}'$

where

$\sigma = \langle\rangle$

$\Delta[x_i] = \langle 0, 1, \ldots, \text{rank}(x_i) - 1\rangle$

$\epsilon = \{\}$

$\text{body}' = [\![\text{body}]\!]^{\sigma,\Delta,\epsilon}$

Figure 4.8: Full Tiling Transformation Algorithm, Part 1

*Statement and Expression Transformations*

$[\![\textbf{return } e]\!]^{\sigma,\Delta,\epsilon}$ $\longmapsto$ $\epsilon$, **return** $[\![e]\!]^{\sigma,\Delta,\epsilon}$

$[\![x\textbf{=}e]\!]^{\sigma,\Delta,\epsilon}$ $\longmapsto$ $\epsilon'$, x = $e'$
where
$\epsilon'[x] = \bigcup\{\epsilon[y] \mid y \in \text{FV}(e)\}$
if $e$ contains an adverb:
$e' = [\![e]\!]^{\sigma,\Delta,\epsilon}$
else:
$\sigma' = \langle \mathsf{Map}_0, \mathsf{Map}_0, \ldots \rangle$ such that $|\sigma'| = |\epsilon'[x]|$
$e' = \text{BuildTree}(\sigma', \epsilon', \text{FV}(e), \langle \textbf{return } e \rangle)$

$[\![block]\!]^{\sigma,\Delta,\epsilon}$ $\longmapsto$ $block'$
where
$block' = \langle\rangle$
$\epsilon' = \epsilon$
for $s \in block$ :
$\epsilon', s' = \langle [\![s]\!]^{\sigma,\Delta,\epsilon'} \rangle$
$block' = block' +\!\!+ s'$

$[\![e]\!]^{\sigma,\Delta,\epsilon}$ $\longmapsto$ $e$ (if $e$ is not an adverb)

Figure 4.9: Full Tiling Transformation Algorithm, Part 2

*Adverb Transformations*

$[\![\mathsf{Map}_\alpha(v_1,\ldots,v_n,f)]\!]^{\sigma,\Delta,\epsilon} \longmapsto \mathsf{TiledMap}_{\alpha'}(v_1,\ldots,v_n,f^\uparrow)$
where
- $x_1,\ldots,x_n = \text{args}(f)$
- $d = |\sigma|$
- $\epsilon'[x_i] = \epsilon[v_i] +\!\!+ \langle d \rangle$
- $\sigma' = \sigma +\!\!+ \langle \mathsf{Map}_\alpha \rangle$
- $\alpha'_i = \Delta[v_i][\alpha_i]$
- $\Delta'[x_i] = \Delta[v_i]$ with element $\Delta[v_i][\alpha_i]$ removed
- If $f$ contains adverbs, $\text{body}^\uparrow = [\![\text{body}(f)]\!]^{\sigma',\Delta',\epsilon'}$
- Otherwise, $\text{body}^\uparrow = \text{BuildTree}(\sigma',\epsilon',x_1,\ldots,x_n,\text{body}(f))$
- $f^\uparrow = \lambda x_1,\ldots,x_n.\text{body}^\uparrow$

$[\![\mathsf{Reduce}_\alpha(v_1,\ldots,v_n,f,\oplus)]\!]^{\sigma,\Delta,\epsilon} \longmapsto \mathsf{TiledReduce}_{\alpha'}(v_1,\ldots,v_n,f^\uparrow,\oplus^\uparrow)$
where
- $x_1,\ldots,x_n = \text{args}(f)$
- $d = |\sigma|$
- $\epsilon'[x_i] = \epsilon[v_i] +\!\!+ \langle d \rangle$
- $\sigma' = \sigma +\!\!+ \langle \mathsf{Reduce}_\alpha \rangle$
- $\alpha' = \Delta[v_i][\alpha_i]$
- $\text{body}^\uparrow = \text{BuildTree}(\sigma',\epsilon',v_1,\ldots,v_n,\text{body}(f))$
- $f^\uparrow = \lambda x_1,\ldots,x_n.\text{body}^\uparrow$
- $\sigma'' = \langle \mathsf{Map}_0, \mathsf{Map}_0, \ldots \rangle$ such that $|\sigma''| = d$
- $c_1,\ldots,c_k = \text{args}(\oplus)$
- $l = \max_i(\epsilon[v_i])$
- $\epsilon''[c_j] = \langle 0,1,\ldots,l-1 \rangle$
- $\oplus^\uparrow = \text{BuildTree}(\sigma'',\epsilon'',\text{args}(\oplus),\text{body}(\oplus))$

$[\![\mathsf{Scan}_\alpha(v_1,\ldots,v_n,f,\oplus)]\!]^{\sigma,\Delta,\epsilon} \longmapsto$ Same as $\mathsf{Reduce}$ but with $\mathsf{Scan}$ and $\mathsf{TiledScan}$

Figure 4.10: Full Tiling Transformation Algorithm, Part 3

```python
def add2(a, b):
  return a+b

def sum_row(row):
  rowPart = row[1:]
  return parakeet.reduce(add2, rowPart, init=0, axes=[0])

def sum_rows(Xs):
  return parakeet.map(sum_row, Xs, axes=[0])
```

Figure 4.11: Indexed Sum of Rows of a 2D Array

real code. For example, consider a particle simulation in which we need to store each particle's mass as well as its $(x, y, z)$ coordinates. In such a scenario we might need to slice off the mass component in order to do spatial calculations.

Our simple algorithm would not be able to tile this code, as it has no rule for what to do with the non-adverb statements in the transform functions of adverbs. One option – adding the indexing statement `row[1:]` both to the transform function of the TiledReduce we generate while also keeping it in the original function that we splice into the innermost tiled adverb – would slice off too much. First we would slice off the first column of the tile of the TiledMap, which would amount to slicing off the first element of each row in the tile (so far so good). However, once we got to the spliced-in original computation, the `sum_row` function would slice its partial row again. Thus, too many elements will have been slice out of the rows, and we would get incorrect results.

To solve this issue and allow us to tile such code, we do two things. First, we disallow control flow in functions being tiled; if control flow is found, the tiling transformation is undone and the code is left untiled. Previous work has shown that ontrol flow can be handled in these cases by predication [11]; we could do something similar, but as discussed in Chapter 8, we leave this for future work.

Next, we execute each non-adverb statement only once, rather than both in the tiled

```
float -> float
def identity(x): return x

float -> float -> float
def add2(a, b):
  return a+b

array1<float> -> float
def sum_row(row):
  return reduce(identity, row, init=0, combine=add2, axes=[0])

array2<float> -> array1<float>
def sum_rows(XsRowTile):
  return map(sum_row, XsRowTile, axes=[0])

array1<float> -> array1<float> -> array1<float>
def tiledadd2(accumulatorTile, reduceTile):
  return map(add2, accumulatorTile, reduceTile, axes=[0,0])

array1<float> -> array1<float>
def index1D(array1D):
  return array1D[1:]

array2<float> -> array1<float>
def tiledsum_row(XsTile):
  tiledRowPart = map(index1D, XsTile, axes=[0])
  return tiledreduce(sum_rows, tiledRowPart, init=0,
                     combine=tiledadd2, axes=[1])

array2<float> -> array1<float>
def tiledsum_rows(Xs):
  return tiledmap(tiledsum_row, Xs, axes=[0])
```

Figure 4.12: Type-Annotated Pseudocode for Tiled Indexed Sum Each Row of a 2D Array

and original versions of the program's functions. Specifically, we keep these statements in the tiled versions of the functions, and remove them from a copy we generate of the original program that includes the original adverbs. In the case of indexing such as in Figure 4.11, this is the only legal possibility, as if the indexing were delayed until the inner regular adverbs executed, portions of the original variables could be present and iterated over that weren't visible the original program. Note that we only remove

statements *that have an adverb in the same scope*. The transform function of the innermost adverb, which can contain arbitrary non-adverb code including control flow, is left unchanged.

Note, of course, that these two added restrictions also apply to the simplified algorithm, since there we require that every tiled function have a single statement with an adverb.

A type-annotated pseudocode version of this program is given in Figure 4.12. Notice how the indexing is only done in the tiled function `tiledsum_row`, and not in the function `sum_row` that contains the regular reduce. In addition, this indexing has been wrapped in a Map, as we need to index into a *tile*, not just a single row. In order to facilitate these changes, we alter our simplified algorithm in various ways. Let's cover them one by one.

- $\sigma$. First, we add another state variable $\sigma$ that contains the list of adverbs tiled thus far in the algorithm. In our example program, $\sigma$ will be $\langle \text{Map}_0 \rangle$ after tiling the Map, and will be $\langle \text{Map}_0, \text{Reduce}_0 \rangle$ after tiling the Reduce. We use $\sigma$ to generate the copy of `sum_row` that has the offending indexing statement removed. We do this by altering the BuildTree function to take a list of adverbs rather than always wrapping a function in Maps - we discuss this new BuildTree function in detail later on. We wrap the innermost transform function `identity` in this series of adverbs in order to generate versions of the functions with only the single adverb statements as desired.

- $\epsilon$. In the full version of the algorithm, we alter the use of $\epsilon$. Now, rather than simply storing the *number* of expansions for each variable, we store the list of nesting depths (starting at index 0) at which each variables was expanded. For

example, the `row` argument variable to the `sum_row` function now gets the list $\langle 0 \rangle$ when we tile the Map in `sum_rows`, as this is the first adverb in the nesting. We use these nesting depths to determine which variables need to be arguments to which adverbs when reconstructing the tree of regular adverb functions that have the non-adverb statements removed.

- FV($e$). This helper function returns a list of all the free variables in the expression $e$. This is used in building the tiled versions of the non-adverb statements placed inside the tiled transform functions, as described next.

- BuildTree($\sigma, \epsilon, x_1, \ldots, x_n, block$). The heart of the changes for supporting nested non-adverb statements is the new version of the BuildTree function. BuildTree now takes a list of adverbs $\sigma$ and a list of nesting depths $\epsilon$, whose meanings are described above. In this way, we can use BuildTree for three separate purposes:

  1. Generating tiled combine functions, as in the simplified algorithm.
  2. Generating versions of the non-adverb statements in transform functions that can operate on tiles for use in the tiled versions of the transform functions.
  3. Generating versions of the transform functions in the original program that have the non-adverb statements removed.

  The first use is the same as in the simplified algorithm.

  The second use is very similar to the first. When we add the non-adverb statements to the tiled versions of the transform functions, these statements need to be altered so as to take as arguments entire tiles rather than arguments of the original rank. Thus we need to peel off the extra ranks added by tiling, just as in the combine function case. In our example, we first call FV(`row[1:]`) and get back the list

$\langle$`row`$\rangle$. This becomes the list of variables that we use as inputs to the Map s with which we wrap the indexing expression. We then call BuildTree to add a single Map to peel off the one rank added from the expansion of the `row` variable.

The third use of the new BuildTree function is to generate the versions of the transform functions that have the regular adverb statements, but all non-adverb statements removed. BuildTree uses the $\epsilon$ parameter to determine which variables from $x_1, \ldots, x_n$ need to be arguments to which of the adverbs in $\sigma$. Specifically, BuildTree only passes the variables that were expanded at nesting depth $i$ to as arguments to the adverb $\sigma_i$. In our example, we pass the arguments $\sigma = \langle$Map$_0$, Reduce$_0\rangle$, $\epsilon = \langle 0, 1 \rangle$, $x_1 = x$, and *body* $= \langle$`return x`$\rangle$ as the arguments to BuildTree to create the `sum_rows` and `sum_row` functions. BuildTree then wraps this function body (the body of the `identity` function, the innermost transform function) in a Map and a Reduce. The only argument to `identity` is the variable `x`, and it was expanded at both depths 0 and 1. Thus it is passed as an argument to each of these adverbs, and the desired function nesting is created.

- $[\![$**return** $e]\!]^{\sigma,\Delta,\epsilon}$. This step of the algorithm simply makes explicit the tiling of the expression $e$. If $e$ is an adverb, it is recursively tiled; otherwise, it is simply returned.

- $[\![x = e]\!]^{\sigma,\Delta,\epsilon}$. This new step of the algorithm captures adding non-adverb statements to tiled transform functions. For example, when we encounter the statement `rowPart = row[1:]` in the program, we need to create a version of this statement than can handle a tile of rows at a time. To do this, we collect all of the depths at which any of the free variables of $e$ were expanded. In this case, `row` is the only free variable, and it was expanded at depth 0, so this list is $\langle 0 \rangle$. We create a new

version of the $\epsilon$ map $\epsilon'$, and set $\epsilon'$[`rowPart`] = $\langle 0 \rangle$. This way, `rowPart` inherits the expansions of the expression `row[1:]`, and these get propagated in the right way later on when `rowPart` is passed to the Reduce.

Next, we call BuildTree on with $\sigma' = \langle \mathsf{Map}_0 \rangle$, $\epsilon'$, the list of free variables $\langle$`row`$\rangle$, and a block with a single statment $\langle$**return** `row[1:]`$\rangle$. BuildTree wraps the block in the single Map, with the result being the `map(index1D, XsTile, axes=0)` expression in the `tiledsum_row` function.

- $[\![\mathsf{Map}_\alpha(v_1, \ldots, v_n, f)]\!]^{\sigma,\Delta,\epsilon}$. The tiling procedures for adverbs are largely the same. The only differences are that we update $\sigma$ and the new $\epsilon$ that stores lists of expansion depths rather than numbers of expansions. We also use the BuildTree function as described above to generate the innermost transform functions and we still use it to generate the tiled combine functions.

This method of dealing with scalar statements has the potential to be wasteful in that it generates array temporaries when originally there were none. It has the benefit of making the algorithm simpler as roughly the same unpacking logic can be applied to all cases of extra ranks due to tiling. If many of these expanded scalar statements exist in a function, adverb fusion is able to combine them all into a single tree of Maps, mitigating some of this cost.

# Chapter 5

# Selecting Tile Sizes

## 5.1 Register Tile Sizes

Register tile sizes need to be selected at compile time, as register tiling requires loop unrolling and scalar replacement to be performed after the tiling step itself. Thus, we cannot leave register tile sizes unspecified until runtime. Some previous work has explored using iterative compilation techniques to try different register tile sizes at runtime [50], and other work has used offline searching to exhaustively try out all different possible register tile sizes [81]. Still further work has developed models of the register tiling problem that cast it as an optimization problem [63].

We use a simple heuristic to set the register tile sizes once at compile time. We always set the innermost register tile size to be 1, and we evenly divide the remaining registers on the processor among the other adverbs' tiles. To do this, we assume that each iteration of an adverb consumes 2 registers, one for the array element read and one for the array element written. This heuristic is very simplistic, but works well on the benchmarks and machines we tried. The key thing with register tiling is not to use too

many registers, and this heuristic does well in ensuring that.

## 5.2 Cache Tile Sizes

There are two methods used to set cache tile sizes in the literature: statically via models, and empirically via autotuning. We discuss both methods in this chapter. We opt for an online autotuning approach in Parakeet.

### 5.2.1 Statically Estimating Cache Tile Sizes

A large body of work exists on using models for general tiling or tiling for specific domains that allow for good static setting of tile size parameters [14, 27, 36, 85, 86]. The general-purpose models typically involve a simplified model of cache replacement for predicting cache performance. Special-purpose models exist for common, important problems such as Matrix Multiplication [85, 86] that incorporate specific aspects of the problem domain. While these models tend to perform reasonably well, empirical autotuning of tile sizes is still considered the best way to get optimal performance on a wide range of problems [6].

### 5.2.2 Online Autotuning Cache Tile Sizes

Autotuning tile sizes can be done either offline via an exhaustive search, or online while the program is running. Offline autotuning is the standard approach for highly-tuned problem-specific libraries such as FFTW [38] or ATLAS [81]. However, the offline approach isn't suitable for Parakeet, as we want to keep the dynamic flavor of Python as much as possible. A big part of what makes a dynamic language like Python so easy and convenient to use is the instant feedback and lack of an explicit compilation

step. In addition, for programs that aren't run multiple times, an offline search would only add overhead. For these reasons, we are unwilling to add a time-consuming exhaustive search to our pipeline and opt for an efficient online search that tries different tile sizes during an actual run of the program.

In our setup, we use the DL and ML algorithms from [71] as low and high initial guesses for tile sizes. The DL algorithm is designed to provide a pessimistic estimate of cache behavior and thus minimum values for tile sizes, while the ML algorithm provides optimistic maximum tile size estimates. These algorithms take as inputs the cache and cache line sizes, as well as information about how the array accesses are nested. The output of each of these algorithms isn't a specific set of tile sizes, but rather an equation for a surface of tile sizes that is meant to serve as the boundary of admissible settings. The focus of the work in [71] is to provide an algorithmic way to bound the search space for an offline exhaustive search.

We use these algorithms to provide initial tile settings to seed our online tile size search rather than to provide bounds for an exhaustive search. Thus we need to pick some specific points on these surfaces to use. We opt for cubic tile sizes for our initial guesses.

A common algorithm for searching across tile sizes used in the literature is the Parallel Rank Ordering algorithm (PRO), similar in flavor to the Nelder-Mead method [74, 78]. In this method, a simplex of tile sizes is maintained with 2 points in the tile space for each tile parameter. In each step of the algorithm, a reflection through each point of the simplex is evaluated by executing the program with the tile sizes corresponding to the point. As many such points are evaluated as possible in parallel. Then the reflection is either accepted or rejected, and further shrinking or expanding of the simplex is potentially done.

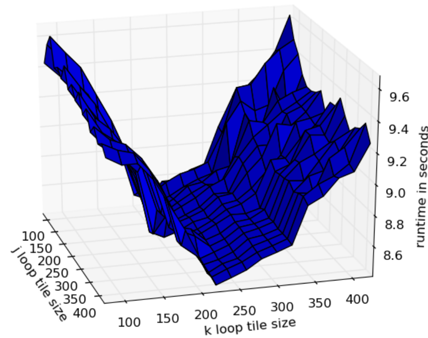



Figure 5.1: Matrix Multiply Performance vs Tile Size

In our experiments, we weren't able to get this method to work well enough as it had too high overhead compared to the benefit of reaching better tile sizes. In our setup, we aren't trying to find the best possible tile size, as we're tuning tiling online for user code rather than tuning it offline for a reusable library. Thus, we need our algorithm to converge quickly to a fairly good tile size, and then stop searching and exploit the good tile size for the rest of the run. Each set of tile sizes tested is relatively expensive, as when the sizes tested are bad they slow down the whole run. Further, for many of the benchmarks we tested, the performance relative to tile size settings involved a region of tile size settings that had good performance, surrounded by settings where performance was bad (shown for matrix multiplication in Figure 5.1). The performance variation within the good region wasn't high enough to justify further tuning once it was reached.

Thus we opted for a different search algorithm that required fewer samples to reach the good region of the tile size space. In each time step, we take the current best performing point, initialized to the average of the DL and ML estimates discussed in the previous section. For each tile size, we take a Gaussian sample with standard deviation equal to one fourth the difference between these estimates. A sample for each tile size

forms a candidate tuple of tile sizes, and a number of these tuples is evaluated in parallel on different cores of the machine. If any perform better than the previous best point, they become the new best.

We set tried various settings for the following parameters of the search:

- Percentage of the runtime (in terms of number of iterations of the outermost adverb) to spend searching before switching to using the current best-found tile sizes.
- How long to wait between taking performance readings and trying out new candidate tile sizes.
- How many times that sampling doesn't result in a new best tile setting before the search terminates.

In the benchmarks we tried (matrix multiplication, K-Means clustering, and Gaussian Blur), we found that our search algorithm has very little variation in performance for different settings of these parameters. If we spend less than 20% of the runtime searching, then the overhead of searching tends to be greater than the benefit the search provides as the search doesn't have enough time to find good tile sizes. Otherwise, the time spent searching doesn't impact performance much, so we set it heuristically to 50%. We also found that if we set the number of tries to find new best tile settings needs to be at least 2 for good settings to be found consistently. We evaluate the search in somewhat more detail in the following chapter.

# Chapter 6

# Experimental Evaluation

In this chapter, we evaluate our cache and register tiling optimizations on Matrix Multiplication, K-Means Clustering, and Gaussian Blurring. We have implemented a number of different programs in Parakeet, but use these as a representative sample.

We evaluate our benchmarks on a system with an Intel i7 960 3.2GHz processor and 16GB of RAM. This processor has 4 hyperthreaded cores, each with a 32KB L1 data cache with 64 byte cache lines. Parakeet reads these hardware characteristics from the `/proc` and `/sys/devices` filesystems and uses them to configure both the cache tiling and register tiling optimizations.

Unless otherwise noted, we turn cache and register tiling on and off together. In addition, unless otherwise noted all performance numbers don't include compilation times. We assess compilation time in Section 6.4.

```
def dot(x, y):
  return sum(x*y)

def mm(Xs, Ys):
  return parakeet.allpairs(dot, Xs, Ys, axes=[0,1])
```

Figure 6.1: Parakeet Matrix Multiply

## 6.1 Matrix Multiplication

In this section, we present results for a 2D matrix multiplication benchmark, the actual Parakeet code for which is in Figure 6.1. Recall that AllPairs is Parakeet syntactic sugar for two nested Maps each of which iterates over one of the two arguments to the AllPairs.

### 6.1.1 Performance On Arrays with Row-Major Layouts

In Figure 6.2, we present performance results when running this matrix multiplication program on arrays that are both laid out in row-major order. In this case, the right matrix suffers from the same problem that the row sums program did from the previous chapter: without tiling, the locality within cache lines is not exploited.

We compare Parakeet's performance against NumPy's. NumPy can use any implementation of BLAS, a standard linear algebra interface, installed on a user's computer to perform matrix multiplications. We configure NumPy to use two different BLAS versions – one written as naive C consisting of three nested `for` loops; and a stock ATLAS [81] implementation that comes with Ubuntu Linux. ATLAS is a heavily hand-tuned matrix multiplication library that uses a number of optimizations, including both cache and register tiling. For optimal performance, one needs to perform a lengthy autotuning step to configure ATLAS for the particular target machine. We use a stock version rather and autotuned one as we simply want to provide a rough reference point

for calibrating the meaning of our results, and we want to simulate an environment of a typical user who installs programs “off the shelf”. In each of these graphs, we used our online autotuner to find good cache tile sizes.

### 6.1.2 Performance On Arrays with Cache-Friendly Layouts

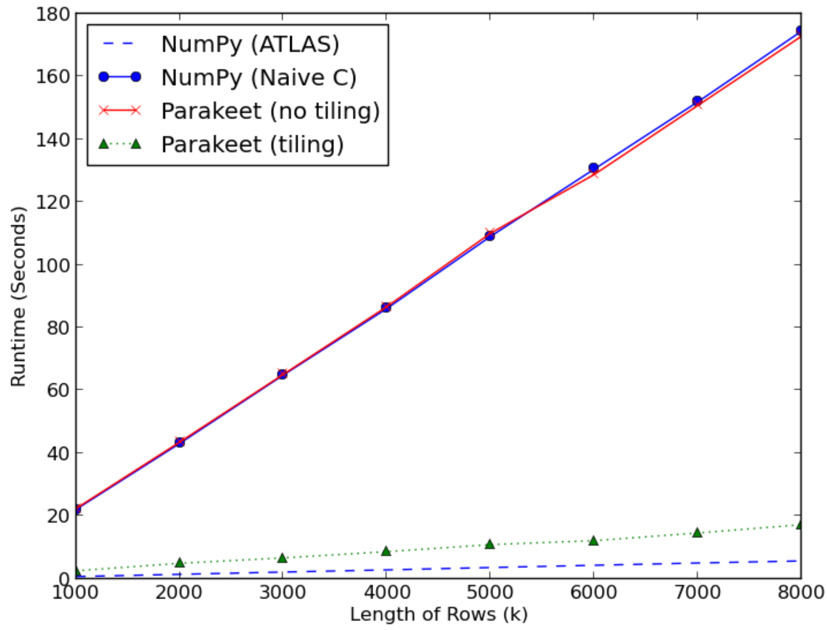
In Figure 6.3, we present performance results when running this matrix multiplication program on arrays that are laid out perfectly for cache: the left array is laid out in row-major order, while the right array is laid out in column-major order. Thus in this case, tiling has the least opportunity for improving performance.

On the top of the figure, the number of rows in the left-hand matrix vary along the X axis. The length of rows and the number of rows in the right-hand matrix are held fixed 3000. In this figure, we see that our tiling optimization improves performance between 26.3% and 30.8% over not tiling. On the bottom the figure, we fix the number of rows in both matrices to be 3000 and vary the length of the rows. Here, the tiling speedup varies between 21.1% and 24.8%. In both cases, Parakeet is around 2 times slower than our ATLAS version, which recall has been heavily hand-optimized.

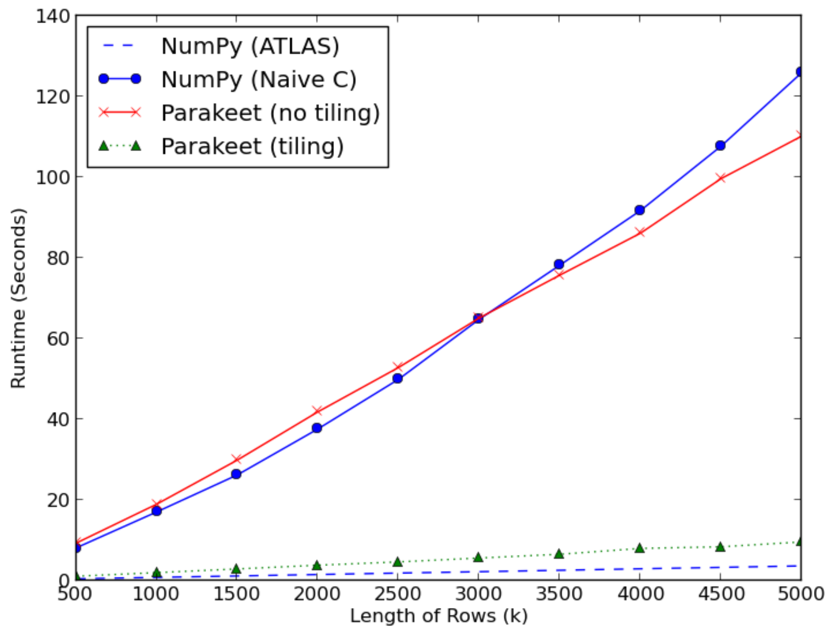
### 6.1.3 Cache and Register Tiling

In this section, we break down the Parakeet performance on matrix multiply for the two different memory layouts by separating out the performance of cache tiling from that of register tiling.

In Table 6.1, we present the performance for the cache-friendly memory layout. Here, we see that while cache and register tiling together result in a performance gain over no tiling, cache tiling alone actually leads to a significant slowdown. We not certain

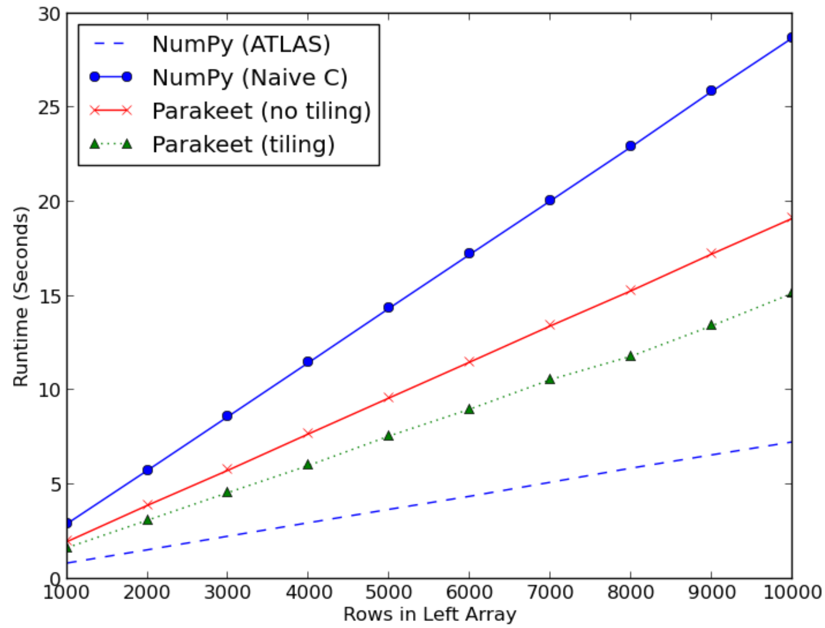



(a) Runtimes with varying rows in left-hand matrix

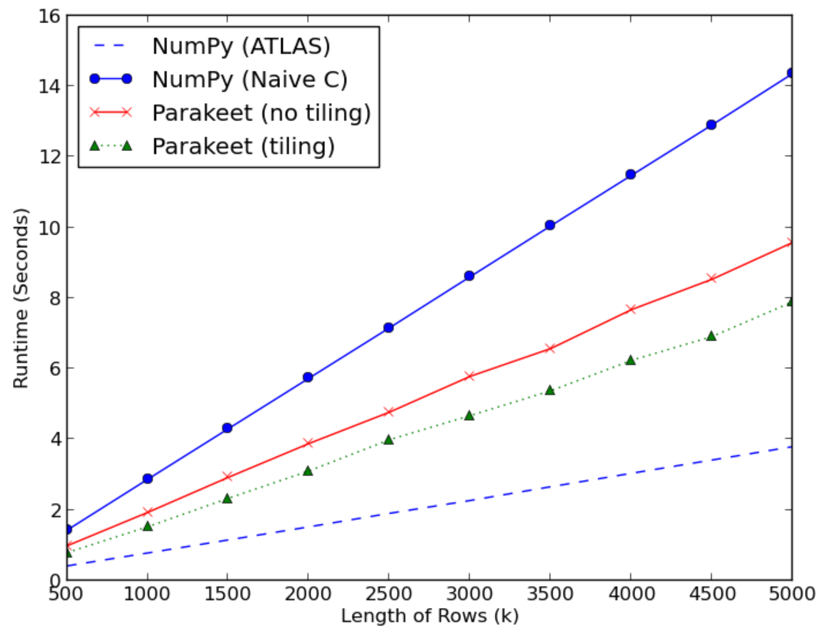



(b) Runtimes with varying lengths of rows and columns

Figure 6.2: Matrix Multiply Runtimes with Row-Major Layouts




(a) Runtimes with varying rows in left-hand matrix




(b) Runtimes with varying lengths of rows and columns

Figure 6.3: Matrix Multiply Runtimes with Cache-Friendly Layouts

| Rows in Left Array | No Tiling | Cache Tiling | Cache+Reg Tiling |
|---|---|---|---|
| 1000 | 1.98s | 2.54s | 1.65s |
| 2000 | 3.91s | 4.68s | 3.11s |
| 3000 | 5.77s | 6.76s | 4.58s |
| 4000 | 7.70s | 9.36s | 6.02s |
| 5000 | 9.61s | 11.39s | 7.59s |
| 6000 | 11.52s | 13.46s | 9.01s |
| 7000 | 13.44s | 16.19s | 10.58s |
| 8000 | 15.30s | 17.78s | 11.82s |

Table 6.1: Matrix Multiply Cache-Friendly Layout Cache and Register Tiling Performance

| Rows in Left Array | No Tiling | Cache Tiling | Cache+Reg Tiling |
|---|---|---|---|
| 1000 | 22.40s | 2.80s | 2.57s |
| 2000 | 43.56s | 5.69s | 4.95s |
| 3000 | 65.10s | 7.90s | 6.68s |
| 4000 | 86.73s | 10.68s | 8.66s |
| 5000 | 110.10s | 13.22s | 10.91s |
| 6000 | 128.81s | 15.71s | 12.16s |
| 7000 | 150.81s | 18.41s | 14.61s |
| 8000 | 172.84s | 20.55s | 17.26s |

Table 6.2: Matrix Multiply Row-Major Layout Cache and Register Tiling Performance

what causes this – perhaps added loop overhead – but with the data laid out in this way, we wouldn't expect much gain from cache tiling.

In Table 6.2, we present a similar breakdown for the row-major layout. In this case, we see that cache tiling provides a very large – between 8.0 and 8.4X – speedup over no tiling. This is exactly as we would expect, as the data is laid out in such a way that there is far more cache pressure. Register tiling adds an additional 30-200% speedup over cache tiling alone.

| Array Size | Parakeet (with search) | Parakeet (cached tile sizes) | DL sizes | ML sizes | Average of DL&ML sizes |
|---|---|---|---|---|---|
| 3000x3000 | 4.58s | 4.45s | 5.03s | 4.55s | 4.61s |
| 10000x3000 | 15.16s | 15.11s | 16.89s | 15.33s | 15.51s |
| 10000x500 | 2.57s | 2.57s | 2.99s | 2.67s | 2.72s |

Table 6.3: Matrix Multiply Cache-Friendly Layout Autotuner Performance

### 6.1.4 Autotuning Performance

In Figure 6.3, we break down the performance of the autotuner by showing Parakeet times both with the autotuning search as well as Parakeet times using the fixed tile sizes found in a previous search. We compare these with using the fixed tile sizes for the DL and ML algorithms from [71], as well as the performance when using the average of the DL and ML estimates as the fixed tile sizes.

By comparing the difference between the runtime with the search and with cached tile sizes, we see that the overhead of performing the search is very small. The time to switch between different tile sizes during a search is close to 0, and so any overhead is almost entirely due to the penalty from running worse tile sizes than those that are eventually found. We also see that, while the autotuner leads to the best runtimes especially on larger data sizes, the performance boost it adds over the ML estimates in particular isn't very large. On average, the autotuner increases performance around 2.3% for all benchmarks and data sizes we tried. While this is only a modest performance boost, it is consistent, and if there are any other programs for which the tile estimates perform badly, the autotuner should be able to increase performance even more by finding better tiles.

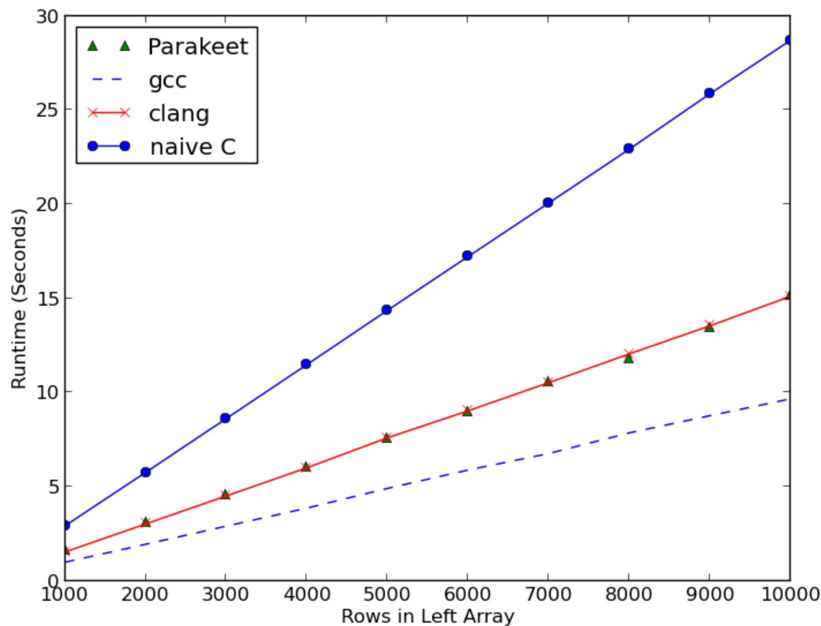



Figure 6.4: Matrix Multiply Performance Compared To C

### 6.1.5 Performance vs. Other Compilers

To provide a test of Parakeet's performance relative to other compilers, we compare Parakeet's performance to a hand-optimized C version with manual cache and register blocking compiled with both gcc and clang (the LLVM C compiler), including all relevent optimization flags. We also include a naive `for` loop C version with only the `-O3` flag for reference. All of these versions were launched on 8 threads on the Parakeet runtime's backend. In practice of course, a programmer would have to manually parallelize the C versions, while the Parakeet version is automatically parallelized. The results are shown in Figure 6.4.

First, notice that Parakeet's performance roughly matches that of clang, while both are roughly 1.5X slower than gcc. We take this as evidence that we are approaching a performance wall due to the underlying performance of LLVM. We discovered that

| Number of Data Points | NumPy | Parakeet (No Tiling) | Parakeet (Tiling) |
|---|---|---|---|
| 10000 | 48.01s | 12.91s | 12.46s |
| 12500 | 59.95s | 16.05s | 15.51s |
| 15000 | 71.83s | 19.16s | 18.51s |
| 17500 | 83.78s | 22.28s | 21.40s |
| 20000 | 97.61s | 25.53s | 24.45s |
| 22500 | 107.62s | 28.54s | 27.35s |
| 25000 | 119.56s | 31.66s | 30.41s |
| 27500 | 131.57s | 34.82s | 33.37s |
| 30000 | 143.32s | 37.89s | 36.42s |

Table 6.4: K-Means Performance with k = 1000, 500 features, and 10 iterations

roughly half of gcc's relative performance gain over clang is due to gcc having a better vectorizer.

## 6.2 K-Means Clustering

We present results for K-Means Clustering in Table 6.4. Here we see that Parakeet is dramatically faster than NumPy, with almost a 4X performance improvement. The performance benefit of tiling on this benchmark is only 4% on average however. This is due to Parakeet's spending a lower percentage of its computation in operations with much data reuse, and the length of its rows (equal to k, or the number of centroids) being smaller than in much of the matrix multiply benchmark. A lower k value means that the amount of data accessed between repeated accessed to the same data points is lower, reducing the cache pressure.

## 6.3 Gaussian Blur

In this section, we present performance results for a Gaussian Blur kernel, a standard computer vision algorithm for blurring images [70]. This benchmark sweeps across every pixel of an image, and returning a new pixel value that is a Gaussian-weighted combination of every pixel in some $nxn$ neighborhood of the original pixel. This results in a blurred-looking image. Our Parakeet code for an 11x11 Gaussian Blur of a 2D image is given in Figure 6.5.

```
def gaussian_kernel(size):
  x, y = np.mgrid[-size:size+1, -size:size+1]
  g = np.exp((-pow(x,2) + pow(y,2)) / float(size))
  return g / g.sum()

s = 5
gaussian = gaussian_kernel(s)

def gaussian_conv(img, i, j):
  window = img[i-s:i+s+1, j-s:j+s+1, :]
  red = 0.0
  green = 0.0
  blue = 0.0
  for it in range(0,2*s+1,1):
    for jt in range(0,2*s+1,1):
      red = red + window[it,jt,0] * gaussian[it,jt]
      green = green + window[it,jt,1] * gaussian[it,jt]
      blue = blue + window[it,jt,2] * gaussian[it,jt]
  return [red, green, blue]

def blur(img):
  def conv_closure(i, j):
    return gaussian_conv(img, i, j)

  iidxs = np.arange(s, len(img)-s)
  jidxs = np.arange(s, len(img[0])-s)
  return parakeet.allpairs(conv_closure, iidxs, jidxs)
```

Figure 6.5: Gaussian Blur Parakeet Code

The main loop of this program is implemented by the call to AllPairs in the `blur` function. We create two arrays of indices, one for each of the x and y dimensions of the

image, and the AllPairs calculates a new pixel value for each pair of indices. Note that this code doesn't handle the boundary region for positions within 5 pixels of the edge of the image. Boundary regions are typically handled by some separate cleanup code which we omit here.

We were curious whether cache tiling would improve the performance of this kernel. There is the potential for the cache tiling to allow for better exploitation of reused data within the windows, as both horizontally and vertically adjacent $n$x$n$ windows have many pixels of overlap.

The results for the 11x11 kernel are given in Figure 6.6. We don't include the results for the NumPy version, as it is so much slower than Parakeet that it would skew the chart. For example, on a 300x300 pixel image, NumPy takes 272.1s while the untiled version of Parakeet takes 0.84s. The reason for this is that there is no suitable NumPy precompiled primitive to use for this benchmark, and so the majority of the work is done in Python `for` loops which are extremely slow. This illustrates well the motivation behind Parakeet – we want to make all high level code fast, not just code for which someone wrote a hand-tuned library.

We see that for this kernel size, tiling actually hurts performance. Cache tiling alone is on average 4.8% slower than no tiling. In this case, it appears that the added overhead of the nested loops overwhelms any benefit from the cache reuse behavior. In the untiled version, there is plenty of cache benefit already, as the horizontal sweep of the window still involves plenty of data overlap between neighboring 11x11 windows. Interestingly, our online search moves toward final tile sizes that are roughly the size of the entire image as a way to compensate for the fact that tiling is a performance loss.

In addition, we wouldn't expect register tiling to be beneficial in this case, and on average it leads to an additional 2.8% slowdown. Recall that the purpose of register

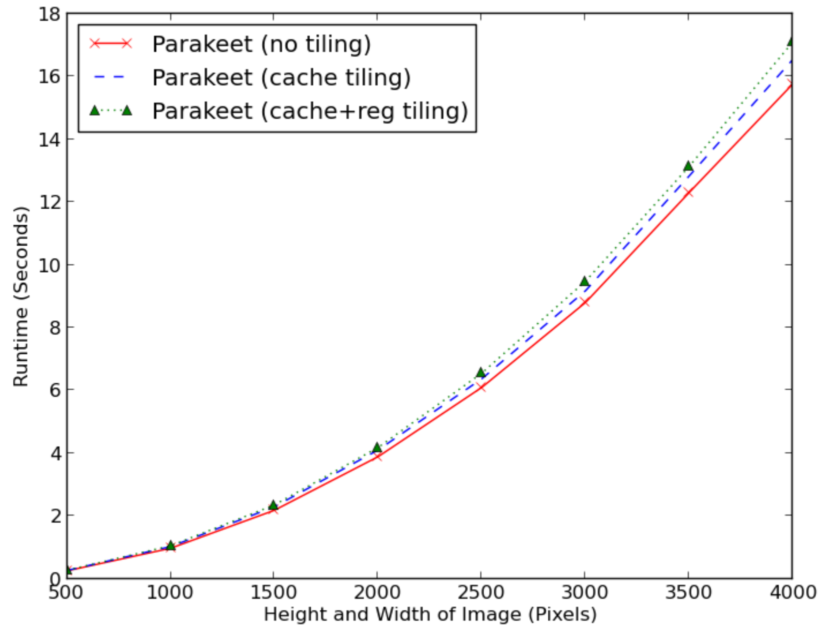



Figure 6.6: Impact of Tiling on 11x11 Gaussian Blur Kernel

tiling is to keep the working set of a small inner loop inside registers for as long as possible. Matrix multiplication is a benchmark where the inner loop is a simple scalar computation (multiplication and addition), and thus the data used by a few iterations of it can fit entirely inside processor register. In the Gaussian blur case, the innermost ested function is the much more complicated `gaussian_conv` function, which contains two nested `for` loops inside it.

## 6.4 Compilation Time

In Figure 6.5, we give the Parakeet compilation times for the benchmarks without tiling, with only cache tiling, and with both cache and register tiling. Cache tiling adds a modest amount of compilation time, while register tiling adds around 1 second for

|  | Matrix Multiply | K-Means | Gaussian Blur |
|---|---|---|---|
| No Tiling | 0.19s | 0.10s | 0.21s |
| Cache Tiling | 0.27s | 0.31s | 0.47s |
| Cache Tiling + Register Tiling | 1.26s | 1.29s | 0.72s |

Table 6.5: Parakeet Compilation Times

each benchmark. We believe that the extra loop unrolling accounts for most of this added time. Our compiler was written entirely in Python and we haven't spent any effort optimizing its compile times, so we are hopeful that these numbers can be brought down in the near future. In addition, we imagine a common use case for Parakeet to be repeated calls to a Parakeet function inside a large numerics computation. In these cases such compile times would only contribute a tiny fraction of overall program runtimes.

# Chapter 7

# Related Work

Our work builds upon a range of existing fields of study including data parallel programming, just-in-time compilation, acceleration of high level array languages, tiling optimizations, analytic performance modeling, and autotuning.

**Data Parallelism.** The first language to feature data parallel abstractions was APL [46], whose central programming constructs involved high-level manipulation of n-dimensional arrays. The eminent parallelizability of the language's core operators inspired early research in vector processors [76] and parallelization [52]. As computers with massively parallel hardware became more common in the 1980s, many languages such as C [43], Fortran [37], and Lisp [73] were retrofitted with data parallel extensions. More recently, data parallel constructs have appeared repeatedly as core primitives for high level languages and libraries which compile to FPGA descriptions [41], GPU programs [20, 75], and even the coordination of distributed computations [88]. The functional skeletons community has a body of work that deals with generalizing the traditional set of data parallel operators to be able to express the core loops of more algorithms via the use of

simple operators [26, 30]. Recently, DryadLINQ has used data parallelism as a useful abstraction for distributed computing [87, 88].

**Just-In-Time Compilation.** Translation of a running program into a more efficient representation has a long history [7] reaching back to the $LC^2$ programming language [54] and early efforts to accelerate APL [2]. The modern age of just-in-time compilation began with the seminal work on the dynamic object-oriented language SELF [22], which pioneered techniques such as *polymorphic inline caching* [45]. Dynamo [8] introduced the concept for trace-based compilation, which was quickly adapted for use on top of the JVM [5]. Trace-based compilation for dynamic languages [39] has recently gained popularity and is the basis for many widely used language implementations.

Parakeet eschews tracing in favor of preserving high-level language constructs and instead peforms run-time type inference when a function is called from Python. This approach falls within the paradigm of *Selective Embedded JIT Specialization* [21].

**High Level Language Acceleration.** Much work has been done on accelerating high level data parallel languages via compilation. MaJIC was an early project that parallelized array computations in Matlab [4].

A large number of projects exist that try to accelerate Python. NumPy is a widely-used toolkit for scientific computing that largely relies on hand-written library functions for accelerating common tasks [34]. NumPyPy is an attempt to reimplement all of NumPy in Python and then let PyPy act as a meta-tracing JIT optimizer [1]. Numba is a project that's just getting off the ground whose main purpose, for the most part, is to unbox numeric values and make looping fast in Python [29]. Blaze is a new project by the creator of NumPy whose goal is to work with more complex data types than arrays

as well as to handle larger-than-memory and streaming datasets [28].

Copperhead takes the direct route to parallelism by forcing you to write your code using data parallel operators which have clear compilation schemes onto multicore and GPU targets [20]. To further simplify the compiler's job, Copperhead forces adverbs' nested functions to be purely functional, while Parakeet supports mutable state.

**Cache and Register Tiling.** There has been much work on automating the process of tiling, including both generating the tiled loops as well as automating tile size selection given a tiled loop nest [51, 82].

It might appear at first glance that determining good tile sizes for a given loop nest might be easy to do analytically. Much work has been done on designing analytic models to estimate good cache tile sizes [23, 27, 40, 51]. In addition, domain-specific models have been developed for problems such as matrix multiplication for estimating both cache and register tile sizes [85, 86]. Wolf et al. developed an analytic model for use in a compiler to determine tiling and loop unrolling settings statically for sequential C and Fortran programs [84].

However, while these models often perform fairly well, the state of the art for achieving the best performance is still offline autotuning to find the best settings for a particular target architecture, even for a problem as well-studied as matrix multiplication [71, 81]. A special problem here is how to generate loops with parameterized tile sizes that can be set at runtime, and much work has been done on that problem for loops [9, 42, 49, 65].

Register tiling is a classic optimization and has been well-studied [19, 55]. This includes work on extracting inner loops from code for register tiling [42, 47, 64, 62], register tiling code generation [68], models for selecting tile sizes [63, 68, 85, 86], and autotuning register tile sizes [50, 81].

**The Polyhedral Model.** The polyhedral model is a well-studied method for modeling the iteration spaces and data dependencies of statements in loop nests. The model represents these by a series of matrices and uses algebraic transformations on them to enable performing various loop optimizations including tiling, skewing, and automatic parallelization [9, 11, 16, 42, 59, 82, 83].

The polyhedral model requires that the indexes into arrays in loops that it tiles all be affine functions of the index variables. This makes it largely more general than our tiling algorithm, which is tailored specifically for the case of tiling uniform multidimensional arrays and the simple iteration patterns of adverbs. However, here are some particular cases where our algorithm can tile code that the polyhedral model can't. Our algorithm can tile adverb nestings where arrays are indexed in arbitrary ways including, for example, with indices that are themselves values read from some other array. The only restriction is that the computation be expressed as a nesting of functions with one adverb each.

Another benefit of our limited-domain approach over the polyhedral model is that code generation in the model is highly nontrivial [10], and it is only recently that techniques for bringing polyhedral optimizations to mainstream compilers have started being developed [16].

Just as in Parakeet, arbitrary control flow in loop nests is not directly representable in the polyhedral model. A solution given in [11] enables control flow in loop nests to be representable in the polyhedral model via the use of predication, which could be used to extend our tiling algorithm in largely the same fashion.

**Autotuning and Iterative Compilation.** In recent years, offline autotuning has emerged

as the accepted best practice for optimizing numerical code [6, 12]. Libraries such as ATLAS for dense linear algebra [81] and FFTW for Fourier transforms [38] deliver the best performance available across a wide range of architectures and platforms for their specific problem domains via an extensive offline search performed at installation time. Other recent work on offline autotuning includes that for stencil computations [31, 48]. While this technique is useful for either basic computational building blocks that are reused extensively (ATLAS and FFTW) or in the hands of expert programmers that are able to hand-program efficient low-level code with autotuning hooks inserted, our system expands the usefulness of autotuning to a broader audience and to rapid prototyping scenarios. Active Harmony is a system for performing online autotuning that exposes a constraint specification language (CSL) for expressing tuning parameters [78]. Tiwari et al. presented a system that combines Active Harmony and Chen's system to autotune loop-based C and Fortran programs offline [77].

Another body of work that uses autotuning for program optimization is that of Iterative Compilation. Many compiler optimizations beyond tiling have parameters that can be tuned for particular programs, but most compilers simply use heuristics for setting these parameters for all programs. Iterative compilation uses autotuning at compile time to try to better tune these optimizations for particular programs. The ADAPT system was an early entrant into this space [79]. Chen et al. developed a system that automatically generates multiple candidate versions of C and Fortran programs by analyzing array references in loops and performing unroll-and-jam, cache tiling, copying, and TLB-oriented optimizations [24]. Knijnenburg et al. presented a system for combining analytic models with offline autotuning to improve parameter settings of cache tiling and unrolling optimizations [50]. Pouchet et al developed systems that use the polyhedral model to generate candidate versions of a program, and then use iterative compilation

to find good loop transformations [58, 59].

# Chapter 8

# Future Work

Our goal with Parakeet is to create a useful system that will enable real programmers to simplify their jobs and to achieve good performance. As such, we plan to continue working on it and to publish an official release in the near future. There are a number of ways we hope to improve Parakeet, and our tiling optimizations in particular.

Our current algorithm disallows control flow in tiled adverb nestings. A solution given in [11] enables control flow in loop nests to be representable in the polyhedral model via the use of predication, which could be used to extend our tiling algorithm in largely the same fashion. We have worked out the details of how this could be done, and it would be interesting to investigate in which cases this would lead to performance gains.

An alternative to expanding all non-adverb statements in tiled functions would be to keep the statements for which it is safe (such as scalar operators) in the inner functions of the tiled computation so as not to generate the temporaries. We have also worked out the details of this, but haven't had time to implement it or explore how much it would benefit performance.

In the future, we hope to use iterative compilation methods to try different compiled versions of programs with different register tile sizes at runtime, or to more methodically explore heuristics for setting register tile sizes at compile time.

While the version of Parakeet discussed in this thesis supports multicore CPUs, we have published a previous version of Parakeet that included a backend for NVIDIA GPUs [66]. In the future, we hope to revive our GPU backend. In order to get good performance on a GPU, it is very important to make good use of fast, software-controlled memories, particularly one called shared memory [56]. Shared memory is used in much the same way as a cache on a CPU, except that the programmer has to manually fill it with data. We believe our tiled adverbs abstraction would make automating this process in our compiler much easier.

Another case where our tiled adverbs might help is in enabling SIMD vectorization. Since a big part of the performance gap between LLVM and gcc is vectorization, SIMD optimizations could lead to large performance gains.

We would like to explore adding an additional adverb to directly support stencil computations. This would allow us to have simple innermost transform functions in stencil such as the Gaussian Blur and increase the opportunities for tiling, while also making expressing stencils with adverbs more natural.

Lastly, we would like to optimize our compilation times. Thus far all of our effort has been devoted to implementing new features.

# Chapter 9

# Conclusion

In summary, we have presented tiled adverbs, natural generalizations of classical data parallel operators that are used for breaking up data parallel programs into locality-friendly pieces. Our tiling transformation automatically generates tiled versions of programs from programs written using regular adverbs in Parakeet, a high level array-oriented DSL embedded in Python. This transform simplifies the job of the compiler writer for data parallel languages by enabling tiling at a much higher level of abstraction than that of loops. We apply this transformation twice, once to enable cache tiling and a second time to enable register tiling. Our system includes an autotuner that, after estimating candidate tile sizes using published algorithms, tunes these while the program runs, resulting in a modest performance boost. We evaluate our optimizations on benchmark programs and show significant performance improvements over untiled code and favorable performance compared to C versions. We thus bring cutting-edge performance to non-expert programmers and to a rapid prototyping environment.

# Bibliography


[1] NumPyPy. `https://bitbucket.org/pypy/numpypy`.

[2] P. S. Abrams. *An APL machine*. PhD thesis, Stanford, CA, USA, 1970.

[3] R. Allen and K. Kennedy. Automatic translation of FORTRAN programs to vector form. *ACM Trans. Program. Lang. Syst.*, 9(4):491–542, Oct. 1987.

[4] G. Almási and D. Padua. MaJIC: Compiling MATLAB for speed and responsiveness. In *PLDI '02: Proceedings of the 2002 ACM SIGPLAN Conference on Programming Languages Design and Implementation (PLDI)*, pages 294–303, 2002.

[5] M. Arnold, S. Fink, D. Grove, M. Hind, and P. Sweeney. Adaptive optimization in the jalapeno jvm. *ACM SIGPLAN Notices*, 35(10):47–65, 2000.

[6] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

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

[8] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. In *ACM SIGPLAN Notices*, volume 35, pages 1–12. ACM, 2000.

[9] M. M. Baskaran, A. Hartono, S. Tavarageri, T. Henretty, J. Ramanujam, and P. Sadayappan. Parameterized tiling revisited. pages 200–209, 2010.

[10] C. Bastoul. Code generation in the polyhedral model is easier than you think. In *PACT '04: Proceedings of the 13th International Conference on Processors, Architectures, and Compilation Techniques*, pages 7–16, 2004.

[11] M.-W. Benabderrahmane, L.-N. Pouchet, A. Cohen, and C. Bastoul. The polyhedral model is more widely applicable than you think. In *Proceedings of the 19th International Conference on Compiler Construction (CC)*, pages 283–303, 2010.

[12] J. Bilmes, K. Asanovic, C.-W. Chin, and J. Demmel. Optimizing matrix multiply using phipac: a portable, high-performance, ANSI C coding methodology. In *Proceedings of the 1997 ACM International Conference on Supercomputing (ICS)*, pages 340–347, 1997.

[13] G. E. Blelloch. Prefix sums and their applications. Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, Nov 1990.

[14] F. Bodin, W. Jalby, D. Windheiser, and C. Eisenbeis. A quantitative algorithm for data locality optimization. In *Code Generation-Concepts, Tools, Techniques*, pages 119–145, 1992.

[15] C. F. Bolz, A. Cuni, M. Fijalkowski, and A. Rigo. Tracing the meta-level: PyPy's tracing JIT compiler. In *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, pages 18–25, 2009.

[16] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN Conference on Programming Languages Design and Implementation (PLDI)*, pages 101–113, 2008.

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

[18] D. Callahan, S. Carr, and K. Kennedy. Improving register allocation for subscripted variables. In *PLDI '90: Proceedings of the 1990 ACM SIGPLAN Conference on Programming Languages Design and Implementation (PLDI)*, pages 53–65, 1990.

[19] L. Carter, J. Ferrante, and S. F. Hummel. Hierarchical tiling for improved superscalar performance. In *IPPS 95: Proceedings of the 9th International Symposium on Parallel Processing*, pages 239–245, 1995.

[20] B. Catanzaro, M. Garland, and K. 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.

[21] B. Catanzaro, S. A. Kamil, Y. Lee, K. Asanovi, J. Demmel, K. Keutzer, J. Shalf, K. A. Yelick, and A. Fox. Sejits: Getting productivity and performance with selective embedded jit specialization. Technical Report UCB/EECS-2010-23, EECS Department, University of California, Berkeley, Mar 2010.

[22] C. Chambers, D. Ungar, and E. Lee. An efficient implementation of SELF, a dynamically-typed object-oriented language based on prototypes. In *Proceedings of the 1989 Conference on Object-oriented programming systems, languages and applications*, pages 49–70, 1989.

[23] J. Chame and S. Moon. A tile selection algorithm for data locality and cache interference. In *In 1999 ACM International Conference on Supercomputing*, pages 492–499, 1999.

[24] C. Chen, J. Chame, and M. Hall. Combining models and guided empirical search to optimize for multiple levels of the memory hierarchy. In *Proceedings of the 2005 International Symposium on Code Generation and Optimization (CGO)*, pages 111–122, 2005.

[25] R. Chen, H. Chen, and B. Zang. Tiled-MapReduce: optimizing resource usages of data-parallel applications on multicore with tiling. In *PACT '10: Proceedings of the 19th International Conference on Processors, Architectures, and Compilation Techniques*, pages 523–534, 2010.

[26] M. Cole. Bringing skeletons out of the closet: A pragmatic manifesto for skeletal parallel programming. *Parallel Computing*, 30(3):389–406, 2004.

[27] S. Coleman and K. S. McKinley. Tile size selection using cache organization and data layout. In *PLDI '95: Proceedings of the 1995 ACM SIGPLAN Conference on Programming Languages Design and Implementation (PLDI)*, pages 279–290, 1995.

[28] Continuum Analytics. Blaze. `https://github.com/ContinuumIO/blaze-core`.

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

[30] J. Darlington, Y.-k. Guo, H. W. To, and J. Yang. Parallel skeletons for structured composition. In *Proceedings of the 5th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 19–28, 1995.

[31] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *Proceedings of the 2008 ACM/IEEE International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)*, pages 4:1–4:12, 2008.

[32] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *Proceedings of the 6th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2004.

[33] J. J. Dongarra and A. R. Hinds. Unrolling loops in FORTRAN. *Software - Practice and Experience*, 1979.

[34] P. F. Dubois, K. Hinsen, and J. Hugunin. Numerical python. *Computers in Physics*, 10(3), May/June 1996.

[35] A. Eichenberger, K. O'Brien, K. O'Brien, P. Wu, T. Chen, P. Oden, D. Prener, J. Shepherd, B. So, Z. Sura, A. Wang, T. Zhang, P. Zhao, and M. Gschwind. Optimizing compiler for the CELL processor. In *PACT '05: Proceedings of the 14th International Conference on Processors, Architectures, and Compilation Techniques*, pages 161–172, 2005.

[36] J. Ferrante, V. Sarkar, and W. Thrash. On estimating and enhancing cache effectiveness. In *Proceedings of the Fourth International Workshop on Languages and Compilers for Parallel Computing*, pages 328–343, 1992.

[37] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C.-W. Tseng, and M.-Y. Wu. Fortran d language specification. Technical report, 1990.

[38] M. Frigo and S. G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93:216–231, 2005. Special Issue on ”Program Generation, Optimization, and Adaptation”.

[39] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. W. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz. Trace-based just-in-time type specialization for dynamic languages. In *PLDI ’09: Proceedings of the 2009 ACM SIGPLAN Conference on Programming Languages Design and Implementation (PLDI)*, pages 465–478, 2009.

[40] S. Ghosh, M. Martonosi, and S. Malik. Cache miss equations: an analytical representation of cache misses. In *Proceedings of the 1997 ACM International Conference on Supercomputing (ICS)*, pages 317–324, 1997.

[41] M. Gokhale and R. Minnich. Fpga computing in a data parallel c. In *FPGAs for Custom Computing Machines, 1993. Proceedings. IEEE Workshop on*, pages 94 –101, apr 1993.

[42] A. Hartono, M. M. Baskaran, C. Bastoul, A. Cohen, S. Krishnamoorthy, B. Norris, J. Ramanujam, and P. Sadayappan. Parametric multi-level tiling of imperfectly nested loops. In *Proceedings of the 2009 ACM International Conference on Supercomputing (ICS)*, pages 147–157, 2009.

[43] P. Hatcher, M. Quinn, A. Lapadula, B. Seevers, R. Anderson, and R. Jones. Data-parallel programming on mimd computers. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):377–383, 1991.

[44] E. Hielscher, A. Rubinsteyn, and D. Shasha. Locality optimization for data parallel programs. In *Proceedings of the 2013 ACM International Conference on Supercomputing (ICS)*, Submitted.

[45] U. Hölzle, C. Chambers, and D. Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *ECOOP'91 European Conference on Object-Oriented Programming*, pages 21–38. Springer, 1991.

[46] K. 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.

[47] M. Jiménez and A. Fernández. Register tiling in nonrectangular iteration spaces. *ACM Transactions on Programming Languages and Systems*, 24, 1999.

[48] S. Kamil, C. Chan, L. Oliker, J. Shalf, and S. Williams. An auto-tuning framework for parallel multicore stencil computations. In *Proceedings of the 24th IEEE International Parallel And Distributed Computing Symposium (IPDPS)*, 2010.

[49] D. Kim, L. Renganarayanan, D. Rostron, S. Rajopadhye, and M. M. Strout. Multi-level tiling: M for the price of one. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, pages 51:1–51:12, 2007.

[50] P. M. W. Knijnenburg, T. Kisuki, K. Gallivan, and M. F. P. O'Boyle. The effect of cache models on iterative compilation for combined tiling and unrolling. *Concurrency and Computation: Practice and Experience*, 16:247–270, 2004.

[51] M. D. Lam, E. E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. In *ASPLOS '91: Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 63–74, 1991.

[52] N. Lincoln. Parallel programming techniques for compilers. *SIGPLAN Not.*, 5(10):18–31, Oct. 1970.

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

[54] J. G. Mitchell, A. J. Perlis, and H. R. Van Zoeren. Lc2: a language for conversational computing. In *Symposium on Interactive Systems for Experimental Applied Mathematics: Proceedings of the Association for Computing Machinery Inc. Symposium*, pages 203–214, New York, NY, USA, 1967. ACM.

[55] N. Mitchell, K. Högstedt, L. Carter, and J. Ferrante. Quantifying the multi-level nature of tiling interactions. *International Journal of Parallel Programming*, 26(6):641–670, Dec. 1998.

[56] NVIDIA. CUDA ZONE. `http://www.nvidia.com/cuda`.

[57] O. Oliphant. Python for scientific computing. *Computing in Science and Engineering*, 9:10–20, 2007.

[58] L.-N. Pouchet, C. Bastoul, A. Cohen, and J. Cavazos. Iterative optimization in the polyhedral model: Part II, multidimensional time. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN Conference on Programming Languages Design and Implementation (PLDI)*, pages 90–100, 2008.

[59] L.-N. Pouchet, U. Bondhugula, C. Bastoul, A. Cohen, J. Ramanujam, and P. Sadayappan. Combined iterative and model-driven optimization in an automatic parallelization framework. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)*, pages 1–11, 2010.

[60] 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.

[61] A. Raman, A. Zaks, J. W. Lee, and D. I. August. Parcae: a system for flexible parallel execution. In *PLDI '12: Proceedings of the 2012 ACM SIGPLAN Conference on Programming Languages Design and Implementation (PLDI)*, pages 133–144, 2012.

[62] L. Renganarayana, U. Bondhugula, S. Derisavi, A. E. Eichenberger, and K. O'Brien. Compact multi-dimensional kernel extraction for register tiling. In *Proceedings of the 2009 ACM/IEEE International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)*, pages 45:1–45:12, 2009.

[63] L. Renganarayana, U. Ramakrishna, and S. Rajopadhye. Combined ILP and register tiling: analytical model and optimization framework. In *Proceedings of the 18th international Workshop on Languages and Compilers for Parallel Computing*, pages 244–258, 2006.

[64] L. Renganarayanan, D. Kim, S. Rajopadhye, and M. M. Strout. Parameterized tiled loops for free. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Languages Design and Implementation (PLDI)*, pages 405–414, 2007.

[65] L. Renganarayanan, D. Kim, M. M. Strout, and S. Rajopadhye. Parameterized loop tiling. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 34(1):3:1–3:41, May 2012.

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

[67] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. W. Hwu. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *Proceedings of the 13th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 73–82, 2008.

[68] V. Sarkar. Optimized unrolling of nested loops. In *Proceedings of the 2000 ACM International Conference on Supercomputing (ICS)*, pages 153–166, 2000.

[69] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens. Scan primitives for gpu computing. In *Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, GH '07, pages 97–106, Aire-la-Ville, Switzerland, Switzerland, 2007. Eurographics Association.

[70] L. Shapiro and G. C. Stockman. *Computer Vision*. Prentice Hall, 2001.

[71] J. Shirako, K. Sharma, N. Fauzia, L.-N. Pouchet, J. Ramanujam, P. Sadayappan, and V. Sarkar. Analytical bounds for optimal tile size selection. In *Proceedings of the 21st International Conference on Compiler Construction (CC)*, pages 101–121, 2012.

[72] J. Sipelstein and G. E. Blelloch. Collection-Oriented languages. In *Proceedings of the IEEE*, pages 504–523, 1991.

[73] G. Steele Jr and W. 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.

[74] V. Tabatabaee, A. Tiwari, and J. K. Hollingsworth. Parallel parameter tuning for applications with performance variability. In *Proceedings of the 2005 ACM/IEEE International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)*, 2005.

[75] D. Tarditi, S. Puri, and J. 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.

[76] K. J. Thurber and J. W. Myrna. System design of a cellular apl computer. *IEEE Trans. Comput.*, 19(4):291–303, Apr. 1970.

[77] A. Tiwari, C. Chen, J. Chame, M. W. Hall, and J. K. Hollingsworth. A scalable auto-tuning framework for compiler optimization. In *Proceedings of the 23rd IEEE International Parallel And Distributed Computing Symposium (IPDPS)*, pages 1–12, 2009.

[78] A. Tiwari and J. K. Hollingsworth. Online adaptive code generation and tuning. In *Proceedings of the 25th IEEE International Parallel And Distributed Computing Symposium (IPDPS)*, pages 879–892, 2011.

[79] M. J. Voss and R. Eigenmann. ADAPT: Automated de-coupled adaptive program transformation. In *Proceedings of the 1999 International Conference on Parallel Programming (ICPP)*, pages 163–170, 1999.

[80] P. Wadler. Applicative style programming, program transformation, and list operators. In *Proceedings of the 1981 conference on Functional programming languages*

*and computer architecture*, FPCA '81, pages 25–32, New York, NY, USA, 1981. ACM.

[81] C. Whaley, A. Petitet, and J. J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27:3–35, Jan 2001.

[82] M. E. Wolf and M. 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.

[83] M. E. Wolf and M. S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):452–471, Oct. 1991.

[84] M. E. Wolf, D. E. Maydan, and D.-K. Chen. Combining loop transformations considering caches and scheduling. In *Proceedings of the 29th annual ACM/IEEE International Symposium on Microarchitecture (MICRO)*, pages 274–286, 1996.

[85] K. Yotov, X. Li, G. Ren, M. Cibulskis, G. DeJong, M. Garzaran, D. Padua, K. Pingali, P. Stodghill, and P. Wu. A comparison of empirical and model-driven optimization. In *PLDI '03: Proceedings of the 2003 ACM SIGPLAN Conference on Programming Languages Design and Implementation (PLDI)*, pages 63–76, 2003.

[86] K. Yotov, X. Li, G. Ren, M. Garzaran, D. Padua, K. Pingali, and P. Stodghill. Is search really necessary to generate high-performance BLAS? *Proceedings of the IEEE*, 93(2):358–386, 2005.

[87] Y. Yu, P. K. Gunda, and M. Isard. Distributed aggregation for data-parallel computing: interfaces and implementations. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP)*, pages 247–260, 2009.

[88] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and J. 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.

[89] H. Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers*. ACM Press, 1990.