Locality Optimization for Data Parallel Programs

Eric Hielscher
hielscher@cs.nyu.edu

Alex Rubinsteyn
alexr@cs.nyu.edu

Dennis Shasha
shasha@cs.nyu.edu

Computer Science Department New York University New York, NY, 10003

ABSTRACT

Productivity languages such as NumPy and Matlab make it much easier to implement data-intensive numerical algorithms. However, these languages can be intolerably slow for programs that don't map well to their built-in primitives.

In this paper, we discuss locality optimizations for our system Parakeet, a just-in-time compiler and runtime system for an array-oriented subset of Python. Parakeet dynamically compiles whole user functions to high performance multi-threaded native code. Parakeet makes extensive use of the classic data parallel operators **Map**, **Reduce**, and **Scan**. We introduce a new set of data parallel operators, **TiledMap**, **TiledReduce**, and **TiledScan**, that break up their computations into local pieces of bounded size so as better to make use of small fast memories. We introduce a novel tiling transformation to generate tiled operators automatically. Applying this transformation once tiles the program for cache, and applying it again enables tiling for registers. The sizes for cache tiles are left unspecified until runtime, when an autotuning search is performed. Finally, we evaluate our optimizations on benchmarks and show significant speedups on programs that exhibit data locality.

1. INTRODUCTION

Productivity languages such as NumPy [19] and Matlab [18] make it much easier to implement data-intensive numerical algorithms. However, these languages rely on library functions to attain acceptable performance and can become intolerably slow for programs which don't map well on to precompiled primitives. In this paper, we discuss our system Parakeet [22], a just-in-time compiler and runtime system for an array-oriented subset of Python. Parakeet dynamically compiles whole user functions to high performance multi-threaded native code. This paper focuses on our use of data parallel operators as a basis for locality enhancing optimizations. We present a high-level syntactic transformation which enables tiling for better use of cache and registers.

The data parallel model allows programmers to expresses 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.

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

In summary, the contributions of this paper are:

- A novel code transformation that tiles data parallel programs to improve both cache and register usage.
- An online autotuning search to select tile sizes.
- A just-in-time optimizing compiler for an array-oriented DSL embedded in Python, which can readily outperform naive C implementations on a variety of tasks.

2. THE PARAKEET DSL AND COMPILER

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

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

Listing 1: Simple Parakeet function

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

This example does not make use of any data parallel operators. In fact, it is possible to generate code with Parakeet using only its capacity to efficiently compile loops and scalar operations. However, even greater performance gains can be achieved through either the explicit use of data parallel operators or, commonly, the use of constructs which implicitly generate data parallel constructs. For example, if you were to call `add1` with a vector, 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:

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

Listing 2: Add 1 to every element

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

Functions which contain data parallel operators are aggressively optimized and executed across multiple cores. Aside from the tiling transformation described later, we also perform operator fusion [29], as well as more standard compiler optimizations such as common subexpression elimination, loop unrolling, and scalar replacement. Parakeet generates native code using the LLVM compiler infrastructure [16]. 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.

$$
\begin{array}{lcl}
\text{expression} & ::= & \text{Const} \\
 & | & x \\
 & | & e_1 \odot e_2 \\
 & | & [e_1, \ldots, e_n] \\
 & | & e_1[e_2] \\
 & | & \lambda x_1, \ldots, x_n.\text{block} \\
 & | & \mathbf{Map}_\alpha(f, e_1, \ldots, e_n) \\
 & | & \mathbf{Reduce}_\alpha(f, \oplus, e_{\text{init}}, e_1, \ldots, e_n) \\
 & | & \mathbf{Scan}_\alpha(f, \oplus, f_{\text{emit}}, e_{\text{init}}, e_1, \ldots, e_n) \\
\text{statement} & ::= & x = e \\
 & | & \textbf{return } e \\
 & | & \textbf{if } e_{\text{cond}} \textbf{ then } \text{block}_{\text{true}} \textbf{ else } \text{block}_{\text{false}} \\
 & | & \textbf{for } x \textbf{ in } e_{\text{seq}} \text{ block}_{\text{body}} \\
\text{block} & ::= & \text{statement}^+
\end{array}
$$

Figure 1: Parakeet's Internal Representation

3. TILED OPERATORS

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.

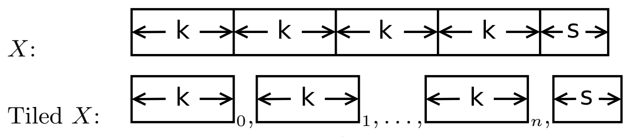


Figure 2: Decomposition of an Array Into n k-length Tiles Plus An s-length Straggler Tile

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

Figure 3: Visual Semantics of Tiled Data Parallel Operators

In order to provide a convenient single abstraction for dividing data parallel operator computations into locality-friendly pieces, we introduce *tiled data parallel operators*, one for each regular data parallel operator. Tiled data parallel operators are a natural generalization of data parallel operators that, rather than applying their nested 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 nested functions on these tiles. Users never program directly with tiled data parallel operators – 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 Section 4.

Figure 2 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 gives a visualization of the semantics of tiled data parallel operators. Tiled data parallel operators can take multiple arguments and axes just like regular data parallel operators, but we omit them from this picture for clarity. For example, a **TiledMap** decomposes its input arguments into tiles, executes its nested function on each tile (including the straggler tile), and then concatenates the results. The semantics of **TiledReduce**s and **TiledScan**s are similarly direct generalizations of their untiled counterparts.

Tiled data parallel operators also take two versions of their nested functions – one specialized for the specific tile size k (denoted f^k), and one generic to array length (simply denoted f) which is used to process the straggler tile. Knowledge that the f^k nested 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.

Decomposing data parallel operators 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 data parallel operators 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. 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.

4. TILING TRANSFORMATION

In this section, we present our algorithm for automatically translating a Parakeet function with data parallel operators into a version with tiled data parallel operators. The basic idea should be intuitive – we wrap data parallel operators in tiled versions of themselves (say, a **Map** in a **TiledMap**) – and in this way use the tiled data parallel operator to break up the original computation into locality-friendly pieces. The original data parallel operators become the nested functions of the tiled data parallel operators, as the original computation still needs to be performed on each tile. However, there are a number of issues that make things somewhat more complicated.

Let us use summing the rows of a 2D matrix as a simple running example. Parakeet code for this is given in Listing 3. The iteration pattern through the elements of the array for this version is shown on the left side of Figure 4 – 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 Section 3. For example, if the data is layed out such that adjacent elements of columns (rather than rows) are adjacent in memory, then whenever an element of the array is read some number of neighboring elements in its column will also be brought into cache as they'll be in the same cache line as the read element. If the rows are larger than 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. Let us walk through the steps required to tile this program for cache.

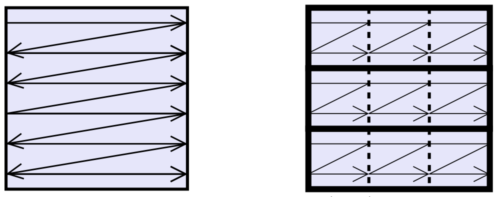


Figure 4: Iteration Order of Untiled (left) and Tiled (right) 2D Row Sums

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

def sum_row(row):
  return Reduce(lambda x:x, row, init=0,
                combine=add2, axes=[0])

def sum_rows(Xs):
  return Map(sum_row, Xs, axes=[0])
```

Listing 3: Sum Each Row of a 2D Array

We would like to break up both the **Map** and the **Reduce** by 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 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 series of partial rows. The divisions added by the **TiledReduce** are represented in the figure by the dashed lines. The iteration order through each of the final tiles is represented on the right side of the figure by the arrows. 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 `Xs`, 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.

The final tiled code for summing the rows of a 2D array is given below in Figure 5. Let's break it down piece by piece to explain how our algorithm produces this code.

4.1 Visited Operators List

First, notice that at a high level the code is as we would expect – a **TiledMap** is contained in the outermost function `tiledsum_rows`, whose nested function `tiledsum_row` contains a **TiledReduce**, whose nested function `sum_rows` in turn contains the original computation. The first thing our algorithm must do is keep track of the nesting of data parallel operators encountered as it steps through a program so as to be able to create a tiled copy of that nesting.

A formalized description of our entire algorithm is given in figures 6, 7, and 8. In the formal description, the sequence of encountered data parallel operators is represented by σ. The generic operation of *tiling* a statement, a block of statements, or a data parallel operator is represented by the $[\![ ]\!]$ operator. Recall that the entry point into Parakeet is always either a data parallel operator or an outermost function called from Python. Tiling an outermost function involves simply tiling its block of statements.

The rest of the parameters of the **TiledMap** match those of the original **Map** – it iterates over the variable `Xs` on axis 0. Thus our algorithm can simply copy these parameters from the original **Map** and use them for the tiled version.

```
def identity(x):
  return x

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

def sum_row(rowTile):
  return Reduce(identity, rowTile, init=0,
                combine=add2, axes=[0])

def sum_rows(XsTile):
  return Map(sum_row, XsTile, axes=[0])

def tiledadd2(as,bs):
  return Map(add2, as, bs)

def tiledsum_row(XsMapTile):
  return TiledReduce(sum_rows, XsMapTile, init=0,
                     combine=tiledadd2, axes=[1])

def tiledsum_rows(Xs):
  return TiledMap(tiledsum_row, Xs, axes=[0])
```

Figure 5: IR for Tiled Sum Each Row of a 2D Array

Next, let's examine the `tiledsum_row` function and its **TiledReduce**. It iterates over the `XsMapTile`, analagous to the original **Reduce** iterating over `row`, and its initial value of 0 is the same as that of the **Reduce**. Its nested function is the original `sum_rows` function that was tiled, which is what we would expect since it is the final tiled data parallel operator in the nesting. However, its axes and combine parameters require further explanation.

4.2 Axes of Tiled Operators

The **TiledReduce**'s iteration axis for the `XsMapTile` is 1, not 0 like in the original **Reduce**. To understand why, look again at the iteration order shown in Figure 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 data parallel operators and regular data parallel operators is that tiled data parallel operators always operate on tiles of the outermost arguments that are the same rank as the entire outermost arguments themselves. In contrast, regular data parallel operators operate on pieces of the arguments with progressively decreasing rank, as each regular operator slices away one dimension. Thus the axes of iteration for regular operators have a local view on the input arguments to the outermost function – they only see the dimensions of the arguments that remain after any previous operators have sliced some dimensions away. This is why, even though the **Reduce** in the original code iterates over what is axis 1 of the outermost 2D input argument `Xs`, its axis of iteration over its 1D row is 0.

In order to properly update the axes of variables such that they fit the tiled operators, we need to keep around some state for each variable we encounter. For this we maintain a mapping from variables to the axes at which prior adverbs in the nesting have sliced them. In the formal description, this mapping is denoted by Δ. Using these lists of axes, we can map from the local axis of the original adverb back to the global axis of the entire tile/outer variable as required.

4.3 Tiled Combine Functions

The combine function of the **TiledReduce** in Figure 5 also needs further explanation. Recall that the purpose of a combine function is to provide a way to combine two partial results of a reduction. For a **TiledReduce**, we need the combine function to combine two partial results of executing the nested function on *tiles*.

Let's walk through expanding the **TiledReduce**'s combine function. The number of data parallel operators in the visited data parallel operators list σ is one adverb prior to this point: the **Map**. Thus we've expanded the arguments to the **TiledReduce**'s nested function as well as the nested function's return value by 1 rank. Hence we need to increase the rank of the arguments and results of the **Reduce**'s combine function by 1 in order to create a combine function suitable for the **TiledReduce**. We do this by calling a helper function we name *BuildTree*. This function wraps the **Reduce**'s combine function `add2` in one **Map**, as shown in Figure 5.

The purpose of this **Map** is to "peel off" the rank that was added by the previous adverb having been tiled. This way, we can get to arguments of the rank that the original combine function expects, and then call the original function on those arguments. By mapping across the extra dimensions added (one in this case), we ensure that we call the combine function on each element of the tiles, thus maintaining the semantics of the original program.

Since there are no more data parallel operators to tile, we are now ready to insert the original computation as the nested function of the final tiled data parallel operator, and the tiling transformation terminates. The reader following along with the formal algorithm will notice that a **Reduce** or a **Scan** always terminates the tiling algorithm. This is because it is not safe in general to split the results of one reduction and have these partial results form the input to another one (for example, the partial results of the minimum of an array aren't meaningful). In addition, notice that the insertion of the original computation in the formal algorithm actually involves another use of the *BuildTree* function. This is in order to support nested non-operator statements, as described next.

4.4 Nested Statements

One last issue not covered in the summing rows example is what happens in the presence of statements that don't include data parallel operators inside functions being tiled (for example, scalar operations, indexing into variables, or control flow). In the case of the outermost function, these statements are simply left alone. However, inside a nested function of a tiled operator, we must be careful to maintain the semantics of the original program. If we simply added these statements both to the nested functions of the tiled operators in addition to keeping them in the nested functions of the inner regular operators, this may result in an error. For example, in the case of indexing into a variable, this would amount to indexing into a tile, and then in the innermost computation, indexing again into a piece of that tile. This would index once too many times, and break the program's semantics.

To solve this issue, we do two things. First, we disallow control flow in functions being tiled; if control flow is

α	::=	Axes along which an adverb slices each of its arguments
σ	::=	Ordered sequence of visited adverbs
Δ	::=	Maps variables to the list of axes remaining of the original variables of which they are a piece
ϵ	::=	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
$\mathrm{FV}(e)$	::=	The set of free variables in expression e

$$
\begin{array}{lll}
\mathrm{BuildTree}(\sigma, \epsilon, x_1, \ldots, x_n, & & \\
\qquad\qquad \mathit{block}) & ::= & \sigma_0(\mathrm{vars}_0, \lambda \mathrm{vars}_0. \\
 & & \quad \sigma_1(\mathrm{vars}_1, \lambda \mathrm{vars}_1. \\
 & & \qquad \ldots \\
 & & \qquad\quad \sigma_d(\mathrm{vars}_d, \lambda \mathrm{vars}_d.\mathit{block}) \ \ldots)) \\
 & & \text{where} \\
 & & \quad d = |\sigma| \\
 & & \quad \mathrm{vars}_i = \{x_j | i \in \epsilon[x_j]\}
\end{array}
$$

$$
\begin{array}{lll}
[\![f = \lambda x_1, \ldots, x_n.\mathrm{body}]\!] & \longmapsto & f' = \lambda x_1, \ldots, x_n.\mathrm{body}' \\
 & & \text{where} \\
 & & \quad \sigma = \langle\rangle \\
 & & \quad \Delta[x_i] = \langle 0, 1, \ldots, \mathrm{rank}(x_i) - 1 \rangle \\
 & & \quad \epsilon = \{\} \\
 & & \quad \mathrm{body}' = [\![\mathrm{body}]\!]^{\sigma,\Delta,\epsilon}
\end{array}
$$

Figure 6: Tiling Transformation, Definitions

Statement and Expression Transformations

$$
\begin{array}{lll}
[\![\mathbf{return}\ e]\!]^{\sigma,\Delta,\epsilon} & \longmapsto & \epsilon,\ \mathbf{return}\ [\![e]\!]^{\sigma,\Delta,\epsilon}
\end{array}
$$

$$
\begin{array}{lll}
[\![x{=}e]\!]^{\sigma,\Delta,\epsilon} & \longmapsto & \epsilon',\ \mathrm{x} = e' \\
 & & \text{where} \\
 & & \quad \epsilon'[x] = \bigcup \{\epsilon[y] \mid y \in \mathrm{FV}(e)\} \\
 & & \quad \text{if } e \text{ contains an adverb:} \\
 & & \qquad e' = [\![e]\!]^{\sigma,\Delta,\epsilon} \\
 & & \quad \text{else:} \\
 & & \qquad \sigma' = \langle \mathbf{Map}_0, \mathbf{Map}_0, \ldots \rangle \text{ such that} \\
 & & \qquad\qquad |\sigma'| = |\epsilon'[x]| \\
 & & \qquad e' = \mathrm{BuildTree}(\sigma', \epsilon', \mathrm{FV}(e), \\
 & & \qquad\qquad\qquad\qquad \langle \mathbf{return}\ e \rangle)
\end{array}
$$

$$
\begin{array}{lll}
[\![\mathit{block}]\!]^{\sigma,\Delta,\epsilon} & \longmapsto & \mathit{block}' \\
 & & \text{where} \\
 & & \quad \mathit{block}' = \langle\rangle \\
 & & \quad \epsilon' = \epsilon \\
 & & \quad \text{for } s \in \mathit{block} : \\
 & & \qquad \epsilon', s' = \langle [\![s]\!]^{\sigma,\Delta,\epsilon'} \rangle \\
 & & \qquad \mathit{block}' = \mathit{block}' +\!\!+\ s'
\end{array}
$$

$$
\begin{array}{lll}
[\![e]\!]^{\sigma,\Delta,\epsilon} & \longmapsto & e \ (\text{if } e \text{ is not an adverb})
\end{array}
$$

Figure 7: Tiling Transformation, Statements

Adverb Transformations

$[\![\mathbf{Map}_\alpha(v_1,\dots,v_n,f)]\!]^{\sigma,\Delta,\epsilon} \longmapsto \mathbf{TiledMap}_{\alpha'}\ (v_1,\dots,v_n,f^\uparrow)$

where

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

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

where

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

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

Figure 8: Tiling Transformation, Adverbs

found, the tiling transformation is undone and the code is left untiled. Control flow can be handled in these cases by predication [3], but we leave this for future work.

Next, we place all of the other non-operator statements only in the nested functions of the tiled adverbs, and remove them from the copy of the original program that includes the original operators. In the case of indexing, e.g., this is the only legal possibility, as if the indexing were delayed until the inner regular operators 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 innermost computation, which can contain arbitrary non-operator code, is left untouched.

When we get to the last operator in the tree and want to splice in the altered version of the original program, we take the innermost nested function (that contains no operators), and call the *BuildTree* function on its body. This function doesn't wrap the body in **Map**s as in the case of tiled combine functions. Instead, it wraps the block in the tree of operators from the original program. Thus we create a tree with the original operators, but devoid of any non-operator statements.

One thing we must ensure, however, is that the proper variables are passed as arguments to each operator in this tree. For this we keep track of the *list of nesting depths* at which each variable was an argument to a data parallel operator. In the formal algorithm, this additional mapping is denoted by ϵ. Often variables are passed into nested functions as closure arguments for use in deeper levels of a program's nesting. We don't want to alter the behavior of the program by adding these as arguments to the operators. Hence for each operator we need to add we look up in the ϵ mapping which free variables of the innermost block were arguments to an operator at that nesting depth. This allows us to add the proper arguments to each operator, and we pass the other needed arguments through as closure arguments to the operator's nested function.

Finally, we must tile the non-operator statements as well. This is because they are present in tiled functions, and so will receive tiles as arguments instead of those of the original ranks. To do this, we compile a list of the statement's free variables, and from this a list of all nesting depths at which these variables were arguments to adverbs. We then call the *BuildTree* function on the statement to wrap it in a number of **Map**s equal to the number of such nesting depths. We use the depths in the same way as in building the inner computation to pass the correct variables to each **Map**. This has the effect of peeling off the extra ranks as discussed in the case of 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, data parallel operator fusion is able to combine them all into a single tree of **Map**s, mitigating some of this cost. An alternative 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 leave this for future work.

5. TILE SIZES

We run our transformation twice, once to general a level of tiles for the L1 cache, and once to generate tiles for registers. The register tiles are set statically at compile time using a heuristic that attempts to use as many of the registers available on the target machine without exceeding that number. There are two methods used to set cache tile sizes in the literature: statically via models, and empirically via autotuning.

5.1 Statically Estimating Tile Sizes

A body of work exists on attempting to devise models for general tiling or tiling for specific domains that allows for good static setting of tile size parameters [7, 23, 33, 34]. However, in our experiments we weren't able to get any of the static models to perform as well as a dynamic tile size search, which backs up the continued widespread use of autotuning. In our setup, we use the DL and ML algorithms from [23] 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 provide a surface of minimum and maximum tile sizes respectively in the tile search space. We generate square-shaped tiles on each boundary surface and use these as starting points for a dynamic tile size search, described in the next section.

5.2 Online Autotuning Tile Sizes

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 [25]. 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.

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 standard matrix multiplication 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 if the sizes 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 9). 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

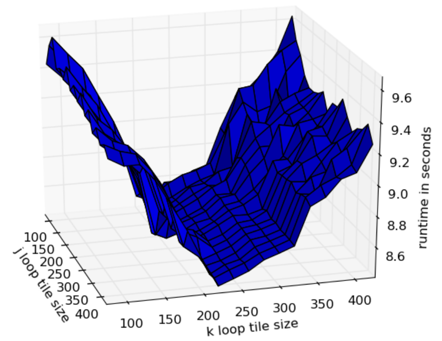


Figure 9: Matrix Multiply Performance vs Tile Size

equal to the half the difference between these estimates. A sample for each tile size forms a candidate setting of tiles, and a number of these 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 heuristics for determining what percentage of the total computation should be spent searching, how long to let each candidate evalutation run, and how many times no change in best point should lead to a termination of the search.

6. EVALUATION

We evaluate our cache and register tiling optimizations on K-Means Clustering and Matrix Multiplication. We have implemented a number of different programs in Parakeet – for example, Gaussian blurring and $O(N^2)$ N-Body simulations – but must omit their results for lack of space.

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.

In all of these performance results, register tiling and cache tiling were turned on and off together. In the interest of space, and since we don't vary any parameters for register tiling across any of the runs, we don't present separate numbers for register tiling. It was very substantial for performance however, accounting for 60% of the speedups due to tiling on average. Also, aside from those in the section on Compile Time, no performance results include compile times.

6.1 Matrix Multiplication

In this section, we present results for a 2D matrix multiplication benchmark, the actual Parakeet code for which is in Listing 4. Note that **AllPairs** is Parakeet syntactic sugar for two nested **Map**s each of which iterates over one of the arguments to the **AllPairs**. Note that for these numbers, we pre-transposed the matrices such that they are both laid out for the best performance. This is actually the worst-case scenario for our tiling optimizations, as they would provide a substantially larger (roughly 12X) performance boost when run on improperly laid out data.

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

def mm(Xs, Ys):
  return AllPairs(dot, Xs, Ys)
```

Listing 4: Parakeet Matrix Multiply

6.1.1 *Overall Performance*

In Figure 10, we present Parakeet's performance on matrix multiplication as compared against NumPy's. NumPy uses an implementation of BLAS, a standard linear algebra interface, 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 [30] implementation that comes with Ubuntu Linux. For optimal performance, one needs to do a lengthy autotuning of ATLAS for a particular machine. We use the stock version as we simply want to provide a rough reference point for calibrating the meaning of our results. In each of these graphs, we used our online autotuner to find good cache tile sizes.

On the left-hand side 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 right-hand side of 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.

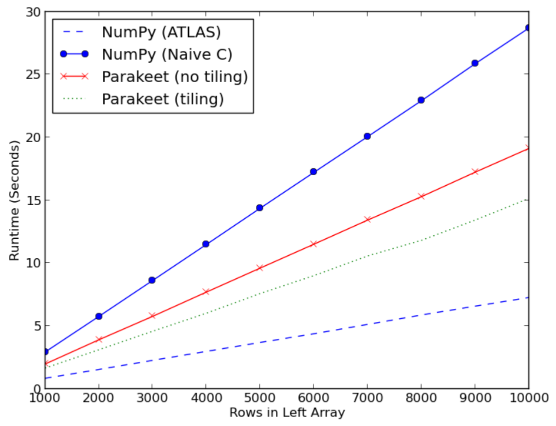
6.1.2 *Autotuning Performance*

In Figure 11, 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 [23], as well as the performance when using the average of the DL and ML estimates as the fixed tile sizes.

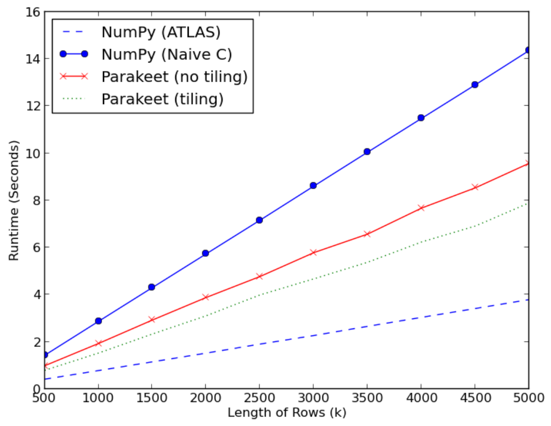
By comparing the difference between the runtime with the search and with cached tile sizes, we see that the overhead of performing the search is very small. The time to switch between different tile sizes during a search is close to 0, and so any overhead is almost entirely due to the penalty from running 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.

6.1.3 *Performance vs. Other Compilers*

To provide a test of Parakeet's performance relative to other compilers, we compare Parakeet's performance to a



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



(b) Runtimes with varying lengths of rows

Figure 10: Matrix Multiply Runtimes

Array Size	Parakeet (with search)	Parakeet (cached tile sizes)	DL sizes	ML sizes	DL&ML Avg
3000x3000	4.58s	4.51s	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

Figure 11: Matrix Multiply Autotuner Performance

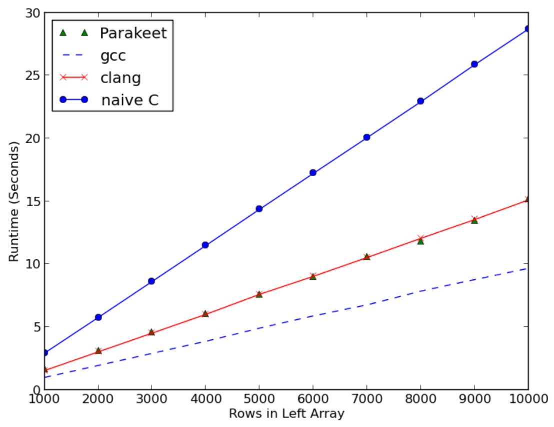


Figure 12: Matrix Multiply Performance Compared To C

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. The results are shown in Figure 12.

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 roughly half of gcc's relative performance gain over clang is due to gcc having a better vectorizer. We hope that in the future, we can either add a vectorizer to Parakeet or that LLVM's vectorizer will improve.

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

Figure 13: K-Means Performance with k = 1000, 500 features, and 10 iterations

6.2 K-Means Clustering

We present results for K-Means Clustering in Figure 13. 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 it 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.

6.3 Compilation Time

In Figure 14, 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 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

	Matrix Multiply	K-Means
No Tiling	0.19s	0.10s
Cache Tiling	0.27s	0.31s
Cache Tiling + Register Tiling	1.26s	1.29s

Figure 14: Parakeet Compilation Times

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.

7. RELATED WORK

Our work builds upon a range of existing fields of study including optimization of data parallel and array programs, just-in-time compilation, loop optimizations, analytic performance modeling, and autotuning. Our system is unique in that it is the first to unify: high level, dynamic array languages, JIT compilation, high level tiling, and online autotuning into one system.

The first language to feature data parallel abstractions was APL [14], 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 [27] and parallelization [17]. As computers with massively parallel hardware became more common in the 1980s, many languages such as C [?], Fortran [10], and Lisp [24] 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 [12], GPU programs [26, 5], and even the coordination of distributed computations [35].

There have been numerous projects that accelerate high level languages, be it via high performance library calls as in NumPy [9]; via JIT compilation as in LuaJIT [20], MaJIC [1], and the fledgling Numba project [8]; or via dynamic compilation to C++ as in Copperhead [5]. Parakeet lies in some sense in between Numba and Copperhead, but .

There has been extensive research on loop optimizations, including cache tiling, register tiling, data copying, data padding, and loop unrolling optimizations (e.g. [15, 31]). Much of this work uses the polyhedral model, a sophisticated technique that transforms the iteration space of a loop nest via algebraic manipulation [4, 31]. This includes work on combining analytic models for selecting tile sizes with offline autotuning [21] and work on on generating parameterized tiles for imperfectly nested loops [13].

There have been a host of analytic models for determining parameter settings for dense matrix multiplication [7, 33, 34]. 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 [32]. Recent work on analytical bounds for optimal tile sizes [23].

In recent years, offline autotuning has emerged as the accepted best practice for optimizing numerical code [2]. Libraries such as ATLAS for dense linear algebra [30] and FFTW for Fourier transforms [11] 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 autotuning systems include Chill [6] and Active Harmony [28].

8. CONCLUSION

We have presented tiled data parallel operators, novel high-level syntactic constructs for breaking up data parallel programs into locality-friendly pieces. Our tiling transformation automatically generates tiled versions of programs from programs written using regular operators in Parakeet, a high level array-oriented DSL embedded in Python. 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 two benchmarks and show significant performance improvements over untiled code and favorable performance compared to C versions.

9. REFERENCES

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

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

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

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

[11] 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".

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] C. Lattner. LLVM: An infrastructure for multi-stage optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002. *See* `http://llvm.cs.uiuc.edu`.

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

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

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

[20] M. Pall. LuaJIT. `http://luajit.org`.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

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

[28] 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.

[29] 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.

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

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.