Techniques for Static and Dynamic Compilation of Array Expressions

Mark Florisson

August 23, 2012

MSc in High Performance Computing
The University of Edinburgh
Year of Presentation: 2012

Abstract

This thesis evaluates compiler techniques to efficiently evaluate local array expressions such as those found in APL, Fortran 90 and many other languages. We show that merely eliminating temporary arrays is not enough, and we contribute a reusable open source compiler called `minivect` that generates code which can outperform commercial Fortran compilers for even moderate data sizes in several test-cases, while retaining full runtime generality such as broadcasting as found in NumPy (similar to Fortran's `SPREAD` intrinsic) and arbitrarily strided arrays. In extreme cases we measure speedups up to 9x compared to GNU Fortran, and up to 1.7x for Intel Fortran. We show how these speedups may be further increased through SIMD vector-sized transposes for certain array expressions, and by computing tile sizes at runtime.

We furthermore provide insights and a working implementation of in-memory Abstract Syntax Tree (AST) remapping from an original AST and type system to a foreign AST and type system, enabling efficient full or partial mappings, allowing reuse of external compiler technology. We provide a working implementation of this for array expressions in the Cython language. We also contribute a minimal library that uses lazy evaluation combined with runtime compilation to generate efficient code. We also show how a compiler may be designed to facilitate adding new code generators with minimal effort without the need for an explicit intermediate representation.

We finally show how we can circumvent temporary arrays by proving data independence using traditional dependence tests in various situations for arbitrary runtime arrays (as opposed to compile-time aliases). In case of possible dependence we construct distance and direction vectors at runtime in order to use well-known compile-time optimizations, which we implement at runtime by adapting the array view on memory.

Acknowledgements

This dissertation would not have been possible without the support of many people. I would like to thank Alan Gray and Alistair Hart for supervising this dissertation, for accepting a proposal that was radically different from the advertised project, for listening patiently to my incessant rambling and for their enthusiasm, support and their suggestions to write a better thesis. I would like to thank Dag Sverre Seljebotn for another year of mentoring for the Google Summer of Code, and for his many suggestions and insights that have made this a successful as well as fun and reusable project, and for having undoubtedly changed the direction and scope of the project and this thesis for the better. Thanks go out to Frédéric Bastien for providing Theano benchmarks, for discussing ways for Theano and our project to cooperate. Additional thanks to Dag Sverre Seljebotn and Frédéric Bastien for their suggestions on how to improve the benchmarks and the readability of this dissertation, which was too often overseen. I would like to thank my family for always supporting my studies, and I would like to thank my wife for all her love and support, and for making the sun shine even when it rains.

Contents

1 Introduction

Array expressions, also known as vector expressions, are expressions on multi-dimensional arrays in which the expression is applied to the individual data items of the arrays. Array expressions, pioneered by APL [1], can now be found in many languages and libraries, such as Fortan 90 and Matlab, many of the C++ expression template based libraries such as Eigen [2], Blitz++ [3] and uBlas [4], and among many others, the NumPy [5] library for Python. Array expressions are useful not only for conciseness, but also to enable optimizations since they make data access patterns more transparent to compilers and obviate any reliance on auto-vectorization capabilities of the compiler. Further restrictions imposed by the language on element-wise functions (for instance Fortran's `pure` restriction) allow compilers to change the element-wise traversal order and enable data parallelism, as well as generate several specializations of the code when data layout is fixed at runtime, allowing portable performance for evaluation across different data sets with different striding patterns and data alignment.

Array expressions may, depending on the language, allow further polymorphism by allowing dimensionality of the operands to be fixed at runtime [6] [7], as well as allow operands to be implicitly stretched (broadcast) along an axis. Reductions are explicit and the language or library can define the special semantics, allowing parallel execution in any appropriate form, and even execution on accelerator devices and GPUs.

We contribute a reusable array expression compiler, which for the purposes of this dissertation only covers element-wise expressions. We implement all NumPy rules and allow broadcasting [8] in any dimension, as well as arbitrary data ordering and overlapping memory between operands and function parameters. We adopt the model where array slicing creates new views on existing data, allowing the user tighter control over when and how data is copied, since data no longer needs to remain contiguous. The price paid for this generality is however that we no longer know at compile time what the exact data layout will be at runtime, unless explicitly annotated by the user and verified at runtime. We operate with the restriction, unlike NumPy, that the dimensionality of the arrays is fixed at compile time instead of runtime, in order to generate code with maximal efficiency. To retain stable performance across any combination of data layouts, several strategies can be taken:

1. We can apply the array expression on small data chunks, retaining the data in the cache while all operations are performed iteratively, or in a fused fashion. This allows data to be retained in the cache before eviction due to capacity constraints.
2. We could defer compilation until runtime, possibly combined with lazy evaluation of the expressions, which may have somewhat higher - but constant in the size of the data - overhead.
3. We could create one or several up-front specializations [1] for various runtime situ-

[1]With specialization we mean a special version of the code which is tailored to the input types in a certain way, in this case efficiency.

ations we want to cover, and select the specialization at runtime. This can further be combined with a runtime component that can reorder the axes of the array operands in question, to reduce the number of specializations (by reducing the number of possible runtime data layouts) and increase the cases that can be handled with near-optimal efficiency.

Our array expression compiler is a reusable compiler backend that can be suitably used with either of the last two aforementioned approaches, and can generate static C specializations as well as generate code in the form of an LLVM [2] [9] intermediate representation, which is further optimized by the LLVM runtime and compiled to native code (or interpreted when the target platform is not supported). The compiler is fully customizable and the generated code parametric, to allow various auto-tuned or otherwise computed runtime parameters. Furthermore, it performs explicit vectorization, allowing multiple versions of the code to be compiled simultaneously, and the optimal version to be selected at runtime based on the supported SIMD CPU instruction set [10].

To this low-level back-end we provide two front-ends. The first front-end is implemented as part of the Cython compiler [11] [12], an ahead-of-time compiler which compiles a language superset of Python to C or C++. We implement support for array expressions in the Cython language by mapping the array expressions at compile time to our lower level compiler, which creates several different specializations and feeds the results back into the compilation unit. The compiler at the Cython side then generates the code needed to select the right specialization at runtime, depending on data layout of the respective operands and possible broadcasting dimensions. The compiler also generates code to perform auto-tuning at runtime for tiling and threading parameters. This cannot be performed at compile time, since Cython is a source-to-source compiler, and the generated source files are often distributed.

Our second front-end is a minimal implementation of the second approach. It provides a library for lazy-evaluation that builds an expression graph at runtime, recording the operations as they are executed at runtime. This expression graph is compiled at runtime and specialized for the data sets and operations in question. Note that our restriction on polymorphic dimensionality at compile time is lifted, since our compilation process now happens at runtime.

Our motivation to implement a reusable compiler for efficient array expressions is that although there are several excellent existing projects such and Theano [13] and NumExpr [14] in the Python community, neither library provides optimal implementations for execution of element-wise expressions on the CPU. This, together with the motive to implement array expressions in the Cython language, has led us to create a reusable, low-level expression compiler which does bring optimal or near-optimal performance, and which can be reused by all respective projects. Each of these projects uses its own internal representation, which is closely tied to how the library or language operates.

[2]The Low Level Virtual Machine, which for our purposes allows us to generate code at runtime in a platform-independent way.

Our goal is to create a shared compiler which is entirely agnostic of language syntax and runtime or ahead-of-time compilation strategies, allowing reuse across these projects and hopefully others in the future.

The rest of the dissertation is laid out as follows: In section 2 we describe the background theory for this thesis, in section 3 we describe the architecture of our compilers, in section 4 we cover the specializations we use. Section 5 describes optimizations that can be applied in various scenarios for ahead-of-time or runtime compilers. Section 6 describes the overall performance results, comparing with various Python libraries and commercial and open-source Fortran compilers. We finally conclude in section 7, followed by future work in section 8.

2 Background Theory

There are many languages and libraries that support array expressions. Many libraries try to retain data in the cache through elimination of temporaries arrays, for instance through the use of expression templates like Blitz++ [3]. A lot of research focusses on data-parallel execution of array expressions with implicit parallelism where communication is an important factor, such as High Performance Fortran or ZPL. We shall focus only on local array expressions, where effective memory and cache use and classic compiler optimizations such as vectorization and dominant factors for performance impact.

We will briefly review the syntax and semantics of array expressions supported by respective libraries. For simplicity, we will write the equivalent expression of Fortran's $a(:,:) = b(:,:) + c(:,:)$. We will focus on technologies present in the scientific Python community that deal with evaluation of array expressions.

2.1 NumPy

NumPy [5] [6] [7] is the de-facto Python library for dealing with multi-dimensional arrays. Although very powerful, arbitrary array expressions are often not supported efficiently, since each operation introduces a new temporary array. This is due to the way the expressions are evaluated, which is through overloads of operators at the Python level. The application of each operator introduces a new temporary array, unless lazy evaluation is used. To evaluate our array expression with NumPy, we write the following:

Listing 1: Simple NumPy Array Expression

```
a[:, :] = b[:, :] + c[:, :]
```

An ellipsis (...) may be used instead of colons, expanding to however many colons are omitted. For instance in 3-dimensional space we can write $a[..., 0]$ instead of $a[:,:,0]$ to select the first column in each dimension. Using either colons or ellipses indicates we are performing *array assignment* as opposed to creating a new array which we assign to a variable called a (this would be spelled $a = b[:,:] + c[:,:]$). Note also that, like Fortran, we may omit any colons from operands on the right hand side, i.e. $b + c$ is equivalent to $b[:,:] + c[:,:]$.

As mentioned in the introduction, array slicing always creates a new view on the data (see also section 2.4.2), instead of performing copies to contiguous memory, which is especially desirable for large data sets. This benefit also has downsides, which is that we can no longer assume data contiguity, and subsequent array computations may become less efficient.

NumPy has broadcasting rules [8] [6], which specify what happens when an array participating in an expression has extent 1 in a certain dimension. In this case the dimension is implicitly stretched (a SPREAD operation is implicitly applied) to match the common extent N. All participating operands must have this common extent N, or an extent of 1, otherwise an exception is raised.

Broadcasting dimensions can be conveniently inserted using *newaxis* indexing [15] [16] or reshaping. For instance given a vector v, we obtain a row vector using $v\_row = v[np.newaxis, :]$ and a column vector v^T using $v\_col = v[:, np.newaxis]$ (the *numpy* module is typically imported under the name *np*). Given a matrix M, we can now stretch v_row along the rows of M and v_col along the columns of M in an expression, e.g. we can compute the matrix-vector product using $np.sum(M * v\_row, axis = 1)$. The Python singleton $None$ may be substituted for $np.newaxis$, which is also the way one-sized dimensions are introduced in Cython. Broadcasting allows for memory-efficient operations since the broadcasting data does not have to be duplicated along broadcasting axes.

2.1.1 nditer

NumPy has an method called *nditer* [17] [18], which can be used to efficiently iterate over multiple array operands. It has both a Python and C API, allowing iteration in Python as well as in C or Cython. Among other features, it allows manual control over the preferred order, allows a manually written inner loop and can allow or disallow broadcasting and reductions. *nditer* allows us to evaluate our expression as follows:

Listing 2: NumPy's nditer Approach

```
for x, y, z in np.nditer([a, b, c], op_flags=["readwrite"]):
    x[...] = y + z
```

nditer sorts the array operands [18] in an agreeable element-wise C-like order (in order of descending strides), allowing efficient cache-wise traversal if there is an agreeable C-wise order. However, when there exists no agreeable order, for instance if some operands are Fortran ordered and some are C ordered, element-wise traversal with *nditer* may not be optimal, since it does not perform optimizations like tiling (see also 2.6). For our research we have not integrated *nditer* as of this writing, and our benchmarks with mixed-ordered operands are mostly tailored towards mixing C and Fortran ordered arrays. However, *nditer*'s sorting approach can be more effective since it handles arbitrary axes swaps (as long as the operands agree on the swapping permutation).

We do not arbitrarily permute our looping order, since we provide static specializations from our Cython compiler and we need to avoid combinatorial code explosion for runtime properties such as data ordering. Integrating *nditer*'s sorting mechanism will be very useful in the future to support arbitrary stride permutations efficiently as well as generate C-ordered specializations and omit the Fortran ordered ones (where the loop order is reversed). Reducing specializations is also important for runtime compilers, in order to reduce runtime compilation overhead and reuse cached versions of the code, even though they can arbitrarily permute loops to match a preferred order.

The performance issues of array expression evaluation present in NumPy gave rise to libraries like NumExpr [14] and Theano [13], discussed in the following sections.

2.2 NumExpr

To overcome the performance issues of array expressions present in NumPy, NumExpr was invented. NumExpr uses the first approach listed in section 1, by blocking up the execution, and furthermore allows threaded execution, providing good speedups over NumPy array expressions for expressions with more than two element-wise operations and for data-sets that justify the interpretation overhead [3].

In NumExpr, a user provides the expression as string input and the array operands as a dictionary. We implement our expression as follows:

Listing 3: A NumExpr implementation

```
numexpr.evaluate("b + c", {"b": b, "c": c}, out=a)
```

The *out* argument is optional and it determines the difference between array assignment and returning a newly allocated array. NumExpr compiles this string to bytecode, which it interprets and executes in blocks small enough to fit in the cache.

Although an excellent effort, NumExpr's approach does not provide optimal execution, since blocked execution means repeated execution of load and store instructions to load the data into registers and to write back the results. NumExpr uses nditer (section 2.1.1) to sort the strides into an agree-able C-like order, but this does not utilize tiling optimizations where appropriate.

2.3 Theano

Theano [19] [13], a mathematical expression compiler, on the other hand uses lazy evaluation with statically typed, symbolic, variables, compiling a mathematical expression graph at runtime when requested. This happens when the user requests a callable function from a given expression with input and output variables. Theano can do much more than just element-wise expressions and reductions, such as symbolic differentiation. Theano can also generate optimized CUDA kernels for most expressions, and optimize for speed and numerical stability. Among many, Theano performs speed optimizations such as eliminating common (array) sub-expressions, constant folding, element-wise fusion and recognizing special cases of multiplication and exponentiation which faster equivalents.

Theano has a strict typing system that is more powerful and static than the type system in Cython, since it considers broadcasting information a part of a variable's type.

[3]NumExpr first compiles the expression to byte-code, which is then interpreted.

For instance it has a matrix type [20] which is guaranteed to not be a row or column vector. This information is crucial to provide an optimal implementation for the expression, since Theano compiles the graph when the user requests a function to be build, when the actual data sets are still unknown. Specializing for broadcasting situations is generally impossible, since the number of possible broadcasting tuples is $\prod_{i=1}^{n} 2^{rank(i)}$, where $rank$ returns the dimensionality of array operand i [4], which grows infeasible very quickly.

To define our expression using Theano, we write the following:

Listing 4: A Theano Implementation

```
import theano.tensor
a, b, c = theano.tensor.dmatrices(['a', 'b', 'c'])
expr = theano.tensor.set_subtensor(a[:, :], b + c, inplace=True)
func = theano.function(inputs=[a, b, c], outputs=expr,
                       accept_inplace=True)
```

The first line imports the `theano` module and its sub-module `tensor`. The second line defines three symbolic array variables of dtype `double` (64-bit floating points). Line three defines the expression, which is a lazy computation returning an expression graph. We specify that we want this operation to operate in-place, i.e. we want to assign directly to the memory of matrix input operand `a`. To use this expression we build a function on line four, specifying three array input matrices of type double, and the output. We also have to specify that inplace operations are permitted.

At this point we have a function ready to evaluate our expression. Through various configuration options [21] [22], Theano decides the way expressions are evaluated. Options include the GPU, a C++-compiled kernel for the CPU and evaluation in Python. To evaluate our expression we write $func(x, y, x)$, where x, y and z are the actual matrices containing data.

2.4 Cython

Python is a highly dynamic programming language where everything is an object and the typing of done entirely at runtime. Due to its dynamic nature the language is very flexible, but it makes it also harder to write a fast interpreter [23], [24], [25]. As pointed out in [26] and [27], Python is typically not suitable for numerical kernels, since the interpreter overhead is too great, especially in the face of scalar computation. [26] mentions that it might be a good idea to move such code to Fortran, C or C++ and wrap it. Fortran code can easily be wrapped using for instance F2PY [28].

Cython [11] is a language superset of Python, which compiles to C or C++, and is often used in two domains. The first is to wrap native code - usually written in C, C++ or

[4]The number of specializations may be greatly reduced in various cases, for instance if a binary operator is commutative for operands of equal dimensionality. It should also be noted that not all permutations in the search space may not be useful to optimize for.

Fortran - and expose it to Python where needed. Another is to move computationally expensive parts from Python to Cython to get a speedup, often through type annotations. When fully typed, Cython generates code that is often without any calls into the Python interpreter, providing speed close to native C programs. Indeed, [27] describes how Cython can be used to write low-level numerical kernels directly in Cython itself. Since Cython is often used by the scientific Python community to speed up numerical codes, we implement array expressions in the compiler for this language.

To write performing numerical kernels in Cython, users need to type variables and buffers (multi-dimensional arrays), allowing efficient indexing operations with native access to un-boxed elements stored in multi-dimensional arrays. Cython also supports slicing of these views in many of the same ways NumPy allows, always creating a new view by adjusting the data pointer and the strides.

2.4.1 Typed Views on Memory

Typed views on memory allow the programmer to declare the type of a multi-dimensional array, by specifying the element type and the number of dimensions, analogous to how Fortran arrays are typed. Additional flags can specify contiguity in individual dimensions, or C or Fortran contiguity can be specified, allowing for some optimizations, as well as validation of data layout [29]. Below we declare a strided (which includes contiguous) array and slice it to get a view on "every other row":

Listing 5: Cython Example of a Typed Memory View and Slicing

```
cdef double[:, :] a = ...
print a[::2, :]
```

These multi-dimensional arrays are obtained through the buffer interface (PEP 3118, [30]), which specifies the C-level protocol for exposing multi-dimensional arrays to other code. Libraries such as NumPy expose their data through the buffer interface, allowing the consumer to access the data directly and more efficiently, without needing to go through any Python or NumPy layer. This is exactly what typed memoryviews are for, and it makes indexing orders of magnitude faster [27]. Typed memoryviews can be initialized by:

1. Assigning another, initialized, memory view to it, i.e. $a = b$.
2. By assigning an object exposing the Buffer Interface to it (see 2.4.2). E.g.

   Listing 6: PEP 3118 buffer object assignment

   ```
   cdef double[:, :] a = numpy.empty((10, 10),
                                     dtype=numpy.float64)
   ```

3. By casting a C pointer or array to a typed memoryview, inserting the appropriate shape information. In example:

Listing 7: Casting a pointer to a multi-dimensional array

```
cdef double[:, :] a = <double[:10, :10]> double_pointer
```

To understand how these views on memory operate, we describe the buffer interface below.

2.4.2 Buffer Interface

As mentioned, the buffer interface specifies the C-level protocol for exposing multi-dimensional arrays to other code. The interface works roughly as follows:

1. The consumer allocates a structure to hold certain information (on either the stack or the heap), and calls a function in the Python C API [31], passing in a pointer to the structure, and the object exposing the interface. It also passes in flags to specify what kind of buffer it is expecting, for instance a C-contiguous buffer. If information which needs to be exported in order to support consuming the array data is not requested, an exception is raised.
2. The function for obtaining buffers retrieves a C-level method exposed on the type structure of the object and calls it, passing in a pointer to the structure. This method fills out the required information, which we discuss below.
3. The consumer now has the information needed and has native access to the data stored in the multi-dimensional array.
4. When the consumer is finished with the buffer, it disposes of it in an analogous manner to item (2). As long as there are outstanding buffers, the array will not be garbage collected (in the CPython reference implementation, the buffer struct holds a reference).

The information needed to support arbitrarily strided views consists of a data pointer, a shape vector, a strides vector, the number of dimensions and whether the data is read-only. It also exposes information regarding the element type, namely the size and a format string, which is a string of characters that supports specification of structures, alignment properties, padding, function pointers, multi-dimensional arrays (nested within the multi-dimensional array that is being exposed), byte-order, and so forth.

We illustrate the workings of strides by example. Let $array$ be a two-dimensional C-contiguous array of doubles of shape $(10, 10)$. This results in the strides vector $(80, 8)$. Below we show what slicing does to the strides:

Listing 8: Strides Example

```
array[:, ::2]  # strides = (80, 16)
array[::2, :]  # strides = (160, 8)
array[:, ::-1] # strides = (80, -8)
array.T        # strides = (8, 80)
```

```
array[np.newaxis, :, :] # strides = (0, 80, 8)
```

The first two examples should be self-explanatory. On the third line we reverse the elements in the rows, which leads to a negative stride. This furthermore leads to the data pointer being shifted by $strides[dim] * shape[dim]$, effectively making it point to the last element of the first row. The transpose operation on the fourth line simply reverses the strides vector. Finally, in the last example we insert a new axis in the array, which leads to a stride of zero. This effects generic low-level loops containing index/stride multiplication to not move the data pointer for that dimension. This is useful when other operands have an extent greater than one, which means you're effectively broadcasting, without changing any code.

The buffer protocol additionally supports indirect C-style arrays, where a dimension can be indirect, which means the data in that dimension is to be dereferenced and an offset applied, allowing the following dimension to be sliced. Although this scheme is flexible, it does not allow one to remove consecutive indirect dimensions which are preceded by at least one indirect dimension, without changing the data or tracking additional information. We only support indexing for these types of arrays, and will not consider these type of arrays in our expression compiler. We justify this decision by pointing out not only are these styles not commonly used, it is also expensive to prove or disprove overlapping memory between the left-hand side and right hand sides of a given expression assignment, since we suddenly have to compare a number of pointers proportional to the data size with the pointers from the array on the left-hand side of the assignment.

2.5 Related Work

[32] discusses how consecutive array expressions using one or multiple operands and various intrinsic functions can be fused together to obtain a composite access function with a single pass over the data. It supports data movement operations such as shifts, as well as masked array operations, reshapes, transposes and spreads. We have not yet implemented this type of fusion, but it is very applicable to what we are doing. We also note the virtue of lazy evaluation in this context, which obviates the need for statements to appear in consecutive source code statements, and trivially bypasses control flow by building the graph at runtime in the order the operations appear.

2.6 Efficient Array Expression Evaluation

To provide good performance for array expressions that may be arbitrarily strided, we may face various sorts of situations which we want to handle efficiently. For this, we specify *order* to mean the vector that specifies for each stride in the strides vector the index into the sorted list of strides. For instance, the order corresponding to the strides $(40, 80, 8)$ would be $(1, 2, 0)$.

1. Array operands have agreeable data order. We say the data order of two operands is agreeable if the orders of the stride vectors are equivalent. The two simplest special cases are when both operands have ascending or both operands have descending order (discarding broadcasting rules for the moment). The solution for these special cases is simple, if the strides are descending, use C-ordered loops (in 2D, index the array from left to right with outer index i and inner index j). If the strides are ascending, use Fortran ordered loops (that is, either reverse the loops, or reverse the indices in the subscript). The more general case is encountered in for instance the case where you start with one of the simple special cases and you swap axes arbitrarily in three- or higher-dimensional space. This general case can be handled through a sorting mechanism like nditer, discussed in section 2.1.1.

2. Array operands do not have agreeable data order. We show how this scenario can be handled through tiling.

3. We have broadcasting operands and we want to avoid repeated computation. Consider for instance $matrix + row\_vector * row\_vector$, where we need to compute the squaring of the elements in row_vector only once. Optimizations for these sorts of scenarios are discussed in section 3.3 and 5.1.

To address the second case listed above, we apply the well-known tiling optimization [33]. This allows us to keep our data in the cache long enough to use all data items that fit in the cache line. For instance, consider the expression $a + transpose(a)$. If a is C contiguous, then $tranpose(a)$ is Fortran contiguous, and a given loop permutation that would be efficient in either C or Fortran order is no longer applicable, because data may be evicted from the cache before successive items may be used. For instance, assume we handle the expression using C-ordered loops. This means that we to fetch successive column items in our inner loop. For the C-contiguous operand these items fall in the same cache line (assuming the elements have a size smaller than the cache line), whereas for the Fortran contiguous operand it means they are far apart in memory. Before we can read the second value of the first column, the data may have already been evicted.

To accurately address the problem, we first review types of reuse presented in [34], for each of which we will determine its applicability for our work.

1. *Self-temporal reuse*. This refers to reuse of data in different iterations. Since we use each data item only once, unless we are broadcasting, this is of limited applicability. If we are broadcasting however, it may be fruitful to tile, allowing the data that is broadcast to stay in the L1 cache.

2. *self-spatial reuse*. This refers to accesses of data that fall in the same cache line. We exploit this type of reuse by making sure we reference data in an order with monotonically decreasing stride. This type of reuse can only be exploited when our strides are small enough to not exceed cache-line sizes.

3. *group-temporal and group-spatial reuse*. This refers to different code references referencing the same data and data in the same cache line, respectively. This

occurs when there is overlapping memory between operands. We do not provide any optimizations for these scenarios.

The performance effects of tiling depend heavily on the data size of the operands [35] [36] [37], and even slight variations in data sizes can lead to large variations in performance [35]. Research shows that self-interference and cross-interference must be kept at a minimum to get good performance. Pathological cases, where the contiguous or smallest strided dimension has a data size that is a power of 2 and divides the cache size, or is a multiple of the cache size, create great self-interference, since tiled rows (assuming row-major order) map to the same cache locations. If the row size divides the cache size, each kth row maps to the same cache line where k is the quotient.

Optimizations like array padding [38] can be used to avoid these pathological cases. However, since we do not have control over allocation and storage order over the operands themselves, we do not further consider this approach. This restriction also precludes blocked memory layouts with efficient mappings from indices to the data items [36], which implicitly turns normal indexing operations into tiled accesses.

To provide more stable performance with varying data sizes and strides and avoid intra-tile interference misses, it may be beneficial to employ a more complicated model than simple linear transformations on auto-tuned tiling parameters. In our approach we will focus mostly on minimizing self-interference, as done in [35], [37] and [39]. Most papers focus however on problems with temporal locality, such as matrix multiplication, where data items are reused. This is not the case for element-wise expressions, so we will focus on eliminating self-interference and thereby maximizing reuse for spatial locality.

3 Architecture

We will now describe the architecture of the low level specializing compiler and how it can be integrated in a runtime or compile-time environment. The compilation process can be roughly divided into the following steps:

1. Create or map the expression which is to be evaluated.
2. Create one or several specialization for the given expression, depending on needs.
3. Generate the code for each specialization.

The next section describes aspects of array expressions and data sets we want to optimize for.

3.1 Specialization Properties

To provide efficient runtime evaluation, we need to establish which aspects we want to optimize for.

We can divide the properties of the array operands into two categories, namely runtime and compile-time properties. A runtime compiler will generate only a single optimized specialization for a given array expression and array operands, since all properties of the operands are known at that point. An ahead-of-time compiler would, on the other hand, generate several specializations based on only the compile-time properties, generating code to select the best specialization based on the runtime properties.

Compile-time properties include:

1. `Dimensionality`.- If the dimensionality is greater than one, a tiled specialization can be created.
2. `Element type`. Based on the element type, our compiler can generate explicitly vectorized specializations.
3. `Data order or striding pattern`. This determines the permutation of the loops as well as whether we generate vectorized or strided accesses to the data.
4. `Contiguity`. In Cython, this information is optional.
5. `Broadcasting` rules for operands with lesser dimensionality. Operands with lesser dimensionality are prepended one-sized leading dimensions until their dimensionality matches the dimensionality of the highest-dimensional operand.
6. `Agreeability of data-order` between the operands (see section 2.6). At compile time, this can only be inferred from contiguity flags from the operand type declarations.

The properties for contiguity include full C or Fortran contiguity as well as contiguity in only a single dimension. The runtime properties are listed below:

1. `Data size`. Based on data size and user-control, we may or may not run the expression in multiple threads.
2. `Data alignment`. This determines the types of SIMD loads and stores we generate, i.e. can make the difference between aligned and unaligned loads and stores. We only generate code with unaligned load and store instructions.
3. `Agreeability of data-order` between the operands. We list this again since this is typically only known at runtime. This determines whether we will perform tiling or not. Depending further on contiguity and data size, it may be vectorized and SIMD transposed and it may perform square or non-square tiling.

The combinations of these given properties determine which code is generated. In our Cython implementation, an ahead-of-time compiler, only the runtime properties are variable. So we select the most important runtime properties, weighed by their expected impact on performance, and select a specialized version of the code, which assumes a value for the each property. At runtime, we test for all these values, and select the best specialization. A discussion of specializations can be found in section 4.

3.2 Ahead of Time Compilation

To use `minivect` [40] from an ahead-of-time compiler, we have to define a mapping from the AST from the original compiler to minivect, as well as a mapping from the original type system to the new one. Furthermore, we need to select the specializations depending on our compile-time properties, and generate the runtime code to select the right specialization. We shall refer to the ahead-of-time compiler as `Cython` and our array expression compiler as `minivect`, but these concepts should be universally applicable (in fact, the point of this separation in the first place is reuse among multiple projects).

3.2.1 AST Mapping

We allow full or partial AST mappings and full or partial type mappings in minivect. This may seem strange, and indeed it brings complications, but what it gives is reuse of code which is available on the original compiler, but not in the new one. For instance in Cython we want to support many element types (in the nomenclature of the NumPy and Python community, this is referred to as `dtype`), including Python objects and complex numbers. However, `minivect` does not, at the time of writing, support arithmetic on scalar of these types. In this case we can provide a partial mapping, which wraps a Cython AST in a minivect AST (in fact, this concept is applied recursively), allowing the minivect compiler to generate code for operations it would not normally support.

The steps of the mapping process are listed below:

1. During type analysis in Cython, recognize element-wise expressions and set a flag in the respective AST node. Among others, this includes unary and binary operation nodes, and calls to functions, in this case calling a function that takes a scalar argument where a multi-dimensional array is given.

2. After type analysis, rewrite the AST in several steps:

   (a) Find the outermost node marked as an element-wise expression, mark it as the root of the expression.

   (b) If the root node of the element-wise expression is not an assignment that should assign to the memory of another array, create a new node which should allocate a new multi-dimensional array variable backed up with memory from the heap. We also have to calculate the strides vector for this new contiguous memory.

   (c) Invoke a new AST transform (a visitor that can replace AST nodes), which purpose is to map nodes and types from Cython to minivect. This bottom-up process registers non-elemental sub-expressions as *operands* to the expression. We define *operands* below (3.2.1). Note that the reference of an array variable is not considered element-wise. Thus, arrays and sub-slices of arrays are mapped automatically to `minivect` AST *variable* nodes with array types. An example of a full mapping is depicted in Figure 1.

      This post-order replacement strategy is only intercepted in a pre-order fashion when an operation is not natively supported by minivect, in which case Cython provides the functionality.

      This case is handled through yet another transform, which operates in an analogous manner to the transform it was invoked from, except that instead of mapping nodes, it changes the type for element-wise nodes from arrays to scalars. This transform again registers any non-elementwise expression nodes as operands to the expression, which for array variables means the respective elements will be used instead of the array variable itself. Although this process could be invoked recursively, i.e. we could run our minivect mapper on the children of the node that cannot be mapped, this is unnecessary since Cython already implements all scalar operations that need to be supported, and since Cython uses the C code generation backend of minivect. So instead only the leaf nodes of a wrapped Cython AST are minivect AST nodes, since they reference the operands passed into the function. A partial AST mapping is depicted in Figure 2. The nodes coloured red are the nodes that bridge from Cython to minivect or vice-versa, and the pink nodes are Cython nodes.

   (d) Wrap the newly generated tree in a Cython node has the responsibility to invoke the minivect code generator. It has further responsibilities, such as

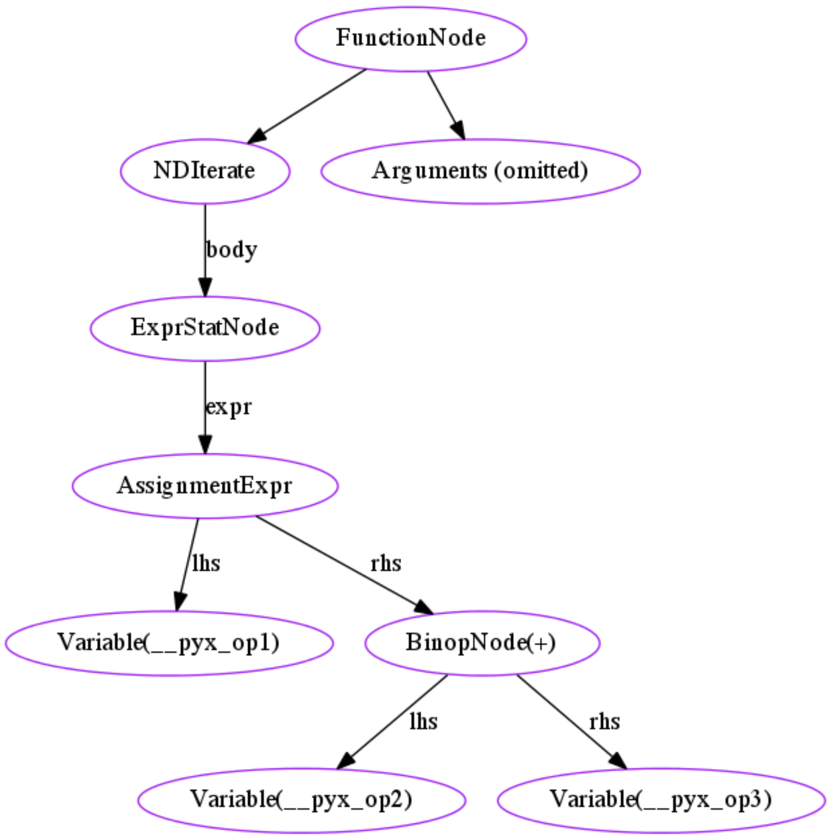

Figure 1: Full AST mapping of the expression $a[:,:] = b[:,:] + c[:,:]$

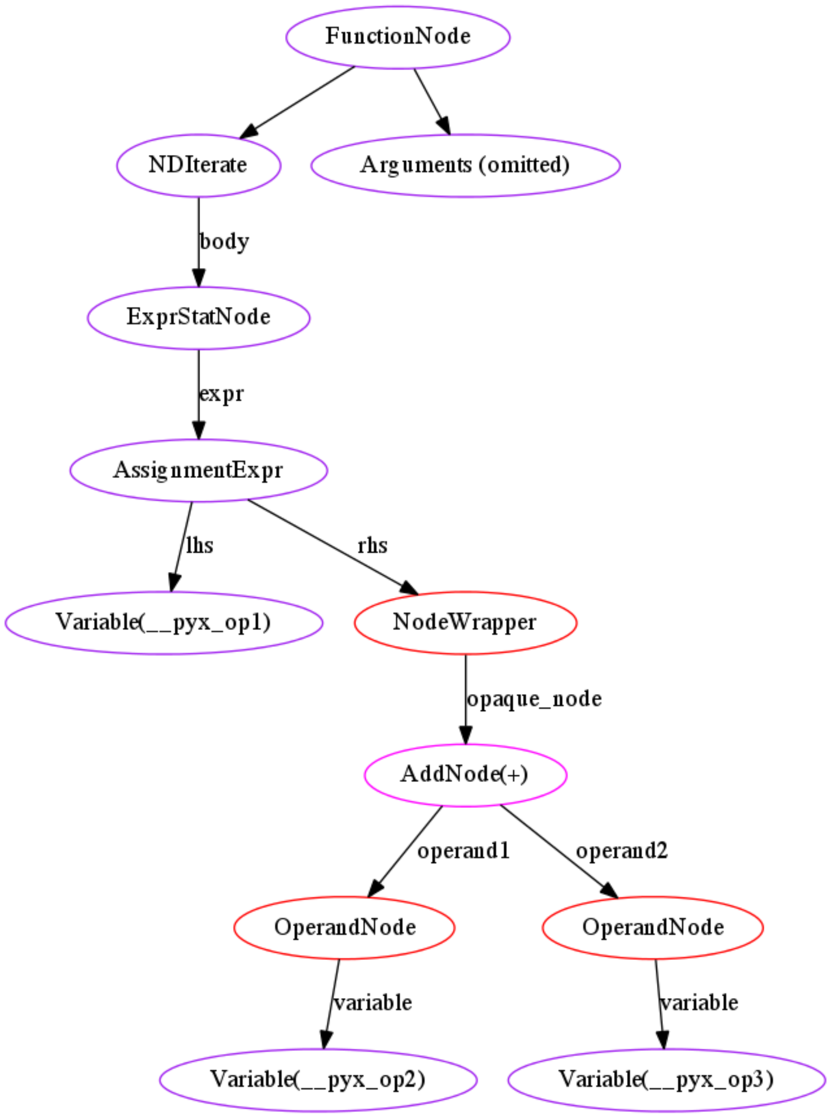

Figure 2: A partial AST mapping of the expression $a[:, :] = b[:, :] + c[:, :]$

checking for possible read-after-writes, to create a broadcast shape, and to select a specialization at runtime. We recognize that reuse of this functionality itself would be beneficial, and will integrate it in the future into minivect itself. However, for the purposes of this dissertation, we recognized that this was easier to implement, given that Cython has extensive support already for writing utilities, needed by the runtime, in C and in Cython itself.

The operands to the array expression are the arrays that partake in the expression as well as any scalars. The operands are the leafs of the expression tree which constitute the actual data. Consider the following expression:

Listing 9: Operands Example

```
a[...] = b[:, :] + f(c[:, :], x, 2)
```

where a, b and c are two-dimensional arrays and x and 2 are scalars, and f and elementwise function. Here a, b and c are array operands, x is a scalar operand and the constant 2 can be mapped directly at the AST level. `minivect` generates specializations as functions to which we pass in the following arguments:

- A pointer to the broadcast shape the length of *ndim*, the maximum of the number of dimensions for all operands.
- A pointer to the data and strides for each respective operand. Stride pointers are omitted when all operands are contiguous and there is no broadcasting involved. Note that a stride in a certain dimension is set to 0 when the operand is broadcasting in that dimension. This way, the data pointer is never advanced through a stride in this dimension. Note also how the dimensionality is fixed at compile time, so there is no need to pass this in as runtime information.
- Optionally, pointers to exception information, for instance for division or operations on objects. See section 3.3.1.
- All scalar non-constant operands. When the function f is called in the example above, the first argument varies with the data from array c, but x remains the same for every invocation in the expression.

3.2.2 AST Specialization

After the AST mapping process has completed, the minivect AST is specialized by a transforming specializer, which rewrites the AST in a form that matches efficient code for the given properties. All properties are *assumed* to be fixed at this point. The specializer rewrites the AST in terms of its own internal nodes. This process uses an AST builder, which is subclassable by a user of minivect, allowing the user to return different AST nodes depending on their needs.

The specializing process operates on a copy of the original expression tree, to allow subsequent specializers to reuse the expression tree and generate a different rewrite of the

AST. The specializing process operates on a minimal representation of the expression (such as shown before in figures 1 and 2), and expands it to a more complex mixture of the statements and expression that very closely match the actual generated code.

We only allow AST nodes that correspond to very elementary, low-level operations, and any high-level nodes are written in terms of these low level operations. This allows a code generator to only implement code generation for low-level nodes, which means the effort needed to support a new code generation backend is near-minimal.

We say near-minimal, since we still allow conditional nodes and *for loop* nodes, which we could rewrite in terms of jumps and labels. However, we refrain from doing so, since code generated in a high-level language may not support such operations, and if it does the generated code will be harder to read and harder for a compiler to analyse. Furthermore, it would not allow us to generate loop annotations in our C backend, such as the OpenMP parallel *do* directive. However, depending on the backend selected, we can optionally choose to rewrite these constructs to a low-level equivalent, which is currently useful for the LLVM backend, and may be useful in the future if we want to add other low-level backends.

This means our intermediate rewrite to a lower-level form of the AST is not fully agnostic of the selected back-end. For instance, for our C code generation backend we rewrite modulo operations on variables of floating point types to a call to the `fmod` function. However, modulo operations are supported natively in LLVM for floating point types through the `frem` instruction, which means such a transformation is unnecessary. We note that these kind of transformations are really part of the low-level code generation process, and should hence be factored out in low-level transforms that operate just before, or during, the actual code generation process. If the transformation is really only needed for a single low-level back-end, it may be equally effortless to special-case the transformation directly in the code generator itself.

3.3 Code Generation

The code generation process is very straightforward, since we have only very simple AST nodes, so we can directly translate a node to its equivalent target code. We generate C with optionally explicitly vectorized equivalents where this is determined beneficial, using compiler intrinsics for SSE2 as well as AVX. In our C backend this is realized through compiler intrinsics provided in the headers `xmmintrin.h` and `smmintrin.h`. We have also considered the VectorClass library [41], which uses C++ classes and inline methods to support SIMD programming. A desirable feature is support for libraries such as Intel® MKL, which can be enabled or disabled through preprocessor macros. However, GPL licensing as well as the need for users to ship header files in source distributions has thus far precluded its use. Explicitly vectorized versions are not yet implemented in our LLVM backend, although we believe this may be especially beneficial to allow optimal code for when data order is fixed at runtime and the specializations are too complicated for auto-vectorization (see section 4.1).

The code generation on the Cython side is somewhat more complicated, since minivect does not itself check for overlapping memory, and it does not allocate temporary arrays to avoid re-computation in the face of broadcasting (see also 5.1). Consider the example $a[:,:] = a[:,:] + f(b[:])$, where we broadcast the element-wise function call on vector b along the rows of matrix a (recall that lesser-dimensional operands get one-sized dimensions prepended). The problem here is that the computation is repeated for every row, whereas the result is only needed once. However, since the indices in vector b occur in the innermost loop, the index set is not an admissible prefix of the index set of a [32].

We say an ordered set B is an *admissible prefix* of ordered set A if the k elements of B form some permutation of the first k elements of A with $k \leq |A|$. This means we have to hoist out the entire computation on the vector and assign the result to a temporary array, in order to avoid this kind of re-computation.

The steps below are taken to generate code which allows overlapping memory and broadcasting. Let $ndim$ be the highest dimensionality of all array operands involved.

1. Ensure all scalar operands are simple (i.e., that they resolve to cheap C expressions or are stored in temporaries).
2. Allocate a shape vector of size $ndim$.
3. Broadcast all operands in this shape vector, raising an exception in case of a shape mismatch.
4. In case of array assignment, check for overlapping memory for all array operands on the right hand side of the assignment with the operand on the left hand of the assignment.
5. In the case of overlapping memory, check whether a read after write is possible. This is discussed in section 5.2.
6. If a read-after-write is possible, allocate a temporary to hold the result of the evaluation of the right-hand side. See section 5.2 for a discussion of possible optimizations.
7. Evaluate the expression. This is where we need to establish the runtime properties of our array operands (see section 3.1). We check the striding patterns for all operands, determine the data order and whether all operands agree on this order. We return flags according to the runtime properties in a generic function, which we branch on when selecting the specialization. The specialization which is most beneficial is selected and called through this mechanism.
8. If the right-hand side is a broadcasting assignment with regard to the left-hand side, or if we allocated a temporary to store the result in, copy the contents into the left-hand side.
9. Finally, generate any code to deallocate allocated resources.

We note that it would be beneficial to permute the axes of the operands in an agreeable element-wise order at runtime, for instance always in C or always in Fortran order. This way only one loop permutation is needed per specialization to support efficient element-wise traversal, whereas we now only support C and Fortran ordered loops. This can furthermore enhance performance when our axes are arbitrary permuted. This is the approach taken by NumPy's *nditer* [17]. Cython does however not have a hard dependency on NumPy, and we have not ported the implementation at this point, although this is planned in the future.

An overview of the code generated to select a specialization for the expression $a[:, :] = b[:, :] + c[:, :]$ may be found in Appendix A, Figure 31. The final copy operation of a potential array temporary is ommited for brevity, and mangled variables names are normalized.

3.3.1 Error Handling

Each node in minivect carries positional information from the exact appearance in the source code. Whenever an exception occurs during the execution of an array expression, this positional information can be set to indicate to the caller where the error occurred. Note that the exception and exception messages themselves are not always set by minivect itself, but for instance by a method called on an object as an overload of some arithmetic operator, or by an element-wise function. Exceptions are then propagated outwards while allowing exception to be processed before propagating them further. For instance, if we add the expression $(a + b) + c$ we first evaluate $a + b$, store this in a temporary and then evaluate $temp + c$. If the second addition goes wrong, we will still need to clean up our $temp$, which holds a reference to the result of the addition. So we generate code along the following lines:

Listing 10: Generated code to handle exceptions

```
    PyObject *temp1, temp2;
    int error_var;

    ...

    temp1 = PyNumber_Add(a, b);
    if (!temp1) goto error;
    temp2 = PyNumber_Add(temp1, c);
    if (!temp2) goto error;

    use temp2

    error_var = 0;
    goto cleanup;
error:
    error_var = 1;
cleanup:
    dispose of temp1 and temp2
    if (error_var)
        goto outer_error_label;
```

The *PyNumber_Add* function is a function in the Python C API which invokes the *__add__* method of objects and returns the result, or `NULL` in case of an error. At the outermost level, instead of jumping outwards we generate a return with an error indicator, which tells the caller an exception occurred.

3.4 Lazy Evaluation and Runtime Compilation

Our lazy evaluation front-end for runtime compilation is somewhat similar to Theano (but much less featureful). However, a key difference is that types and symbolic variables are not declared, but instead we wrap NumPy arrays in lazy objects. Subsequent operations on the lazy objects then build an expression graph. The expression graph is then compiled when we perform slice assignment to another array (when we assign directly to another array's memory):

Listing 11: Lazy Evaluation

```
1 a, b, c = lazy_array(x), lazy_array(y), lazy_array(z)
2 expr = sqrt(a**2 + b**2 + c**2)
3 a[...] = expr
```

Here line 2 is a lazy operation, and only on line 3 do we compile and evaluate the expression.

Note that our implementation is extremely minimal, lazy evaluation could be taken much further, and turn slice assignment into a lazy operation as well, while taking care to preserve original program correctness. This is however not relevant for the scope of this dissertation, since we use lazy evaluation to demonstrate the effectiveness of runtime compilation, especially in the face of missing type information such as broadcasting information, for which we cannot statically specialize. For a demonstration of the effectiveness in the case of broadcasting, we refer the reader to section 5.1.

3.5 Auto Tuning

We provide runtime auto-tuning to obtain parameters for square tiling and a parameter to cancel the overhead of parallel OpenMP sections, to reduce overhead for expressions with small data sizes. The overhead is measured in terms of the minimum number of floating point operations needed in order to break even. We do this through simple benchmarking and caching the results process wide. The auto-tuning process is triggered on the execution of the first array expression. We recognize that an increase in array operands means an increase in cache utilization, and apply a transformation to the tile size based on the number of operands. Currently, we divide the auto-tuned size for a single operand by the number of new operands multiplied by the *itemsize* (Python jargon for $sizeof(element\_type)$) of the new operands over the itemsize of the auto-tuned version. This is a rough metric, since we really want to account for the actual strides of the operands, which only equals *itemsize* when they are contiguous.

As section 2.6 mentions, static tiling bounds (even when auto-tuned at runtime!) cannot provide stable performance across different data sizes. So we explore an alternative to square tiling for k-way set associative caches tailored to element-wise expressions.

3.6 Testing

We have tested the Cython front-end using system tests, i.e. by writing the expressions, compiling them with Cython and a C compiler, by then executing them and matching against the expected output. We have tested our lazy evaluating front-end and the LLVM code generator back-end in a similar manner. We have also written unit tests, although to a limited extent. These were written using XPath by first converting the AST to an XML document. Finally, in the benchmark suite (see section 6) we test our implementations in the benchmark before executing and timing them, ensuring that they are performing the same operations.

4 Specializations

We support several specializations which we deem appropriate based on the given compile-time properties. A specialization is selected based on the runtime properties by an ahead-of-time compiler (in our case, Cython), or a specialization is instantiated and called directly by a runtime compiler (since all properties are fixed at compile time). In the following sections we will describe the specializations we generate, and as an example we show the generated code for the expression $a[:,:] = b[:,:] + c[:,:]$. Our compiler is parametric, which means we can switch on or off optimizations like strength reduction [5], or choose between tiling in 2 or n dimensions.

4.1 Contiguous

In the simplest case, all operands are contiguous and have the same data order. In this case we generate a single `for` loop that loops over the product of the shape, and which directly indexes the data pointers. This specialization is auto-vectorized by the compilers we used (GCC, ICC). We can also generate an explicitly vectorized specialization for SSE2 and AVX, with a fixup loop to account for extents that are not multiples of the vector size. Below we show the unvectorized (but auto-vectorized by the compilers we tried) specialized code:

Listing 12: Contiguous Unvectorized Specialization

```
static int
array_expression0contig(
            Py_ssize_t const *const CYTHON_RESTRICT shape,
            double *const CYTHON_RESTRICT op1_data,
            double const *const CYTHON_RESTRICT op2_data,
            double const *const CYTHON_RESTRICT op3_data,
            Py_ssize_t const omp_size)
{
    Py_ssize_t const temp0 = ((shape[0]) * (shape[1]));
    Py_ssize_t temp1;
    #ifdef _OPENMP
    #pragma omp parallel for if((temp0 > omp_size)) \
                             lastprivate(temp1)
    #endif
    for (temp1 = 0; temp1 < temp0; temp1++) {
        (op1_data[temp1]) = ((op2_data[temp1]) +
                             (op3_data[temp1]));
    }
    return 0;
}
```

[5] Strength reduction turns potentially more costly operations into cheaper operations, e.g. it may turn multiplication of an induction variable by a loop-invariant variable into repeated addition.

For readability purposes we disable name mangling. The `CYTHON_RESTRICT` macro resolves to the C99 *restrict* equivalent when supported by the compiler, and `Py_ssize_t` is simply an integral type large enough to support indexing of in the entire memory address space [42].

The function takes a pointer to the shape, which is the same for all operands (if it is not the same, but we are broadcasting, the contiguous specialization may not be selected). The second, third and fourth parameter are simply the data pointers of the arrays, and the last paramter is for the OpenMP *if* clause. This parameter is auto-tuned, to avoid parallel loop overhead for smaller data arrays. On our system, using GNU GCC 4.7, we need between 2^{14} and 2^{15} iterations to cancel this overhead for 4 threads.

The *lastprivate* declaration for the iteration variable is not necessary in this specialization, but it is needed for the vectorized equivalent.

Functions always return an integer, which indicates whether an error occurred. This is significant when dealing for instance with arrays of objects, or for operations like division which may raise exceptions.

The code below is the explicitly vectorized SSE equivalent of the above:

Listing 13: Contiguous Vectorized SSE Specialization

```
static int
array_expression0contig(
            Py_ssize_t const *const CYTHON_RESTRICT shape,
            double *const CYTHON_RESTRICT op1_data,
            double const *const CYTHON_RESTRICT op2_data,
            double const *const CYTHON_RESTRICT op3_data,
            Py_ssize_t const omp_size)
{
    Py_ssize_t const temp0 = ((shape[0]) * (shape[1]));
    Py_ssize_t temp1;
    #ifdef _OPENMP
    #pragma omp parallel for if((temp0 > omp_size)) \
                        lastprivate(temp1)
    #endif
    for (temp1 = 0; temp1 < (temp0 - 1); temp1 += 2) {
        __m128d xmm2 = _mm_loadu_pd((op2_data + temp1));
        __m128d xmm3 = _mm_loadu_pd((op3_data + temp1));
        _mm_storeu_pd((op1_data + temp1), _mm_add_pd(xmm2, xmm3));
    }
    if ((temp1 < temp0)) {
        (op1_data[temp1]) = ((op2_data[temp1]) +
                             (op3_data[temp1]));
    }
    return 0;
}
```

It has the same loop, except it now takes a step of 2, since each operation now operates on two doubles at the same time. In the loop we simply load the data in two SSE registers, add them together and store the result. The final *if* statement handles the case when we have an odd number of data elements, to add the final elements. If we had used floats instead of doubles, our compiler would have generated a loop to account for any remaining items instead of the branch. Our motivation for explicit vectorization is summarized below.

1. Auto-vectorization passes may be expensive for JIT compilers [43].
2. Auto-vectorizing compilers may not perform CPU dispatching properly [44].
3. We can generate a manual dispatching function, allowing users of our compiler to ship binaries that can take advantage of the latest SIMD instructions their processor supports [44] [10].
4. The auto-vectorizing compilers we tried did not auto-vectorize all of our specializations, such as the specialization with mixed strided and contiguous loads or our tiled specialization with contiguous operands.
5. As we shall see in section 6.1, even when a loop is auto-vectorized, the runtime may still decide to choose the unvectorized version.

4.2 Inner Dimension Contiguous

This specialization targets the case when the first or last dimension is contiguous for all operands. Without strength reduction enabled, this means we perform stride multiplication for the dimension preceding the contiguous dimension, and we generate a direct index for the innermost dimension. With strength reduction enabled, the default, we generate pointer additions in outer dimensions. This is discussed in detail in section 4.3.

This specialization is auto-vectorized by the compilers we tried, and we can additionally generate explicitly vectorized equivalents. The unvectorized generated code is shown below, when the arrays in the expression $a[:,:] = b[:,:] + c[:,:]$ are all contiguous in the first dimension.

In this code, we also need to pass in the stride pointers, since outer dimensions are strided. Strides are given in bytes, which means we need to cast our data pointer to a `char *` and add the index multiplied by the stride.

Listing 14: Inner Dimension Contiguous Specialization without Strength Reduction

```
static int
array_expression4inner_contig_fortran(
            Py_ssize_t const *const CYTHON_RESTRICT shape,
            double *const CYTHON_RESTRICT op1_data,
            Py_ssize_t const *const CYTHON_RESTRICT op1_strides,
            double const *const CYTHON_RESTRICT op2_data,
            Py_ssize_t const *const CYTHON_RESTRICT op2_strides,
            double const *const CYTHON_RESTRICT op3_data,
            Py_ssize_t const *const CYTHON_RESTRICT op3_strides,
                Py_ssize_t const omp_size)
{
    Py_ssize_t const temp0 = ((shape[0]) * (shape[1]));
    Py_ssize_t temp1;
    #ifdef _OPENMP
    #pragma omp parallel for if((temp0 > omp_size)) \
                             lastprivate(temp1)
    #endif
    for (temp1 = 0; temp1 < (shape[1]); temp1++) {
        double *CYTHON_RESTRICT temp2 = (
            (double *) (((char *) op1_data) +
                        (temp1 * (op1_strides[1]))));
        double const *CYTHON_RESTRICT temp3 = (
            (double *) (((char *) op2_data) +
                        (temp1 * (op2_strides[1]))));
        double const *CYTHON_RESTRICT temp4 = (
            (double *) (((char *) op3_data) +
                        (temp1 * (op3_strides[1]))));
        Py_ssize_t temp5;
        #ifdef __INTEL_COMPILER
        #pragma simd
        #endif
        for (temp5 = 0; temp5 < (shape[0]); temp5++) {
            (temp2[temp5]) = ((temp3[temp5]) + (temp4[temp5]));
        }
    }
    return 0;
}
```

4.3 Strided

The strided specialization is the least specialized version we generate, since it is fully generic. It multiplies each index with the stride and performs a lookup. We only specialize on approximate data order, that is if more operands are approximately row-major ordered than column-major ordered, we favour row-major over column-major. That is, in the column-major specialization we simply reverse the loops. Note again how a future `nditer`-like [17] sorting component will obviate this specialization. Generated code for this specialization can be found in Appendix A, Listing 32.

Our compiler generates code with explicit strength reduction of index calculation at a given loop level, and loop-invariant code motion of index calculation. This makes these optimizations independent of the target translator, in our case the C compiler or LLVM translator. This optimization can have a significant effect on performance, depending on the specialization and the target translator in question. We measure speedups up from 20% for up to 90%, depending on data sizes, specialization and compiler.

A conventional way to calculate indices is shown below.

Listing 15: No Hoisting or Strength Reduction

```
for (i = 0; i < (shape[0]); i++) {
    for (j = 0; j < (shape[1]); j++) {
        use *(double *) (((char *) p) + i * strides[0] +
                                        j * strides[1])
    }
}
```

If we assume a square $N * N$ matrix, this means we have $2N^2$ multiplications and $2N^2$ additions. Our compiler rewrites the above code to the following:

Listing 16: Hoisting and Strength Reduction

```
stride0 = strides[0] / sizeof(double)
stride1 = strides[1] / sizeof(double)
p = data_pointer
for (i = 0; i < (shape[0]); i++) {
    temp_pointer = p
    for (j = 0; j < (shape[1]); j++) {
        use *temp_pointer
        temp_pointer += stride1
    }
    p += stride0
}
```

This means that we now have only $N + N^2$ additions. We implement this optimization for all specializations, and this pass is the final compiler pass, since it needs to run after optimizations like hoisting in the face of broadcasting (section 5.1).

Note that our strength reduction is now incompatible with the parallel for loops we generate, since our variable p is initialized to the data pointer, i.e. the first element. This no longer corresponds to the right position according to the index. To remedy the situation, we simply generate two versions, which are conditional at C preprocessing time:

Listing 17: Fix Strength Reduction for Parallelism

```
stride0 = strides[0] / sizeof(double)
stride1 = strides[1] / sizeof(double)

#ifdef _OPENMP
p = data_pointer
#endif

#ifdef _OPENMP
#pragma omp parallel for private(p)
#endif
for (i = 0; i < (shape[0]); i++) {
    #ifdef _OPENMP
```

```
        p = data_pointer + i * stride0
        #endif

        temp_pointer = p
        for (j = 0; j < (shape[1]); j++) {
            use *temp_pointer
            temp_pointer += stride1
        }

        #ifndef _OPENMP
        p += stride0
        #endif
    }
```

Note that our LLVM code generation does not yet support parallel loops, and simply ignores children of an OpenMP conditional AST node.

4.4 Tiled

The tiled specialization is again specialized on row-major or column-major favouring, which means for column-major favouring we reverse the controlling loops, and we reverse the tiled loops. We can tile in 2 or in all dimensions. If we tile in two dimensions, any leading loops for row-major ordering, or any trailing loops for column-major ordering precede the controlling loops, with the outer loop order reversed for column-major ordering. We favour column-major over row-major when we have more column-major ordered operands than row-major ordered operands. The idea is that in the case of self-interference, we maximize spatial locality by minimizing the average stride in the innermost loop dimension. A tiled version of the generated code can be found in Appendix A, Figure 33, which is too verbose to display here.

4.4.1 Tiled with Mixed Contiguous Operands

If all operands are contiguous, but have differing data order, we select a tiled specialization. However, this specialization is not auto-vectorized by the compilers we tried (GCC 4.7 and ICC v12). We can vectorize this case manually by using SIMD loads and stores, and by transposing vector-sized squares in SIMD registers.

Vectorizing tiled code will mean we select a vector-sized square inside the tile to operate on. This square is strided in the first dimension for row-major operands, and the last for column-major operands. To match up items from to operands of differing data order would hence mean that for one operand the elements would be consecutive in memory, whereas for another the elements at those same index vectors would mean strided accesses in memory. Two SIMD loads hence correspond to different index vectors, so we match them up using a transpose. For the simple expression $a[:,:] = a[:,:] + b[:,:]$ we measure a 50% speedup using single precision and SSE (4 floats). We have not integrated this optimization into our compiler yet, but have adapted a generated tiled version to its SSE equivalent.

This vectorization approach is even more beneficial for multiple operands, e.g. consider a and b to be row-major ordered and c and d column-major ordered in the expression $a + b + c + d$. We can now evaluate $a + b$ and $c + d$ in SIMD registers, and transpose either of the temporary results for a final addition. We will follow the rule of transposing according to the data layout of the left-hand side of the expression, so that the result can be stored back into memory with a single SIMD store instruction. Depending on the expression and the respective data layouts of the operands, it may be fruitful to reorder the operands - if legal - in the expression, to minimize the number of transposes needed. For instance with our commutative operator $+$ we can arbitrarily reorder our operands, so we can group all row-majored and all column-majored operands together, evaluate them separately, and transpose only the final result of one of the sub-expressions.

The transpose operation can be described in three operations on each vector-sized row or column. Let N be the vector size and $v_{0..N-1}$ the ith row in the matrix at any given time, and let $stride$ be the stride in the non-contiguous tiled dimension, and j the innermost tiled index for the contiguous dimension:

1. Load N rows of size N

$$\left\|\begin{matrix} a_0 & a_1 & a_2 & a_3 \\ b_0 & b_1 & b_2 & b_3 \\ c_0 & c_1 & c_2 & c_3 \\ d_0 & d_1 & d_2 & d_3 \end{matrix}\right\|$$

Listing 18: Lower and Upper Unpacking

```
__m128 l0 = _mm_load_ps(p + j * stride);
__m128 l1 = _mm_load_ps(p + (j+1) * stride);
__m128 l2 = _mm_load_ps(p + (j+2) * stride);
__m128 l3 = _mm_load_ps(p + (j+3) * stride);
```

2. Unpack the lower and upper $N/2$ items of (v_0, v_1) and (v_1, v_2) into 4 new vectors of size N:

$$\left\|\begin{matrix} a_0 & b_0 & a_1 & b_1 \\ c_0 & d_0 & c_1 & d_1 \\ a_2 & b_2 & a_3 & b_3 \\ c_2 & d_2 & c_3 & d_3 \end{matrix}\right\|$$

Listing 19: Lower and Upper Unpacking

```
__m128 u1 = _mm_unpacklo_ps(l0, l1);
__m128 u2 = _mm_unpacklo_ps(l2, l3);
__m128 u3 = _mm_unpackhi_ps(l0, l1);
__m128 u4 = _mm_unpackhi_ps(l2, l3);
```

3. Perform the permutations to combine (v_0, v_1) and (v_2, v_3) into the final transposed rows:

$$\left\|\begin{matrix} a_0 & b_0 & c_0 & d_0 \\ a_1 & b_1 & c_1 & d_1 \\ a_2 & b_2 & c_2 & d_2 \\ a_3 & b_3 & c_3 & d_3 \end{matrix}\right\|$$

Listing 20: Final Shuffle

```
__m128 r1 = _mm_shuffle_ps(u0, u1, _MM_SHUFFLE(1, 0, 1, 0));
__m128 r2 = _mm_shuffle_ps(u0, u1, _MM_SHUFFLE(3, 2, 3, 2));
__m128 r3 = _mm_shuffle_ps(u2, u3, _MM_SHUFFLE(1, 0, 1, 0));
__m128 r4 = _mm_shuffle_ps(u2, u3, _MM_SHUFFLE(3, 2, 3, 2));
```

5 Optimizations

5.1 Broadcasting (SPREAD)

Like [32] we seek to optimize Fortran SPREAD operations, which we refer to as broadcasting. This operation is implicit in the user's code, unless some operands have lesser dimensionality, in which case leading broadcasting dimensions are prepended to match the dimensionality. Broadcasting information is optional in `minivect`, and it used only by the lazy evaluation front-end.

As mentioned in section 3.3 and described in [32], broadcasting operations are good candidates for optimizations, to avoid repeated computation of the same data. We keep track of a tuple that indicates for each dimension whether it will broadcast or not. These tuples are created directly from the given array data and contain the values $(shape_i == 1)$ for $0 < i < dimensionality$.

Depending on the admissibility of the indices used (see [32]), we either hoist out the expression to an outer loop and assign to a scalar, or we hoist out the sub-expression entirely. Consider the example where we have three operands, each of which broadcasts in a different dimension. We can conveniently construct these three vectors using `numpy.ogrid` [6]:

Listing 21: Three Broadcasting Vectors

```
i, j, k = np.ogrid[:N, :N, :N]
result[...] = i * j * k
```

In this example, i, j and k have data in dimensions 0, 1 and 2 respectively, and are broadcasting in the remaining dimensions. A static compiler, such as Cython, would in the absence of broadcasting information generate loops containing $4N^3$ index calculations [6] for 4 operands, $3N^3$ data load instructions, N^3 store instructions and $2N^3$ multiply instructions.
We can optimize this situation by hoisting out partial computations on the same data to outer loops. We can further hoist loop-invariant array variables to outer loops and store data in scalar variables. This optimization is implemented in our compiler in the presence of broadcasting information, fed in by our lazy evaluating component.

Appendix A, Figure 34 shows the optimized C code for the expression above and Appendix A, Figure 35 shows the LLVM IR (Intermediate Representation) generated by our compiler. The C code is generated by artificially feeding broadcasting information to our compiler, since broadcasting information is not part of the type system in Cython. The generated code now has only $N + N^2 + N^3$ load instructions and $N^2 + N^3$ multiply instructions, and index calculation overhead is down to $4N + 3N^2 + 2N^3$ operations (pointer additions). Note that we can lose even more pointer additions, since our code generator generates pointer additions for leading broadcasting dimensions, which

[6]Technically, $4(N + N^2 + N^3)$, see also section 4.3 on strength reduction

means it is adding zero to the strides pointer. We have not yet optimized this (this would bring the total number of additions to $2N + 2N^2 + 2N^3$, since we have one additions per operand that has data in that dimension, plus an addition for the strides pointer for the array on the left-hand side of the expression).

The performance results are shown in table 1, showing good speedups for lazy evaluation combined with just-in-time specialization. Trivially, the use of a more expensive operation, for instance the application of expensive binary element-wise functions (e.g. $f(g(i, j), k)$), would result in only a greater speedup, since the additional cost would weigh heavier.

Compared to NumPy we register a speedup of around $3x$ (discounting overhead for copying the result to a destination array). Note however that NumPy uses temporary array results, which means it will evaluate the same number of computations as our lazy evaluator (N^2 for $t = i * j$ and N^3 for $t * k$). The overhead comes from the temporary and the additional load and stores needed, as well as presumably from repeated index calculation.

We can write the Fortran equivalent to our expression in two ways, assuming variable a, b and c correspond to the vectors i, j and k, except in one-dimensional space:

1. Turn each vector into a cube and perform element-wise multiplication:

   Listing 22: Spreading Cubes

   ```
   result(:, :, :) = spread(spread(a, 1, N), 2, N) * &
                     spread(spread(b, 1, N), 3, N) * &
                     spread(spread(c, 2, N), 3, N)
   ```

2. Turn a and b into planes and multiply. Turn the result and c into cubes and multiply again:

   Listing 23: Spreading Planes and Cubes

   ```
   result(:, :, :) = spread(spread(a, 1, N) * &
                            spread(b, 2, N), 1, N) * &
                     spread(spread(c, 2, N), 3, N)
   ```

Note that both expressions have constant parameters for the repeating dimension, which means broadcasting information is known at compile time. The performance results are shown in the Table 1.

Note that we exclude runtime compilation overhead from our runtime compiler. Note furthermore that the SPREAD parameters are constants. If we insert runtime bounds, performance for Fortran compilers we tried is similar to GFortran's performance (Intel® Fortran did not seem to handle runtime bounds correctly, and Cray Fortran showed performance similar to GFortran).

On our system (see section 6), the runtime overhead is approximately 0.09s, which includes building the lazy expression, building and optimizing an AST and finally gen-

Table 1: Performance Results 3D Broadcasting

Lazy Evaluation + Hoisting + JIT Compilation	1600 MFlops
Cython Expression Evaluation	841 MFlops
NumPy	536 MFlops
GFortran Spreaded Cubes	172 MFLops
GFortran Spreaded Plane and Cube	294 MFlops
Intel® Fortran Spreaded Cubes	1182 MFLops
Intel® Fortran Spreaded Plane and Cube	1558 MFlops

erating and optimizing the LLVM IR and generated instructions. This overhead may be reduced in many ways, such as though caching generated implementations. For smaller data sets JIT compilation may not be worthwhile, and native evaluation (e.g. through NumPy) may be more feasible.

We have not yet implemented hoisting and array temporaries when index sets are not admissible prefixes (see section 3.3), and we are dealing with repeated computation. Consider again the expression mentioned in [6], reusing our i, j and k vectors: $result[:,:,:] = sqrt(i{*}{*}2 + j{*}{*}2 + k{*}{*}2)$. It only needs to perform $3N$ squaring operations, but our code generator will currently generate code with $N + N^2 + N^3$ squaring operations (squaring data in the first, second and third loop level respectively). Instead, we would like to generate the following:

Listing 24: Use Array Temporaries to avoid Re-computation

```
temp_vector_j = j ** 2
temp_vector_k = k ** 2
do t1 = 1, N
    temp_scalar_i = i(t1, 1, 1) ** 2
    do t2 = 1, N
        temp_scalar_ij = temp_scalar_i + temp_vector_j(1, t2, 1)
        do t3 = 1, N
            result(i, j, k) = temp_scalar_ij + &
                                  temp_vector_k(1, 1, t3)
        end do
    end do
end do
```

5.2 Eliminating Array Temporaries

Array expression semantics dictate that the right-hand side of the expression be evaluated independently of the left-hand side. This semantic is consistent with semantics of statements operating on scalars. For array expressions, these semantics prevent us from generating code which recklessly assigns to the memory of the left-hand side in two situations:

1. The right-hand side may have overlapping memory with the left-hand side, in which case we may write to locations before we read from them. Consider for instance the expression $a[1 : n] = a[0 : n - 1]$ with inclusive *start* and exclusive *stop* bounds. [45] discusses how the generated code (referred to as *scalarization*)

   Scalarization of $a[1 : n] = a[0 : n - 1]$

   ```
   for (i = 1; i < n; i++)
       a[i] = a[i - 1]
   ```

   has a loop-carried dependence. In this code, we write to a location $a[i]$ in iteration i, before reading it in iteration $i + 1$. This is incorrect, since we have to evaluate the right-hand side independently.

2. The operations on the individual data items have the potential to throw exceptions. In this case we cannot do anything but use an array temporary, since we cannot write partial results to the left-hand side, since the user-code is free to handle the exception and expect the program state to be consistent.

So we can only optimize for the first case. To provide correct semantics, we generate a runtime check between the left-hand side and each operand on the right hand side. The check itself is linear in the number of dimensions, and simply checks if there is a possible overlap between the *start* and *end* pointers of the two arrays. The pointers refer to the lowest and highest memory address that will be referenced, respectively. This means they have to account for negative strides in the *start* pointer and positive strides in the *end* pointer.

Based on the result of the check for overlapping memory, we perform a check for a read-after-write situation, which may violate the program semantics (the check has false positives). We then allocate a runtime array temporary for the left-hand side, and not the right-hand side, since we want to avoid multiple temporary arrays on the right-hand side. This means that in our static compiler we always have to generate a copy function which copies this temporary array to its final destination (the left-hand side) after executing the array expression, since we will not typically know at compile time whether this situation will be encountered [7].

Temporary arrays are clearly undesirable, since not only may dynamic memory allocation be expensive, it means an extra load and store for each data item. If the array is large enough, it may even mean the temporary cannot remain in the cache and has to be streamed in from memory twice, or worse, has to be swapped in and out from disk. In the worst case, the host may not be able to allocate enough memory at all, in which case we raise an exception.

To avoid array temporaries, we implement the aforementioned read-after-write checking function. This function has to test for the potential existence of a data dependence,

[7]Compile-time aliasing analysis could produce certain positives, but it cannot disprove overlaps between arbitrary operands which are not compile-time aliases.

and if present, determine its nature and implications. This is described in the following section (section 5.2.1).

5.2.1 Testing for Data Dependence

To test for data dependence between two arrays operands, we need to find out if there is a memory address that is referenced in both arrays. Consider for instance the array sub-slices $A[:: 2]$ and $A[1 :: 2]$, constituting respectively the even and odd elements of A. These elements obviously never overlap. To provide exact runtime proofs for data dependence or independence, we need to develop a generic way of verifying overlaps. Remember that our data elements are located at locations $data\_pointer + \sum_{i=0}^{ndim} stride_i * loop\_index_i$. The data pointer really constitutes an integer offset, so we need to check, for array operands A and B and shape bounds $0 \leq I_i < shape_{A_i}$ and $0 \leq J_i < shape_{B_i}$, whether

$$o_A + I_0 * s_{A_0} + I_1 * s_{A_1} + ... = o_B + J_0 * s_{B_0} + J_1 * s_{B_1} + ...$$

where o_A and o_B are the data pointer offsets in memory, and s_{A_i} and s_{B_i} are the strides of the operands. This means we need to check whether there is an integer solution within the given shape shape bounds. This is a linear Diophantine equation [46] [47], and testing for integer solutions is unfortunately an NP-complete problem [45].

In order to provide quick dependence tests, many inexact tests have been developed, including the GCD test [45] [48] [49], the BanerJee test [50], the ZIV, SIV and MIV tests [51], the omega test [52] [49], the I test [53] or the Delta test [45] [51], as well as exact tests for several common special cases. Most tests are inexact in that they only provide tentative dependence, but certain independence. [54], [45] and [51] evaluate the effectiveness of dependence analysis for certain benchmark suites. Note that we have to be conservative, in case we cannot prove independence, we must assume dependence, even though that may or may not actually be the case.

The GCD test is the simplest test, and it states that any linear Diophantine equation has a solution if and only if the greatest common divisor of the coefficients (the strides) divides the difference between the constant offsets. However, this test is not generally accurate, since it does not tell us whether solutions exist in our overlapping region, just that there is a solution in general. Often the greatest common divisor is 1, which means it divides any number, reducing its effectiveness. The test is useful however, since it is relatively cheap and simple, and proves independence when the greatest common divisor does not divide the difference.

In order to understand the problem and how to analyse data dependences beyond the capabilities of the GCD test, we need to understand data dependences. We paraphrase common definitions relating to data dependence below [45] [51]:

1. A *data dependence* from statement S_1 to S_2 occurs when there is a path from statement S_1 to statement S_2 at runtime where both statements use the same memory location and at least one statement writes to that memory location [45].

2. A *true dependence* or *flow dependence* is a data dependence where S_1 stores to and S_2 reads from the same memory location [45].

3. An *anti-dependence* is a data dependence where S_1 reads from and S_2 stores to the same memory location [45].

All our data dependences occur in iterations of generated loops. The distance between these iterations where the same memory location is read and written is called the dependence distance. The distance for each dimension then constitutes a *distance vector*. The sign of each respective distance determines the nature of the dependence. If the distance is zero, the sign is '=', if the distance is negative the sign is '>' and if positive '<' [55] [51]. Distance and direction vectors correspond to iteration instances, and multiple distinct vectors may be associated with a given number of nested loops.

Consider the following example $A(1 : N, 2 : N) = A(1 : N, 1 : N - 1)$. If we generate a loop nest where the first dimension is the outer loop, we obtain the distance vector $(1 - 1, 2 - 1) = (0, 1)$, which corresponds to direction vector $(=, <)$. In this case, we have a true dependence in the inner loop (the second dimension), since we would read a value in location j which we write to location $j + 1$. When we read iteration $j + 1$, we then read a value that was written in iteration $j - 1$. Hence, the generated code would be incorrect. If the situation would be reversed, we would have an anti-dependence, and the code would be correct, since we read from locations before writing to them.

To apply any of the dependence tests, we need to define our problem in terms of references to a single array, since that is what traditional dependence tests were developed to tackle. These tests allow a compiler to check dependences between compile-time aliases, and this is valid since the compiler knows in a language like Fortran that two different array variables have different regions of contiguous memory [8]. However, we have a situation where we must cater to ahead-of-time compilers performing static dependence analysis as well as runtime systems, which receive arrays as objects, and aliasing information is entirely unknown in this environment. So our problem is different from the traditional problem, and we are not guaranteed that different sets of indices resolve different elements in the different arrays, considering the memory may overlap in an arbitrary way. So to match the traditional problem and hence benefit from the solutions, we need to describe how the memory overlaps in terms of references to a single arrays using only linear transformations on the loop indices. This is covered in the next section.

5.2.2 Re-defining the Problem

In order to apply dependence tests in multiple dimensions, we re-define our problem in terms of references to a single array to match the problem and hence the solutions of traditional dependence testing mechanisms. Given two arbitrary arrays with overlapping rectangular memory regions [9], we have to define an N-dimensional loop-nest

[8]A compiler is however free to choose other approaches, as long as they provide the same semantics.

[9]Rectangular in N-dimensional space, we do not restrict ourselves to 2 dimensions.

with linear transformations on the loop index subscripts that define our data accesses in a single base array. We will address the situation where the user starts with a single array which is then sliced (repeatedly) to obtain a sub-slice of the array, containing an N-dimensional subset of the elements, since this is the most natural way leading to overlapping memory between two arrays (the left-hand side and an operand on the right-hand side).

To determine this single, mutually agreed upon, base array, we have several choices. We could for instance choose a rectangular region encompassing the regions of the two arrays, or we could choose the exact region of overlap. However, choosing the former will ensure that each sub-array will have an equal number of data references within the region (since they have equal shape), whereas this may not be true for the latter. If the number of data references is different, we cannot represent loop bounds and loop index transformations for both arrays. This is why we choose the former approach.

We have to solve two problems, namely to find a common array, and to find linear subscript transformations that preserves data dependences introduced by the loop order of the generated code.

For simplicity, we will assume the user is using NumPy arrays, and we follow the *base* attribute until the find the root. The *base* attribute provides a link to the array that was sliced to obtain the given sub-slice, i.e. `a[1:].base is a`. We only consider cases where the two roots of the two given sub-slices are equivalent. Since it is likely that arrays with overlap are sub-slices of some base array, we deem this adequate for our purposes.

We could alternatively compute this encompassing region, which is more general, but this is not trivial, since we have linearized values and not coordinates in N-dimensional space. We would need to compute the relative offsets in each dimension based on the strides, and construct and verify a solution.

The base array, whether actual or computed, needs to be verified in order to ensure that dependences resulting from the computed subscripts in our base array match the dependences between our original sub-slices and original subscripts, since that is what we generate our code for. Failure to do so may treat dependences as independent, which can lead to incorrect semantics.

Specifically, given array slices B and C with runtime strides and single integer data offsets, and a loop nest L_i of depth $0 \leq i < ndim$ with shape bounds $0 \leq I_i < shape_i$, we attempt to find linear subscript transformations for references B and C for all subscript dimensions on base array A, given the same loop nest. For example, given $B = A(:: 2, :: 2)$ and $C = A(1 :: 2, 1 :: 2)$, we want to find values for all variables in the subscripts in the loop nest shown below:

Listing 25: Example Loop Nest

```
do i = 1, shape0
    do j in 1, shape1
        A(offset0 + i * step0, offset1 + j * step1) = &
                A(offset2 + i * step2, offset3 + j * step3)
    end do
end do
```

In this case, the offsets correspond simply to the start slicing bounds, and the step variables to the stepping bounds. This is trivial to realise, but notice how, given the strides of A to be $(N, 1)$, we would receive runtime parameters 0 and $N + 1$ as data offsets.

Although it is easy to see that the new loop preserves the dependences, we have to consider that users may craft new slices from data manually, or based on for instance reshapes (which returns a view on the same data if possible). This means that our arrays B and C may have, as an example, individual rows which actually span multiple rows in A. In this case it is no longer possible to provide index transformations given the equivalent loop bounds, without out-of-bounds indexing, which violates the constraints for dependence testing. In this case, we will be conservative and assume dependence (and accept that we have to create a temporary array).

To verify a given base array A, we add the constaint that the elements of the sub-slice must be a subset of the elements of the base array. Furthermore, to preserve dependence, the dimensionality constaints must be preserved, i.e. distinct index sets must reference distinct data elements, and independent dimensions in the sub-slice must be independent in the base array, and vice versa. For instance, a single row in B must map to a single row in A, and not multiple (it is easy to craft situations which violate these constraints).

To verify our constraints, we perform the following checks:

1. The memory region of the sub-slices is contained fully within the memory region of the base array. This is a simple pointer check between the normalized (to account for positive and negative strides) start and end pointers of a sub-slice and the base array.

2. We must be able to find integer offsets in each dimension within the bounds of A that correctly reference the start of the sub-slice. Thus, we have again a linear Diophantine equation, and we have to prove that a solution exists within the bounds of A. We know that there is a solution if and only if the greatest common divisor of the coefficients divides the offset. However, we don't know if this satisfies the equation within our region.

   To solve the problem efficiently, we will only consider cases where we have only positive strides, or only negative strides for A (the sign of the strides of the sub-slices is immaterial). To ensure this we can negate the strides (and adjust the data pointer accordingly) in the dimensions in A for all non-matching dimensions. This means we also have to negate the same strides in the dimension in B and C and adjust the respective data pointers. This operation may reverse dependences in those dimensions, i.e. turn anti-dependences into true dependences and vice versa, but this transformation is valid since it matches the proposed reversed looping order in those dimensions.

We further mandate that the strides of A, B and C must define an unambiguous order, and that they specify exclusive partitions of data elements. With this we mean that in each dimension different constant indices will result in references to different data elements, e.g. $A[i, 0]$ and $A[i, 1]$ always refer to different elements (this must be satisfied for all dimensions). This can be verified by simply checking, in order of ascending strides, whether $|stride_i * shape_i| \leq |stride_{i+1}|$.

Then, in order of descending strides, we verify the following:

$$offset_{B_i} = \lfloor distance/stride_{A_i} \rfloor \text{ , with } 0 \leq offsetB_i < shape_{A_i}$$
$$distance = distance \bmod stride_{A_i}$$

and similarly for C. This procedure verifies that the data offset of sub-slices B and C can be expressed as a linear combination of the strides of A within the shape bounds of A, i.e. that the offsets for each subscript are integers. It also computes the offsets which we seek for all dimensions. If the distance after computation is non-zero, it is not a linear combination and it means we cannot apply our transformation.

This computation is valid because of the constraints imposed on the strides. That is, an absolute offset O must have index $\lfloor O/strides_{A_i} \rfloor$ in dimension i since $strides_{A_i} * (\lfloor O/strides_{A_i} \rfloor + 1) > O$, and any remaining strides are positive, which means we always get an answer bigger than the offset we are looking for (if instead all strides are negative, it means the inequality symbol is flipped and the computed offset will always be smaller than the offset we are looking for).

Our model does not currently handle broadcasting (strides of zero), since the division is undefined and strides of zero in the sub-slices would also violate the rule that different index sets must refer to different data items. However, our constaint is too strict, since steps of zero are allowed, so we can trivially adapt our procedure to generate zero step values for broadcasting dimensions in the sub-slices. We will not further consider broadcasting in our model.

3. Each stride in base array A for dimension i much divide the corresponding stride i in sub-slice B. This constaint ensures that the step is always an integer. Given that we also know that the offset is a linear combination of the strides of A, any multiplication of loop indices with integer steps will result in a subset selection of data elements from A.

4. All that is left at this point is to prove that the dimensions in the sub-slices have the same meaning as in the base array. For this, we only need to check whether

$0 \leq offset_{B_i} + stride_{B_i} * shape_{B_i} < stride_{A_i} * shape_{A_i}$ (analogously for C). This is equivalent to checking whether the final solution would constitute valid in-bound array accesses. This constraint now ensures that independence in a dimension in A implies independence in sub-slices B and C, and equivalently so for dependence.

The constaints above ensure that dependences are preserved, and that we operate within the constaints of the base array. We can now trivially compute the integer steps by dividing the strides of the sub-slices B and C by the strides in the base array A. We have now computed the offsets and steps, and are ready to start dependence testing.

5.2.3 Proving Data Independence

Building on the previous sections, we are ready to start dependence testing using traditional dependence testing techniques. To prove data independence, all we have to do is prove independence in any single dimension [51] [45]. This is easy to realise through a simple example. Consider a two-dimensional dependence test, where either the rows or the columns are independent. Now consider a true dependence for the rows. The order in which the rows are written is irrelevant, since the columns don't overlap (they are independent). Analogously, the true dependence for columns can be ignored if the rows are independent, since we are reading from and writing to different rows.

In a single dimension i, we now have the equation:

$$offset_{B_i} + x * step_{B_i} = offset_{C_i} + y * step_{C_i}$$

where the steps and offsets are computed as in the previous section.

To test for independence, we can try several tests (besides the GCD test in the dimension selected). The Single Index Variable (SIV) test may be applicable [51] [45], since each dimension contains a different loop index variable. The exact version may be used, which solves the linear Diophantine equation in the two variables [50] [45]. Depending on the coefficients (the steps we computed), special cases developed in [51] [45] allow one to apply an easy and quick test. For instance, if the coefficients are equal, we can calculate

$$distance = \frac{offset_{C_i} - offset_{B_i}}{step_i}$$

which only implies dependence if it is an integer and $|distance| \leq U - L$ [45] [51], where L is the lower bound and U is the upper bound on the index. In our case, we have to check whether $|distance| \leq shape_i$.

Several other tests can be applied in other cases, and they can compute the direction of the dependence in the dimension. For instance, consider the expression $A(1 :: 2) =$

$A(2 : N/2 + 1)$. The direction in this dimension starts of as '>', changes to '=' in the second iteration and to '<' in the third iteration. Fortunately, the BanerJee test can cope with these forms of dependence vectors [50]. We will not further describe well-known dependence tests, since they are better described in many of the aforementioned texts ([45] [51] [50] [52] [49] [53] [54] [48]).

If we can not prove independence, but instead know the direction vector (based on the dimensional offsets and steps), we may be able to circumvent potential read-after-write situations. We have implemented our method, but have not yet integrated it into the compiler. It implements only the strong SIV test as in the equation above, the GCD test for subscript pairs, and the Weak-Crossing SIV test [45] [51]. This way is already much more effective then the test for overlapping memory combined with the general GCD test to disprove solutions to the full Diophantine equation. For instance, in row-major storage, $A(:, : N/2)$ and $A(:, N/2 :)$ with exclusive end bounds results in memory addresses that indicate overlap, but the two regions are clearly exclusive. Where the GCD test may only get lucky in disprovig overlap, our test trivially captures all such situations through the SIV test, in this case in the second dimension, since it considers the bounds. The implementation of the dependence tester can be found in a public repository: [56].

[45] describes loop transformations that can eliminate the array temporary for several situations. This is covered in the next section.

5.2.4 Loop Reversal

The simplest way to avoid an array temporary is loop reversal, which means instead of writing to a location before reading it, we read from it before we write to it. More formally, this turns a true dependence into its anti-dependence [45]. As mentioned in [45], only a loop containing true dependences can be reversed, since reversal turns true dependences into anti-dependences and vice versa.

If we allow our compiler to eliminate temporaries for arbitrary runtime array slicing bounds and aliases using loop reversal, we would either need to create an additional specialization or generate a generic specialization with runtime loop bounds. Since we like to reduce the amount of generated code, we choose the latter option. We rewrite

Listing 26: Original Loop

```
for (i = 0; i < n; i++)
    ...
```

to

Listing 27: Loop After Reversal

```
i = start
n = (stop - start) / step
for (temp = 0; temp < n; temp++)
    ...
    i = i + step
```

To establish whether loop reversal should be applied, we need to do the following:

1. For each dimension find all arrays on the right-hand side that overlap with the array on the left-hand side.
2. If we have not proved independence using the techniques of the previous section, but have been able to establish dependence direction vectors, we need to examine whether we have a potential inconsistency. Given all dependence directions vectors of length $ndim$ for the operands with dependences, construct a dependence matrix of N vectors (one for each array with overlap) by $ndim$ direction symbols [45]. In this matrix the columns list all dependence directions from the different operands. The symbol '*' is used to indicate any direction.

   If all dependences in a column dimension are true dependences (if there are no anti-dependences), we can apply loop reversal in that dimension, turning true dependences into anti-dependences. This means we are left only with rows starting with the symbol $=$, since they may still have a dependence in subsequent dimensions. So we remove the first column and any rows not containing the direction symbol $=$ as the first symbol from the matrix, and move to the next column, repeating the process until all true dependences are eliminated. For a formal explanation and an algorithm we refer the reader to [45].

   This technique works well when we can select some outer loop with either only true dependences or only anti-dependences. If a loop contains both, we need to use other techniques such as input prefetching [45] or loop interchange with a loop that does satisfy these requirements [45]. If none of these optimizations can eliminate true dependences, we have to use an array temporary.

Note that since our compiler generates strength reduced code to calculate the pointer to the current element, reversing the loop will not have any effect since only the number of iterations are important and not the actual iteration values. To remedy this situation we could instead reverse the dimension for each array operand, i.e. adjust the data pointer by $shape[i] * strides[i]$ and negate $strides[i]$ in the $strides$ vector for each dimension i selected for reversal.

The listed optimizations have not yet been implemented in our compiler and are marked as future work.

5.3 Effective Cache Utilization

As section 2.6 mentions, spatial locality is the only form of cache reuse we will exploit, since every data item is referenced only once, unless an operand is being broadcast or unless some operands have overlapping memory.

Most research focusses on problems with temporal locality, like matrix multiplication ([35], [37] and [39]). Our problem is somewhat different in that we have an arbitrary number of operands, and we use each data item only once (see also section 2.6). Blindly fusing all operands together may hamper instead of benefit performance, and a blocked approach may or may not be beneficial (the first approach described in section 1). For clarity we re-iterate our meaning of blocked evaluation. With blocked evaluation we mean that we process the expression in successive chunks, applying each part of the expression on a small chunk that fits in the cache before moving on to the next chunk. This is exactly what NumExpr [14] does. This approach is shown in pseudo-code below, and is copied verbatim from the NumExpr website.

Listing 28: NumExpr's Blocked Evaluation

```
for i in xrange(0, len(a), 1024):
    r0 = a[i:i+256]
    r1 = b[i:i+256]
    multiply(r0, 2, r2)
    multiply(r1, 3, r3)
    add(r2, r3, r2)
    c[i:i+256] = r2
```

We have to provide stable performance in a variety of situations, namely:

1. We need to select an agreeable tile size for our operands that is a trade-off between minimizing self- and cross-interference and minimizing tiling overhead. To minimize the former, we may need small tile sizes, but to minimize the latter, we need larger tile sizes.
2. We have a higher number of array operands. We need to determine if and how to split up the operands for blocked evaluation to minimize the execution time.

The next two sections shall cover these respective cases.

5.3.1 Tiling

To effectively use spatial locality we use tiling when the arrays in an array expression have different data orders. Literature shows that self-interference is a great contributor to performance reduction, due to the regularity of strides and the wrap-around nature of the cache [38] [35] [37] [33]. There are many choices to avoid self-interference, such as square tiles as shown in [39], but as proposed in [35], rectangular tiles may use the cache capacity more effectively while still avoiding self-interference. However, it must

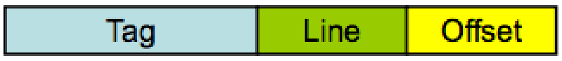

Figure 3: Address mapping for direct-mapped caches. Figure taken from [57].

also be noted that wide rectangles are subject to TLB (Translation Lookaside Buffer) thrashing [38] [35].

To understand the interference problem and how to address it, we summarize important observations from literature below. When we will talk about the *effective cache size* we will mean the size of a single cache set, i.e. $CS_E = cache\_size/associativity$ [10].

1. Two data elements fall in the same cache location [11] (but possibly in different cache sets) when their addresses are congruent modulo the effective cache size, i.e. whether $\&a[i, j] \equiv \&a[i', j'] \pmod{CS_E}$ [39]. This observation is valid since caches use the layout shown in Figure 3 [57].

   The least significant bits of the memory address indicate the *offset* into the cache line, and the bits before that indicate the *cache line* the word falls in. The most significant bits are used to compare with the higher bits of memory addresses, since we have many more memory locations than cache locations. Caches use the least significant bits for the displacement in the cache, since nearby memory addresses should map to different cache locations.

   Set-associative caches use the same scheme, except that the *line* now maps to a *set* [57], and the number of bits that indicate the displacement of the set depends on the effective cache size instead of the total cache size.

   So we can easily compute where an arbitrary memory address maps in the cache. The congruence relation above then follows from the cylic properties of lower bits of the memory addresses. This observation is important since it allows us to observe that $\&a[i, j] \equiv \&a[i', j'] \pmod{CS_E}$ implies $\&a[i + a, j + b] \equiv \&a[i' + a, j' + b] \pmod{CS_E}$ where CS_E is the effective cache size [39]. In other words, the amount of self-interference (the interference during the execution of a single tile of that tile with itself) depends only on the strides and the tile sizes involved.

   Since indices $a[i, j]$ resolve to $i * stride_0 + j$, the relation above is again a linear Diophantine equation, since $a \equiv b \pmod{n}$ is equivalent to $a = b + k * n$ with $k \in Z$ [58]. This means we now have the equation $offset_A + i * stride_{A_0} + j = offset_B + i' * stride_{B_0} + j' + k * cache\_size$. [59] describes a general framework of accurate cache miss equations, which has many applications, including computing cache interference in loop nests to help compilers and linkers compute array offsets and padding sizes. The equations in [59] are more accurate than the

[10] How these parameters are obtained, such as through a configuration file, the CPUID instruction or other platform dependent ways is not of interest for the purposes of this dissertation.

[11] To measure cache interference, we need to know how often cache *lines* will map to the same location in a single tile. In our examples we will assume data is aligned on cache-line boundaries.

equation above, since they account for cache line boundaries (as well as loop boundaries, of course).

2. We can use the remainders from the Euclidean algorithm, which computes the greatest common divisor between two numbers, as a good starting point to compute the column tiling size, as demonstrated in [35]. This observation is best demonstrated in a simple example. We will use the same parameters as in [35].

   Example: In this example we disregard the element and cache line sizes for simplicity. Assume the effective cache size CS_E is 1024 and a square contiguous matrix of $(M, N) = (200, 200)$, stored in row major order. We can fit 5 consecutive rows in the cache set, which means our tiling parameters constitute the two-tuple (5, 200). To fit one more row of size 200 will cause self-interference. If we look at the remainder of $CS_E \bmod N = 1024 \bmod 200$ we obtain 24. This remainder is useful since it tells us we can occupy 24 additional elements without incurring self-interference. It furthermore tells us we can keep occupying the cache with rows, since they are offset by -24 (or 200 - 24). After having fit 10 rows, the next 5 rows will be offset by 2 * -24. Since the gap grows linearly, we can simply divide the stride (200) by the gap to see how many times we can repeat this process. We obtain the number of columns by computing $ncols = CS_E \bmod N$, and the number of rows using $nrows = (CS_E/N) * (N/ncols) + r_0/ncols$. We need the final addition to account for the occupation in the last cache location (the first remainder, r_0), which is not counted as part of the rows. We need to divide this by the remainder we are using from the Euclidean algorithm. In our example, this yields the values (41, 24). We cannot simplify the last equation since we need the truncation.

   To see the amount of interference with minimal variations to these computed tiling sizes, we write an accurate simulation by generating the set of occupied cache locations and comparing with the actual amount of needed data.

   Listing 29: Cache Interference Simulation

```
def occupy_cache(m, n, N, CS):
    result = set()
    location = 0
    for i in range(m):
        for j in range(m, m + n):
            datum = (i * N + j) % CS
            result.add(datum)

    return len(result), m * n
```

   Function *occupy_cache* returns the a two-tuple $(cached\_items, total\_items)$, where $cached\_items$ is the total number of items that can be in the cache at any given time during the execution of the tile. The results for $occupy\_cache(41, 24, 200, 1024)$ is $(984, 984)$. However, any addition to a tiling parameter will create interference, for instance using tiling parameters $(41, 25)$ returns $(989, 1025)$, about 3.6% interference. Bringing the tiling column size (the number of rows) all the way down to 25 brings the interference down to 3.3%. If we now slightly adjust our row size N to 202, the square tile $(25, 25)$ results in the tuple $(405, 625)$, a self-interference of 54%. This shows us that choosing (small) square tiles without regard to the data size is not enough and cannot provide stable performance across even slight variations in data size (or more generally, strides).

To find optimal tiling sizes for an arbitrary number of operands we could take several approaches, given these findings above. We could construct reuse vectors, which tell the cache miss equation framework in [59] for a given iteration point which directions contain spatial or temporal reuse. We could then create equations for self- and cross-interference, and try to maximize the tiling area while minimizing the number of solutions. We do not further explore this option in this work.

We could alternatively adapt the algorithm from [35], which considers successive remainders from the Euclidean algorithm (with fall-backs in case the remainders are zero or otherwise not suitable, e.g. select entire rows), and maximizes the tiling size while minimizing the expected cross-interference rate. This algorithm always makes sure the total working size does not exceed the cache capacity. However, our problem is somewhat harder, since we have to deal with an arbitrary number of operands, and with arbitrary strides in the non-contiguous dimension (if the inner tiling dimension has a stride for a given array that exceeds the cache line size, there can be no spatial reuse for that array).

If we assume, for simplicity, a number of operands smaller than the cache associativity, we can simply aim to maximize the tiling area while avoiding self-interference. If we generate for each array a tile tuple according to the algorithm presented in [35], we can simply intersect the areas by taking the minimum value in each dimension. Generating multiple candidate tile sizes for each array and finding the maximum area may lead to a better tile size, but trying each combination may be intractable. However, it may be sufficient to try, for each item in the first candidate set, to find the best candidate in successive candidate sets. If we then have more operands than cache associativity, we can simply use a heuristic to weight final candidate tile sizes based on their area and the expected cross-interference (based for instance on some probability function).

In our implementation, which is not integrated into our compiler, we generate the tile size $(128, 5)$ for two square matrices for 200 by 200 elements in a 1024 element set-associative cache from the sets $[(5, 200), (41, 24), (128, 8)]$ and $[(200, 5), (24, 41), (8, 128)]$, resulting in 62.5% cache occupation. If we give the algorithm arrays with strides $(280, 1)$ and $(282, 1)$ and the respective transposes, it generates the sets below

Listing 30: Tiling Candidate Sets

```
[(3, 280), (4, 184), (7, 96), (11, 88), (128, 8)]
[(280, 3), (184, 4), (96, 7), (88, 11), (8, 128)]
[(3, 282), (4, 178), (7, 104), (11, 74), (29, 30),
 (69, 14), (512, 2)]
```

```
[(282, 3), (178, 4), (104, 7), (74, 11), (30, 29),
 (14, 69), (2, 512)]
```

from which it intersects the areas to find $(69, 8)$ as the final tile size, which has a cache utilization of approximately 54%. Our algorithm is depicted in Appendix A, Figure 36.

5.3.2 Blocking

In this section we evaluate the effectiveness of blocking as discussed in section 5.3. It must be noted that since we provide a compiler, and not an interpreter like NumExpr [14], blocked evaluation means we need to generate different code for each subexpression selected for blocked evaluation (except for contrived cases). Consider the expression $a[:,:] = b[:,:] + c[:,:] + d[:,:] + e[:,:] + f[:,:]$. If we want to block up this execution, we could decide do evaluate - in chunks - $a[:,:] = b[:,:] + c[:,:] + d[:,:]$ followed by $a[:,:] = a[:,:] + e[:,:] + f[:,:]$. In this case we can generate a generic specialization that takes a as the left hand side and 3 other operands on the right hand side. In fact, it can reorder the operands in the expression, considering that the operator used is commutative.

Blocking may be useful when we have a high number of array operands, and they may have pathological alignment properties. For instance, if all arrays are aligned on the the effective cache size, all references will fall in the same set location. In this case it may be beneficial to perform blocking where a arrays are executed at a time.

This case may seem contrived, but note that a lazy evaluation runtime may create large expression graphs with many expressions and different operands. In our benchmarks we align a number of arrays on the cache set-size boundary of our CPU's cache. We register a 10% speedup by executing the expression in a blocked fashion with 8 arrays, and around 20% to 30% with 16 arrays.

6 Results and Analysis

To evaluate the general performance of our library we have written a benchmark suite that evaluates certain array expressions using our Cython front-end. In these benchmarks we will not target special optimizations such as broadcasting as discussed in section 5.1 or elimination of temporary arrays as discussed in section 5.2, since those are better discussed in their respective sections. We evaluate our approach with auto-tuned tiling parameters and threshold parameters to cancel overhead of OpenMP parallel loops for small data sizes. All benchmarks are executed on an Intel® Core™ i5-760 2.8 GHz quad-core processor.

We evaluate each benchmark with NumPy, NumExpr, Theano (with thanks to Frédéric Bastien) and Fortran. To compare with Fortran we use the GNU Fortran 4.7 compiler for the Fortran benchmarks, which we compare with our Cython benchmark compiled with GCC 4.7. We also compare array expressions compiled with Intel® Fortran to array expressions in Cython compiled with the Intel® C compiler. We shall separate these results in different graphs for clarity.

Note that although we support any element type and any arithmetic operator as well as element-wise functions, we will evaluate simple expressions on single precision floating points, since it is important to evaluate performance with the computationally cheapest operators possible in order to highlight the inefficiency of the execution strategies.

6.1 Contiguous

Our first benchmark is a simple expression $a[...] = a + b$ with Fortran contiguous two-dimensional operands. The results are shown in Figure 4 and Figure 6. It would appear that GFortran's implementation performs nearly twice as fast for the smaller data sets as compared to the Cython implementation compiled with GCC. However, compilation with the Intel® C compiler results in approximately equivalent performance to the Fortran version.

It seems that GCC generates a test that checks whether our pointers are actually `restrict`, i.e. whether they actually constitute exclusive memory regions (we are uncertain whether this involves a pointer comparison, or whether it takes the actual loop bounds into account). Since we use a on the left-hand as well as the right-hand side, the code will execute a generated non-vectorized version of the code. This results in the performance shown in Figure 4. Fortunately, our compiler can generate manually vectorized code, for which the performance is shown in Figure 5, which shows our compiler has roughly the same performance as GFortran for this expression and these data layouts.

Note how we usually merge equivalent compile-time operands into a single argument. E.g. $a[:,:] + a[:,:]$ should result in only a single data and strides pointer being passed in to the function. The code that detects this didn't however recognize $a[...]$ and a as

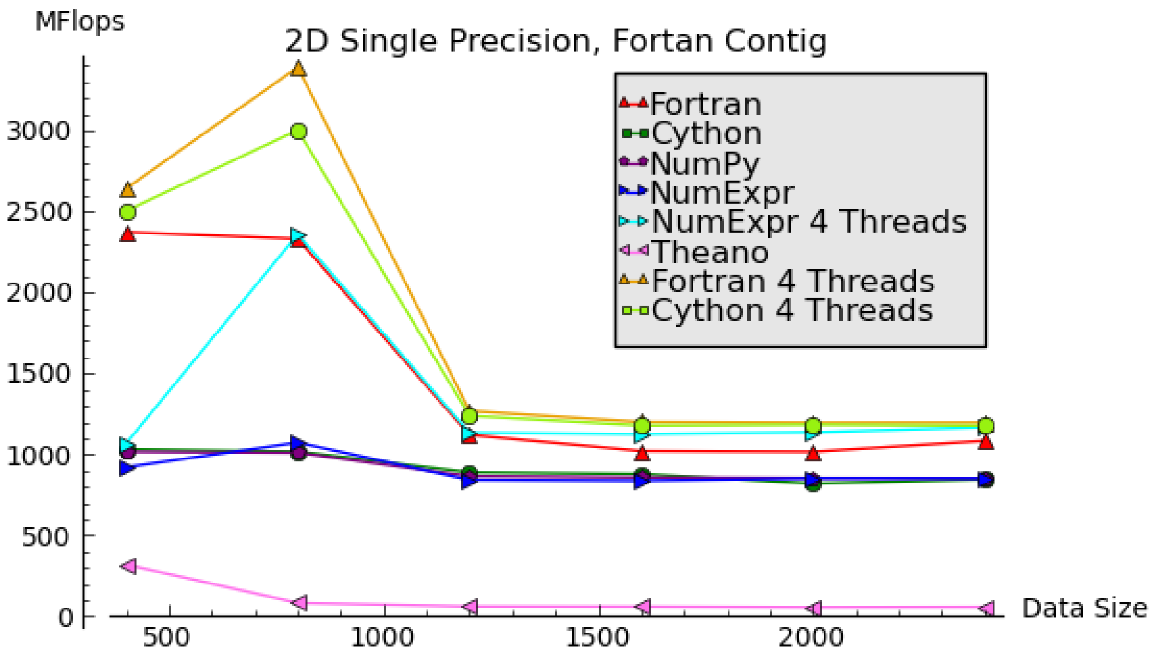

Figure 4: Benchmark with Fortran-contiguous operands, GCC/GFortran 4.7, unvectorized.

the same operand. Fortunately, this is entirely trivial to implement (but we have not yet done so, since these results are interesting).

Theano performs bad here, since it doesn't optimize for Fortran order or contiguity. Instead, it assumes C-ordered operands, since Theano itself always allocates memory in this order. With user input in Fortran order this means it is traversing the memory in the worst fashion possible, since the outer loop iterates over the contiguous dimension and the inner loop over the strided dimension. However, integration of our project as a backend for Theano has been discussed, and enthusiasm from the community suggests this will likely happen in the near future.

Note how Intel® Fortran with parallel array expressions, using the OpenMP *workshare* directive, is consistently slower than the sequential equivalent, whereas we do see a good speedup for the parallel Cython and GFortran versions. We can only conclude that Intel® Fortran does not either support parallel expressions, or has a bad implementation.

6.2 Strided

Our strided benchmark executes a simple expression on strided operands. Instead of varying the data size we now vary the stride, and grow the data size accordingly (starting with a data size of 400 by 400). The graphs in figures 7 and 8 depict the performance using GNU and Intel® compilers respectively. The performance characteristics are mostly the same for all libraries with larger strides, since the problem becomes

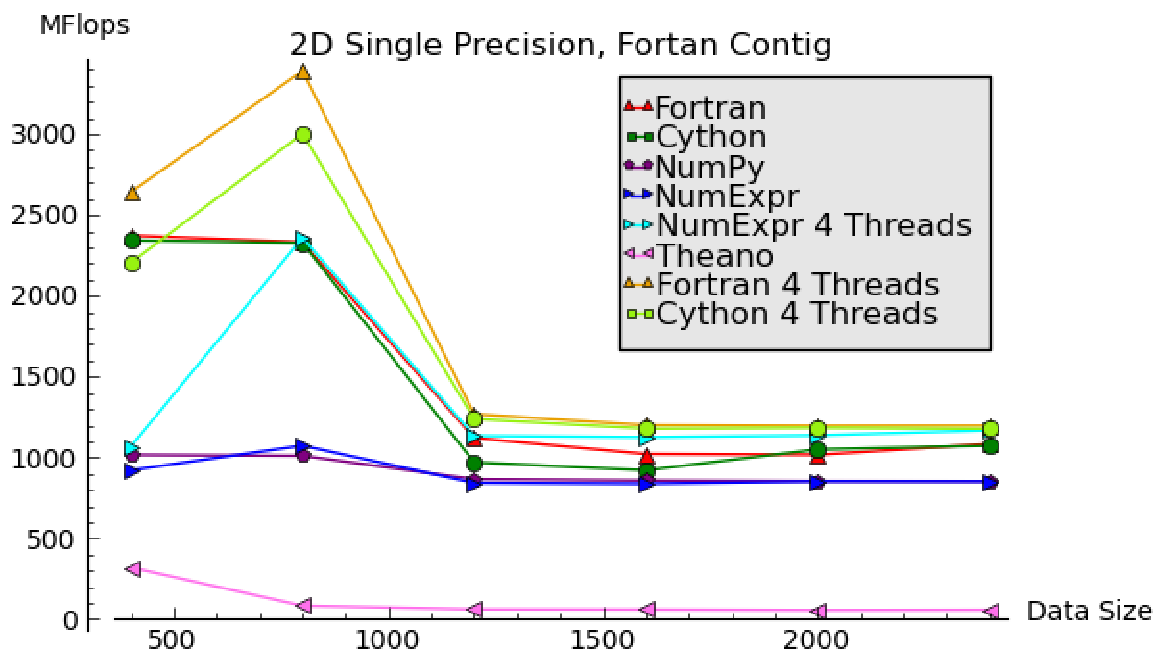

Figure 5: Benchmark with Fortran-contiguous operands, GCC/GFortran 4.7. This benchmark was explicitly vectorized by our compiler.

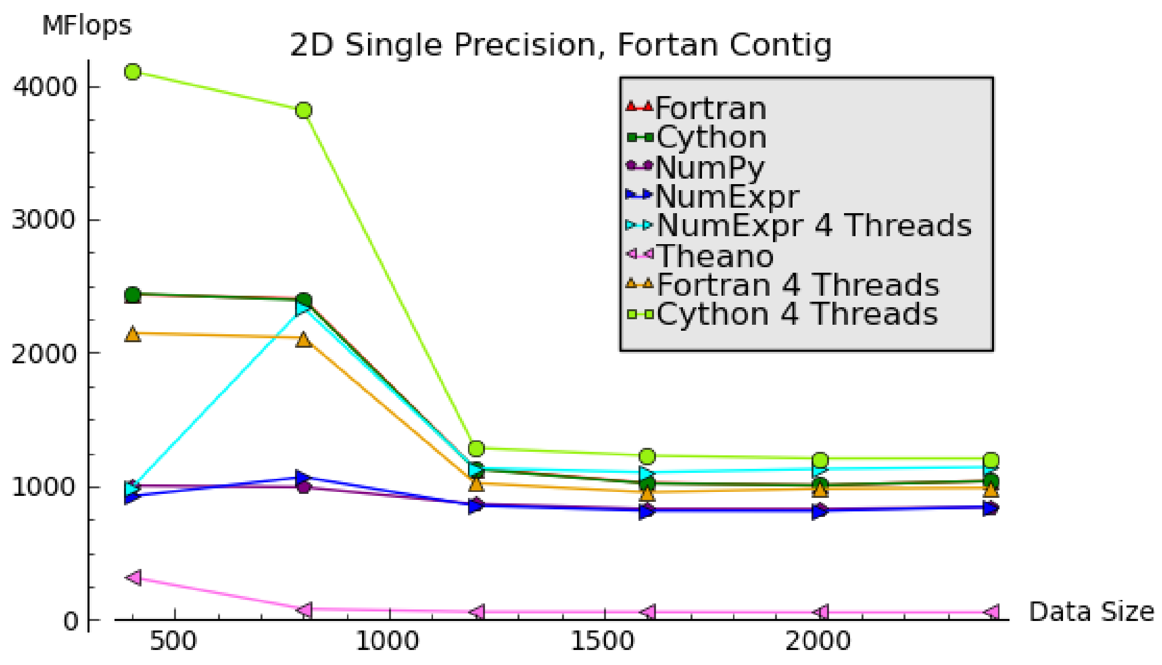

Figure 6: Benchmark with Fortran-contiguous operands, Intel® C/Fortran version 12.1.5.

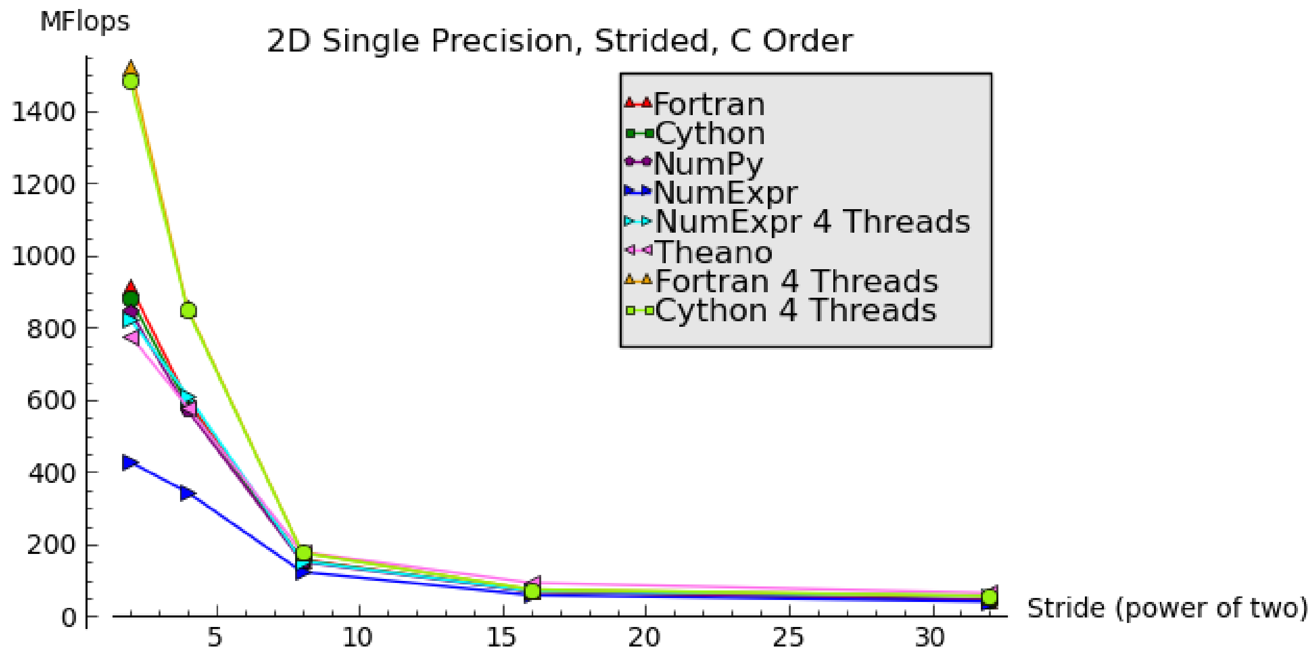

Figure 7: Benchmark with strided operands, GCC/GFortran 4.7.

entirely memory bound and there is no spatial reuse of data when the strides exceed cache-line sizes. For smaller strides we see that NumExpr doesn't perform well, likely due to startup overhead.

6.3 Contiguous in Inner Dimension

This benchmark operates on Fortran ordered arrays with a contiguous inner dimension (the columns). The rows are however strided (we take the even rows). We see again in Figure 9 how GCC does not take the vectorized path, which results in approximately half the performance for small data sizes. Comparing the explicitly vectorized version in Figure 10 however, we see that the performance of the Cython implementation is roughly equal to the GNU Fortran implementation, although slightly lower for small data sizes, likely due to constant overhead needed to select the specialization and checks for read-after-write situations. Fortran does not need to perform such checks, since the semantics of the language dictate that memory of input arrays to a subroutine do not overlap.

We note again how Theano performs bad, due to the Fortran ordering. Comparing Theano in this way is not very enlightening. Indeed, when we use C-ordered arrays, it shows performance that comes much closer to the performance of Fortran and Cython.

Analyzing the performance with Intel® compilers, shown in Figure 11, leads again to the same conclusion as in section 6.1, namely that it does take the vectorized path for the Cython expression, and that it does not properly parallelize array expressions.

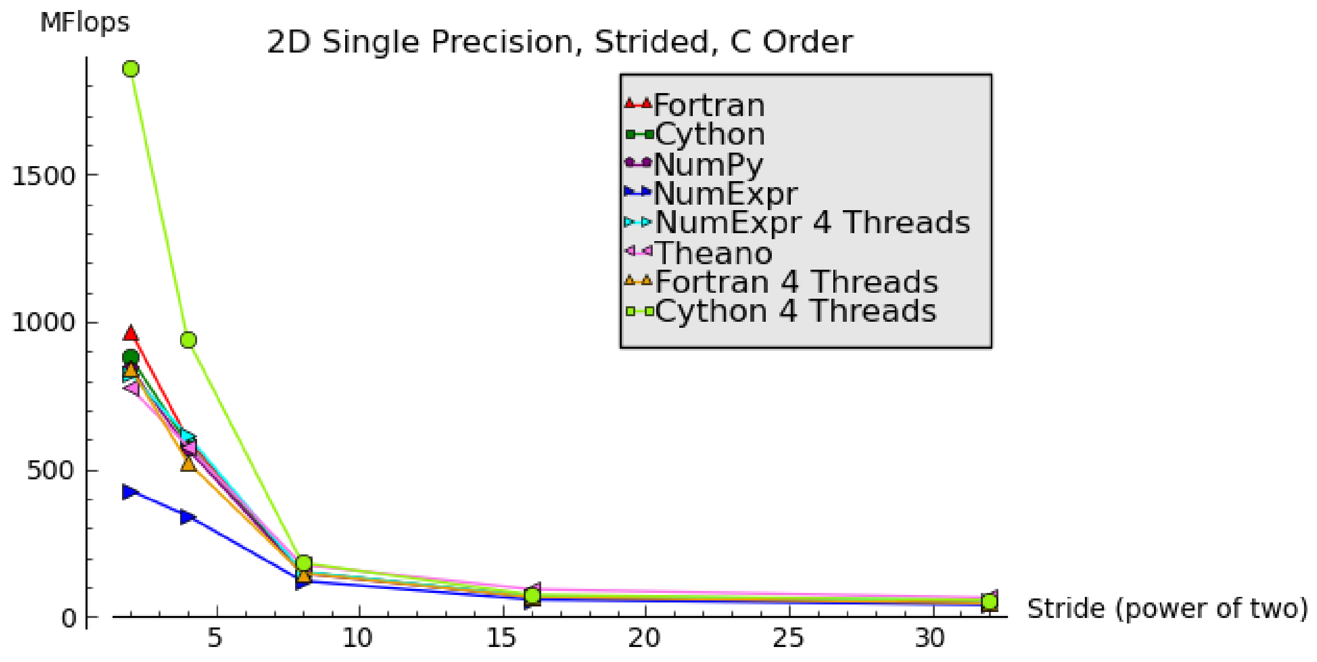

Figure 8: Benchmark with strided operands, Intel® C/Fortran version 12.1.5.

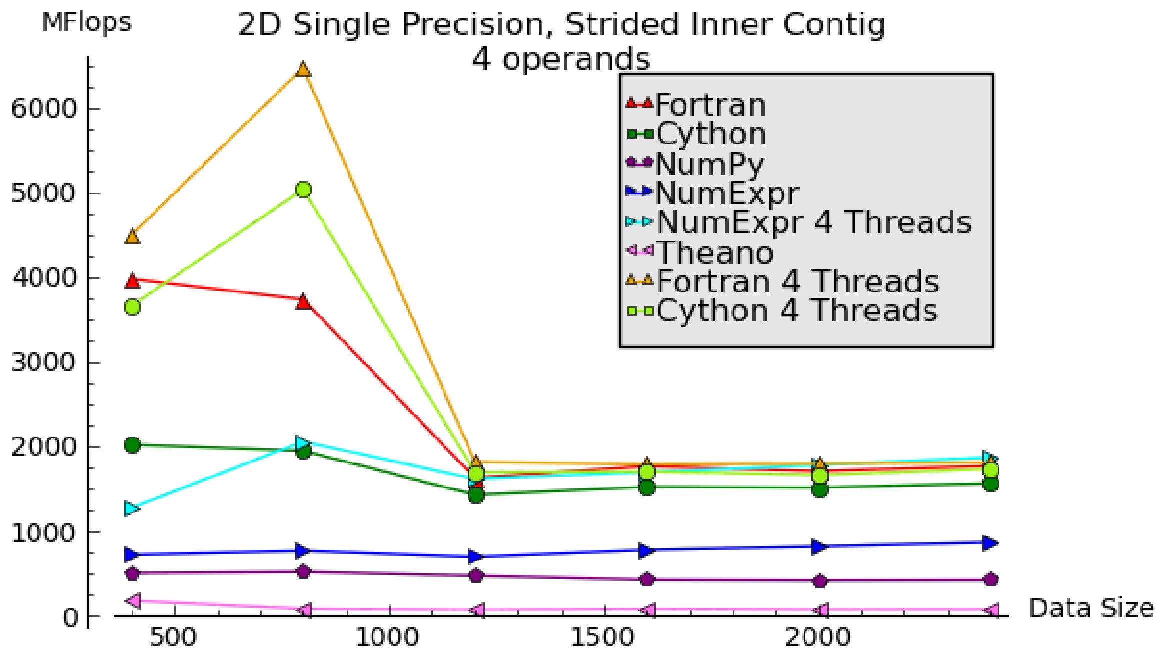

Figure 9: Benchmark with operands contiguous in the first dimension, GCC/GFortran 4.7, unvectorized.

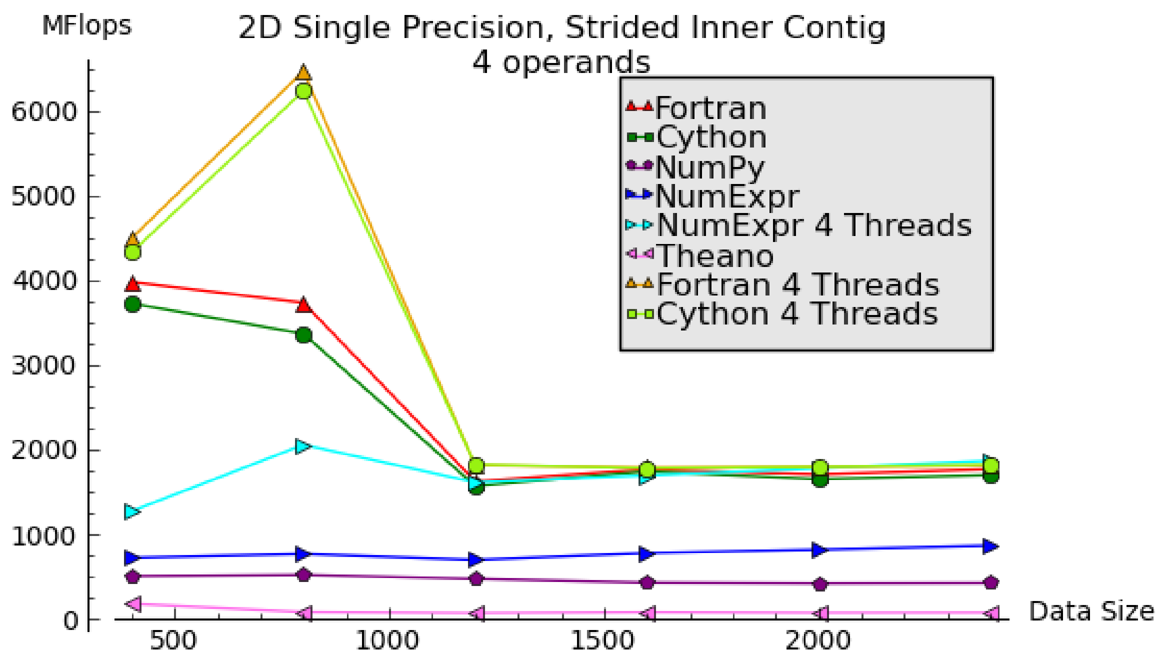

Figure 10: Benchmark with operands contiguous in the first dimension, GCC/GFortran 4.7. This benchmark was explicitly vectorized by our compiler.

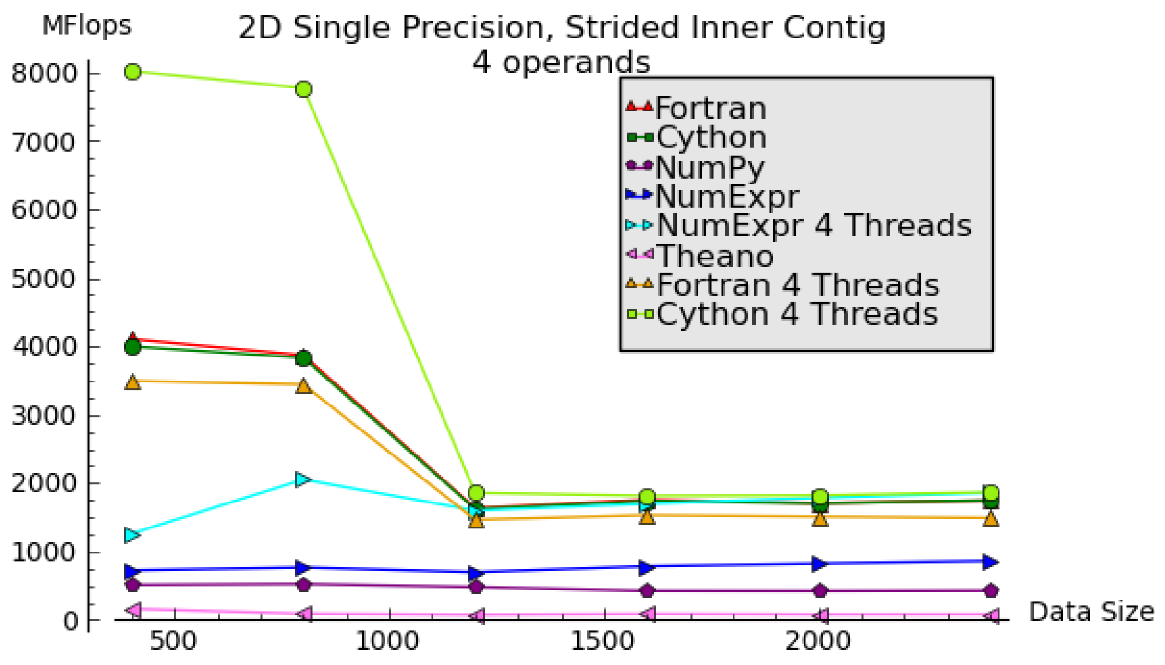

Figure 11: Benchmark with operands contiguous in the first dimension, Intel® C/Fortran version 12.1.5.

6.4 Tiled

The tiled benchmarks are perhaps the most interesting in the suite. Even though we do not use optimal tiling parameters as discussed in section 5.3 but instead auto-tuned ones, we see great speedups over both GFortran and Intel® Fortran.

This benchmarks mixes C-contiguous operands with Fortran-contiguous ones. In Figure 12 we see very similar performance between GFortran and Theano. This is very likely due to both approaches using a single iteration order (either C or Fortran). As discussed in sections 2.6, 4.4 and 5.3, tiling is an important optimization to utilize spatial locality. We see that we outperform GNU Fortran by almost a factor of three for larger data sizes, and the threaded Fortran version only marginally outperforms our single-threaded implementation for larger data sizes.

In Figure 13 we see that Intel® Fortran does a better job than GFortran, but we still outperform it by a good margin. For instance for an 800x800 matrix we register a speedup of 1.86, and for 2000x2000 a speedup of 1.43.

We see that in our benchmarks we outperform NumPy and NumExpr by great margins, and that even a threaded NumExpr implementation does not come close to our single-threaded tiled implementation, which is because neither library performs tiling. Note finally how we have not implemented our SIMD transpose approach as discussed in section 4.4.1, which can bring an additional significant speedup.

The results for a tiled and strided implementation can be found in Appendix A, Figure 14 and Figure 15, since they show great similarity with the results of the contiguous tiled benchmark.

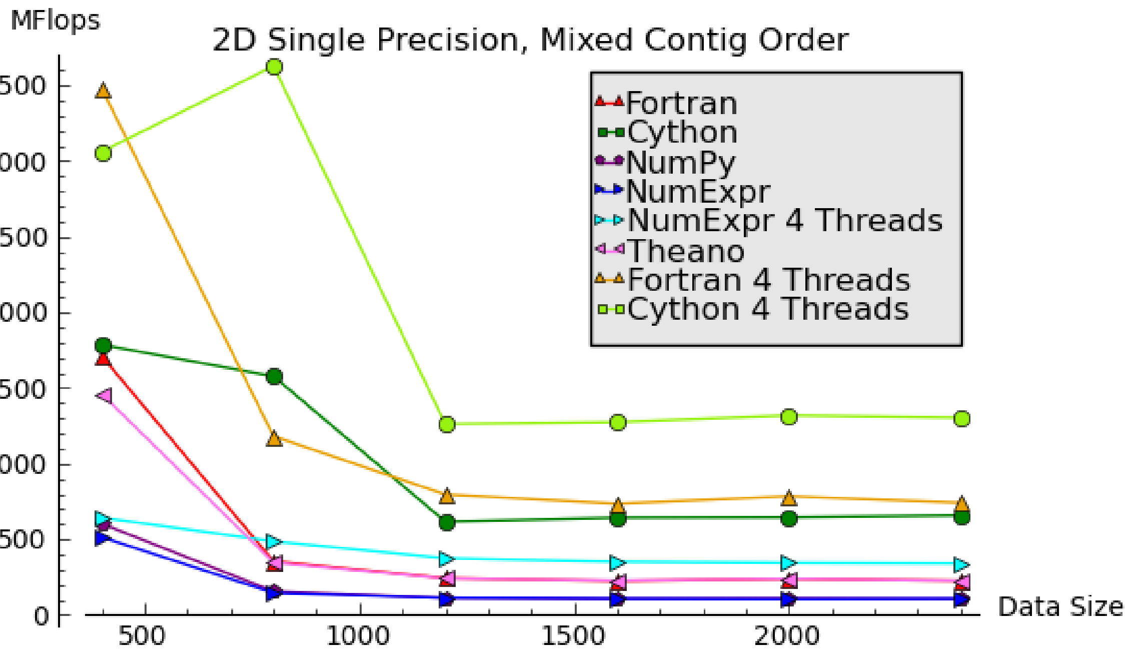

Figure 12: Benchmark with contiguous operands, mixed data order, GCC/GFortran 4.7.

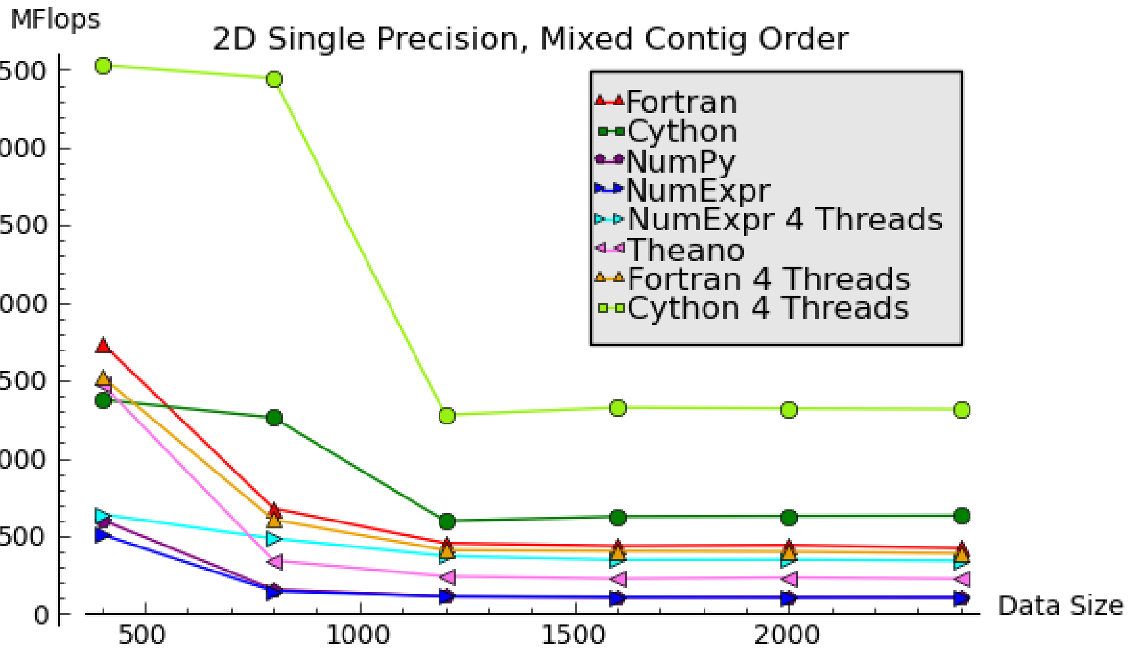

Figure 13: Benchmark with contiguous operands, mixed data order, Intel® C/Fortran version 12.1.5.

7 Conclusion

We have successfully implemented and evaluated a compiler for array expressions. We saw that there is much more to array expressions than straightforward scalarization, and we demonstrated the power of static and runtime compilation techniques. We have seen our compiler outperform open source and commercial Fortran compilers by a good margin using these techniques, but we have also seen that static specialization is limited by the compile-time information. It is not possible to apply optimizations such as loop-invariant code motion when broadcasting, unless the broadcasting information is part of the array types, which may not only be awkward, but will result in a loss of generality in user code. Lazy evaluation combined with runtime just-in-time specialization results in a speedup of nearly a factor of 2 over what our static specialization technique can provide in such situations.

We have also considered that runtime compilation will also be particularly useful when data layouts are fixed at runtime, to enable SIMDization using vector-sized transposes in the tiling specializations. Although we have not conducted a thorough study into lazy evaluation, its power and ability to trivially bypass control flow and span subroutines (and different modules and even projects) are clear.

We also showed how we can use traditional dependence tests to avoid temporary arrays for overlapping memory between arrays on the right-hand side and the left-hand side. This technique applies to our runtime as well as our compile-time environment, and is more general and effective than what compile-time analysis could provide for arrays represented as views on memory.

Finally, we have seen that it is possible to design stand-alone compiler components that can be reused among projects. This promotes interoperability between projects and allows developer resources to be focussed on improving a single project that benefits everyone, which is in better accordance with open source spirit than an ever increasing disparity and rivalry between technologies. We have seen how such a component can be designed, and how we can minimize the efforts needed to develop different code generation back-ends by defining complex operations in terms of simple ones, a concept that can be applied recursively.

8 Future Work

Although our array expression compiler is effective, a lot of work remains. For instance, we have mostly focussed on the performance aspects of element-wise functionality, and have not yet implemented reductions. We have demonstrated the effectiveness of various techniques, but have not implemented all of them in our compiler, including the SIMD vector transposes, computing optimal tiling parameters, using the blocking technique, or using the dependence tester to prove independence or suggest techniques such as the runtime equivalent of loop interchange or loop reversal to avoid the dependence. We are however happy with the results, since a demonstration of the effectiveness is in itself a virtue.

We have also implemented an explicit vectorizer, but this is not yet implemented in the runtime compiler (the LLVM code generator), and additional study is needed to establish the runtime cost of auto-vectorization. The runtime compiler has also no ability to generate parallel loops, which should be addressed in the future through calls to libgomp [60] or our own implementation. Further efforts are needed to implement techniques to reduce runtime compilation overhead, including code caching and improving the performance of the compiler itself. Although our project supports element-wise (math) functions, our vectorizer can not handle vector math at the time of writing using libraries like Intel® MKL [61].

Our project is only partially integrated in Numba [62], which only uses the type system and not the array expression functionality at the time of writing. Integration into the Theano project is pending, and we imagine integration into additional projects in the future.

A Appendix

Listing 31: Simplified code for Specialization Selection for $a[:, :] = b[:, :] + c[:, :]$

```
/* LHS */
  if (unlikely(!v_5test2_a.memview)) {
      __Pyx_RaiseUnboundLocalError("a"); {filename = f[0]; lineno =
       36; clineno = __LINE__; goto L1_error;} }
  t_2 = v_5test2_a;
  __PYX_INC_MEMVIEW(&t_2, 1);
  /* Evaluate operands */
  if (unlikely(!v_5test2_b.memview)) {
      __Pyx_RaiseUnboundLocalError("b"); {filename = f[0]; lineno =
       36; clineno = __LINE__; goto L1_error;} }
  t_4 = v_5test2_b;
  __PYX_INC_MEMVIEW(&t_4, 1);
  if (unlikely(!v_5test2_c.memview)) {
      __Pyx_RaiseUnboundLocalError("c"); {filename = f[0]; lineno =
       36; clineno = __LINE__; goto L1_error;} }
  t_5 = v_5test2_c;
  __PYX_INC_MEMVIEW(&t_5, 1);
  /* Check overlapping memory */
  t_6 = (slices_overlap(t_2, t_4, 2, 2) && read_after_write(t_2,
      t_4, 2)) || (slices_overlap(t_2, t_5, 2, 2) &&
      read_after_write(t_2, t_5, 2));
  /* Broadcast all operands in RHS expression */
  t_7 = 0;
  t_3.shape[0] = 1;
  t_3.shape[1] = 1;
  if (unlikely(memoryview_broadcast(&t_3.shape[0], &t_4.shape[0],
      &t_4.strides[0], 2, 2, &t_7) < 0)) {filename = f[0]; lineno =
       36; clineno = __LINE__; goto L1_error;}
  if (unlikely(memoryview_broadcast(&t_3.shape[0], &t_5.shape[0],
      &t_5.strides[0], 2, 2, &t_7) < 0)) {filename = f[0]; lineno =
       36; clineno = __LINE__; goto L1_error;}
  if (unlikely(verify_shapes(t_2, t_3, 2, 2) < 0)) {filename = f
      [0]; lineno = 36; clineno = __LINE__; goto L1_error;}
  /* Allocate scratch space if needed */
  if (unlikely(t_6)) {
    t_8 = (double *) malloc(sizeof(double) * t_3.shape[0] * t_3.
        shape[1]);
    if (!t_8) {
      PyErr_NoMemory();
      {filename = f[0]; lineno = 36; clineno = __LINE__; goto
          L1_error;}
    }
    t_3.data = (char *) t_8;
    fill_contig_strides_array(&t_3.shape[0], &t_3.strides[0],
        sizeof(double), 2, get_best_slice_order(t_3, 2));
  } else {t_8 = NULL;
    t_3.data = t_2.data;
    t_3.strides[0] = t_2.strides[0];
    t_3.strides[1] = t_2.strides[1];
    }
    /* Evaluate expression */
    {
        const __Pyx_memviewslice *array_ops[3] = { &t_3, &t_4, &t_5
            };
        int ndims[3] = { 2, 2, 2 };
        Py_ssize_t itemsizes[3] = { sizeof(double), sizeof(double),
            sizeof(double) };
        t_9 = get_arrays_ordering(array_ops, ndims, itemsizes, 3);
    }
    /* Contiguous specialization */
    if (t_9 & __PYX_ARRAYS_ARE_CONTIG && !t_7) {
      (void) array_expression_0contig(&t_3.shape[0], (double *) t_3.
          data, (double *) t_4.data, (double *) t_5.data,
          vector_get_omp_size(2));
    }
    else if (t_9 & (__PYX_ARRAYS_ARE_MIXED_CONTIG|
        __PYX_ARRAYS_ARE_MIXED_STRIDED)) {
      /* Tiled specializations */
      if (t_9 & __PYX_ARRAY_C_ORDER) {
        (void) array_expression_1tiled_c(&t_3.shape[0], (double *)
            t_3.data, &t_3.strides[0], (double *) t_4.data, &t_4.
            strides[0], (double *) t_5.data, &t_5.strides[0],
vector_get_tile_size(sizeof(double), 2), vector_get_omp_size(2));
      } else {
        (void) array_expression_2tiled_fortran(&t_3.shape[0], (
            double *) t_3.data, &t_3.strides[0], (double *) t_4.data,
             &t_4.strides[0], (double *) t_5.data, &t_5.strides[0],
vector_get_tile_size(sizeof(double), 2), vector_get_omp_size(2));
      }
    }
    else if (t_9 & __PYX_ARRAYS_ARE_INNER_CONTIG) {
      if (t_9 & __PYX_ARRAY_C_ORDER) {
        (void) array_expression_3inner_contig_c(&t_3.shape[0], (
            double *) t_3.data, &t_3.strides[0], (double *) t_4.data,
             &t_4.strides[0], (double *) t_5.data, &t_5.strides[0],
vector_get_omp_size(2));
      } else {
        (void) array_expression_4inner_contig_fortran(&t_3.shape[0],
             (double *) t_3.data, &t_3.strides[0], (double *) t_4.
            data, &t_4.strides[0], (double *) t_5.data,
&t_5.strides[0], vector_get_omp_size(2));
      }
    }
    else {
      /* Strided specializations */
      if (t_9 & __PYX_ARRAY_C_ORDER) {
        (void) array_expression_5strength_reduced_strided(&t_3.shape
            [0], (double *) t_3.data, &t_3.strides[0], (double *) t_4
            .data, &t_4.strides[0], (double *) t_5.data,
&t_5.strides[0], vector_get_omp_size(2));
      } else {
        (void) array_expression_6strength_reduced_strided_fortran(&
            t_3.shape[0], (double *) t_3.data, &t_3.strides[0], (
            double *) t_4.data, &t_4.strides[0], (double *) t_5.data,
&t_5.strides[0], vector_get_omp_size(2));
      }
    }
    /* Cleanup */
    if (unlikely(t_6)) {
      free(t_8);
      t_8 = NULL;
    }
    __PYX_XDEC_MEMVIEW(&t_4, 1);
    __PYX_XDEC_MEMVIEW(&t_5, 1);
    __PYX_XDEC_MEMVIEW(&t_3, 1);
    __PYX_XDEC_MEMVIEW(&t_2, 1);
```

Listing 32: Strided Implementation for $a[:,:] = b[:,:] + c[:,:]$

```
static int array_expression_5strength_reduced_strided(
    Py_ssize_t const *const CYTHON_RESTRICT shape,
    double *const CYTHON_RESTRICT op1_data,
    Py_ssize_t const *const CYTHON_RESTRICT op1_strides,
    double const *const CYTHON_RESTRICT op2_data,
    Py_ssize_t const *const CYTHON_RESTRICT op2_strides,
    double const *const CYTHON_RESTRICT op3_data,
    Py_ssize_t const *const CYTHON_RESTRICT op3_strides,
    Py_ssize_t const omp_size)
{
    Py_ssize_t const op1_stride3 = (op1_strides[0] /
                                    sizeof(double));
    Py_ssize_t const op1_stride4 = (op1_strides[1] /
                                    sizeof(double));
    double *CYTHON_RESTRICT temp5 = op1_data;
    Py_ssize_t const op2_stride7 = (op2_strides[0] /
                                    sizeof(double));
    Py_ssize_t const op2_stride8 = (op2_strides[1] /
                                    sizeof(double));
    double const *CYTHON_RESTRICT temp9 = op2_data;
    Py_ssize_t const op3_stride11 = (op3_strides[0] /
                                    sizeof(double));
    Py_ssize_t const op3_stride12 = (op3_strides[1] /
                                    sizeof(double));
    double const *CYTHON_RESTRICT temp13 = op3_data;
    Py_ssize_t const temp0 = (shape[0] * shape[1]);
    Py_ssize_t temp2;
    #ifdef _OPENMP
    #pragma omp parallel for if((temp0 > omp_size)) \
                             lastprivate(temp2)      \
                             private(temp5, temp9, temp13) \
                             default(none)
    #endif
    for (temp2 = 0; temp2 < shape[0]; temp2++) {
        double *CYTHON_RESTRICT temp6;
        double const *CYTHON_RESTRICT temp10;
        double const *CYTHON_RESTRICT temp14;
        Py_ssize_t temp1;
        #ifdef _OPENMP
        temp5 = (op1_data + (op1_stride3 * temp2));
        #endif
        temp6 = temp5;
        #ifdef _OPENMP
        temp9 = (op2_data + (op2_stride7 * temp2));
        #endif
        temp10 = temp9;
        #ifdef _OPENMP
        temp13 = (op3_data + (op3_stride11 * temp2));
        #endif
        temp14 = temp13;
        #ifdef __INTEL_COMPILER
        #pragma simd
        #endif
        for (temp1 = 0; temp1 < shape[1]; temp1++) {
            (*temp6) = ((*temp10) + (*temp14));
            temp6 += op1_stride4;
            temp10 += op2_stride8;
            temp14 += op3_stride12;
        }
        #ifndef _OPENMP
        temp5 += op1_stride3;
        temp9 += op2_stride7;
        temp13 += op3_stride11;
        #endif
    }
    return 0;
}
```

Listing 33: Tiled implementation for $a[:,:] = b[:,:] + c[:,:]$

```
static int
array_expression1tiled_c(
        Py_ssize_t const *const CYTHON_RESTRICT shape,
        double *const CYTHON_RESTRICT op1_data,
        Py_ssize_t const *const CYTHON_RESTRICT op1_strides,
        double const *const CYTHON_RESTRICT op2_data,
        Py_ssize_t const *const CYTHON_RESTRICT op2_strides,
        double const *const CYTHON_RESTRICT op3_data,
        Py_ssize_t const *const CYTHON_RESTRICT op3_strides,
        Py_ssize_t const blocksize,
        Py_ssize_t const omp_size)
{
    Py_ssize_t const op1_stride7 = (op1_strides[0] /
                                    sizeof(double));
    Py_ssize_t const op1_stride8 = (op1_strides[1] /
                                    sizeof(double));
    double *CYTHON_RESTRICT temp9 = op1_data;
    Py_ssize_t const op2_stride13 = (op2_strides[0] /
                                    sizeof(double));
    Py_ssize_t const op2_stride14 = (op2_strides[1] /
                                    sizeof(double));
    double const *CYTHON_RESTRICT temp15 = op2_data;
    Py_ssize_t const op3_stride19 = (op3_strides[0] /
                                    sizeof(double));
    Py_ssize_t const op3_stride20 = (op3_strides[1] /
                                    sizeof(double));
    double const *CYTHON_RESTRICT temp21 = op3_data;
    Py_ssize_t const temp0 = (shape[0] * shape[1]);
    Py_ssize_t temp2;
    #ifdef _OPENMP
    #pragma omp parallel for if((temp0 > omp_size)) \
                             lastprivate(temp2)      \
                             private(temp9, temp15, temp21) \
                             default(none)
    #endif
    for (temp2 = 0; temp2 < shape[0]; temp2 += blocksize) {
        double *CYTHON_RESTRICT temp10;
        double const *CYTHON_RESTRICT temp16;
        double const *CYTHON_RESTRICT temp22;
        Py_ssize_t temp1;
        #ifdef _OPENMP
        temp9 = (op1_data + (op1_stride7 * temp2));
        #endif
        temp10 = temp9;
        #ifdef _OPENMP
        temp15 = (op2_data + (op2_stride13 * temp2));
        #endif
        temp16 = temp15;
        #ifdef _OPENMP
        temp21 = (op3_data + (op3_stride19 * temp2));
        #endif
        temp22 = temp21;
```

```
            for (temp1 = 0; temp1 < shape[1]; temp1 += blocksize) {
                double *CYTHON_RESTRICT temp11;
                double const *CYTHON_RESTRICT temp17;
                double const *CYTHON_RESTRICT temp23;
                Py_ssize_t temp3;
                Py_ssize_t temp4;
                Py_ssize_t temp6;
                temp11 = temp10;
                temp17 = temp16;
                temp23 = temp22;
                temp3 = (((temp2 + blocksize) < shape[0]) ?
                                (temp2 + blocksize) : shape[0]);
                temp4 = (((temp1 + blocksize) < shape[1]) ?
                                (temp1 + blocksize) : shape[1]);
                for (temp6 = temp2; temp6 < temp3; temp6++) {
                    double *CYTHON_RESTRICT temp12;
                    double const *CYTHON_RESTRICT temp18;
                    double const *CYTHON_RESTRICT temp24;
                    Py_ssize_t temp5;
                    temp12 = temp11;
                    temp18 = temp17;
                    temp24 = temp23;
                    for (temp5 = temp1; temp5 < temp4; temp5++) {
                        (*temp12) = ((*temp18) + (*temp24));
                        temp12 += op1_stride8;
                        temp18 += op2_stride14;
                        temp24 += op3_stride20;
                    }
                    temp11 += op1_stride7;
                    temp17 += op2_stride13;
                    temp23 += op3_stride19;
                }
                temp10 += (op1_stride8 * blocksize);
                temp16 += (op2_stride14 * blocksize);
                temp22 += (op3_stride20 * blocksize);
            }
            #ifndef _OPENMP
            temp9 += (op1_stride7 * blocksize);
            temp15 += (op2_stride13 * blocksize);
            temp21 += (op3_stride19 * blocksize);
            #endif
        }
        return 0;
    }
```

Listing 34: Generated C Code for Optimized Broadcasting

```c
static int lazy0_0inner_contig_c(npy_intp * shape, double *
    op0_data, Py_ssize_t * op0_strides, double * op1_data,
    Py_ssize_t * op1_strides, double * op2_data, Py_ssize_t *
    op2_strides, double * op3_data, Py_ssize_t * op3_strides) {
     Py_ssize_t const op1_stride6 = (op1_strides[0] /
                                     sizeof(double));
     Py_ssize_t const op1_stride7 = (op1_strides[1] /
                                     sizeof(double));
     Py_ssize_t const op1_stride8 = (op1_strides[2] /
                                     sizeof(double));
     double * temp9 = op1_data;
     Py_ssize_t const op2_stride10 = (op2_strides[0] /
                                      sizeof(double));
     Py_ssize_t const op2_stride11 = (op2_strides[1] /
                                      sizeof(double));
     double * temp12 = op2_data;
     Py_ssize_t const op0_stride13 = (op0_strides[0] /
                                      sizeof(double));
     Py_ssize_t const op0_stride14 = (op0_strides[1] /
                                      sizeof(double));
     double * temp15 = op0_data;
     Py_ssize_t const op3_stride17 = (op3_strides[0] /
                                      sizeof(double));
     Py_ssize_t const op3_stride18 = (op3_strides[1] /
                                      sizeof(double));
     double * temp19 = op3_data;
     npy_intp temp0 = ((shape[0] * shape[1]) * shape[2]);
     npy_intp temp3;
     #ifdef _OPENMP
     #pragma omp parallel for if((temp0 > 1024))     \
                              lastprivate(temp3)      \
                              private(temp9, temp12, \
                                      temp15, temp19)\
                              default(none)
     #endif
     for (temp3 = 0; temp3 < shape[0]; temp3++) {
         double * temp16;
         double * temp20;
         double hoisted_temp4;
         npy_intp temp2;
         #ifdef _OPENMP
         temp9 = (op1_data + (op1_stride6 * temp3));
         temp12 = (op2_data + (op2_stride10 * temp3));
         temp15 = (op0_data + (op0_stride13 * temp3));
         #endif
         temp16 = temp15;
         #ifdef _OPENMP
         temp19 = (op3_data + (op3_stride17 * temp3));
         #endif
         temp20 = temp19;
         hoisted_temp4 = (*temp9);
         for (temp2 = 0; temp2 < shape[1]; temp2++) {
             double hoisted_temp5;
             npy_intp temp1;
             hoisted_temp5 = (hoisted_temp4 * temp12[temp2]);
             #ifdef __INTEL_COMPILER
             #pragma simd
             #endif
             for (temp1 = 0; temp1 < shape[2]; temp1++) {
                 temp16[temp1] = (hoisted_temp5 * temp20[temp1]);
             }
             temp16 += op0_stride14;
             temp20 += op3_stride18;
         }
         #ifndef _OPENMP
         temp9 += op1_stride6;
         temp12 += op2_stride10;
         temp15 += op0_stride13;
         temp19 += op3_stride17;
         #endif
     }
     return 0;
 }
```

Listing 35: Generated LLVM IR for Optimized Broadcasting

```
define i32 @lazy0inner_contig_c(i64* noalias nocapture, double*
    noalias nocapture, i64* noalias nocapture, double* noalias
    nocapture, i64* noalias nocapture, double* noalias nocapture,
    i64* noalias nocapture, double* noalias nocapture, i64* noalias
     nocapture) {
entry_0:
  %9 = load i64* %4
  %10 = udiv i64 %9, 8
  %11 = getelementptr i64* %4, i32 1
  %12 = load i64* %11
  %13 = getelementptr i64* %4, i32 2
  %14 = load i64* %13
  %15 = load i64* %6
  %16 = udiv i64 %15, 8
  %17 = getelementptr i64* %6, i32 1
  %18 = load i64* %17
  %19 = load i64* %2
  %20 = getelementptr i64* %2, i32 1
  %21 = load i64* %20
  %22 = load i64* %8
  %23 = getelementptr i64* %8, i32 1
  %24 = load i64* %23
  %25 = load i64* %0
  %26 = getelementptr i64* %0, i32 1
  %27 = load i64* %26
  %28 = mul i64 %25, %27
  %29 = getelementptr i64* %0, i32 2
  %30 = load i64* %29
  %31 = mul i64 %28, %30
  %32 = lshr i64 %22, 3
  %33 = mul i64 %32, 8
  %34 = lshr i64 %24, 3
  %35 = mul i64 %34, 8
  %36 = lshr i64 %19, 3
  %37 = mul i64 %36, 8
  %38 = lshr i64 %21, 3
  %39 = mul i64 %38, 8
  br label %for.cond_1

for.cond_1:                                       ; preds = %for.
    exit_8, %entry_0
  %lsr.iv6 = phi double* [ %54, %for.exit_8 ], [ %1, %entry_0 ]
  %lsr.iv = phi double* [ %53, %for.exit_8 ], [ %7, %entry_0 ]
  %temp3.0 = phi i64 [ 0, %entry_0 ], [ %52, %for.exit_8 ]
  %temp12.0 = phi double* [ %5, %entry_0 ], [ %51, %for.exit_8 ]
  %temp9.0 = phi double* [ %3, %entry_0 ], [ %50, %for.exit_8 ]
  %40 = load i64* %0
  %41 = icmp slt i64 %temp3.0, %40
  br i1 %41, label %for.body_3, label %for.exit_4

for.body_3:                                       ; preds = %for.
    cond_1
  %42 = load double* %temp9.0
  br label %for.cond_5

for.exit_4:                                       ; preds = %for.
   cond_1
  ret i32 0

for.cond_5:                                       ; preds = %for.
   exit_12, %for.body_3
  %lsr.iv9 = phi double* [ %64, %for.exit_12 ], [ %lsr.iv6, %for.
     body_3 ]
  %lsr.iv2 = phi double* [ %63, %for.exit_12 ], [ %lsr.iv, %for.
     body_3 ]
  %temp2.0 = phi i64 [ 0, %for.body_3 ], [ %62, %for.exit_12 ]
  %sunkaddr = ptrtoint i64* %0 to i64
  %sunkaddr13 = add i64 %sunkaddr, 8
  %sunkaddr14 = inttoptr i64 %sunkaddr13 to i64*
  %43 = load i64* %sunkaddr14
  %44 = icmp slt i64 %temp2.0, %43
  br i1 %44, label %for.body_7, label %for.exit_8

for.body_7:                                       ; preds = %for.
   cond_5
  %45 = getelementptr double* %temp12.0, i64 %temp2.0
  %46 = load double* %45
  %47 = fmul double %42, %46
  br label %for.cond_9

for.exit_8:                                       ; preds = %for.
   cond_5
  %48 = bitcast double* %lsr.iv to i1*
  %49 = bitcast double* %lsr.iv6 to i1*
  %50 = getelementptr double* %temp9.0, i64 %10
  %51 = getelementptr double* %temp12.0, i64 %16
  %52 = add i64 %temp3.0, 1
  %scevgep = getelementptr i1* %48, i64 %33
  %53 = bitcast i1* %scevgep to double*
  %scevgep8 = getelementptr i1* %49, i64 %37
  %54 = bitcast i1* %scevgep8 to double*
  br label %for.cond_1

for.cond_9:                                       ; preds = %for.
   body_11, %for.body_7
  %temp1.0 = phi i64 [ 0, %for.body_7 ], [ %59, %for.body_11 ]
  %sunkaddr15 = ptrtoint i64* %0 to i64
  %sunkaddr16 = add i64 %sunkaddr15, 16
  %sunkaddr17 = inttoptr i64 %sunkaddr16 to i64*
  %55 = load i64* %sunkaddr17
  %56 = icmp slt i64 %temp1.0, %55
  br i1 %56, label %for.body_11, label %for.exit_12

for.body_11:                                      ; preds = %for.
   cond_9
  %scevgep5 = getelementptr double* %lsr.iv2, i64 %temp1.0
  %57 = load double* %scevgep5
  %58 = fmul double %47, %57
  %scevgep12 = getelementptr double* %lsr.iv9, i64 %temp1.0
  store double %58, double* %scevgep12
  %59 = add i64 %temp1.0, 1
  br label %for.cond_9

for.exit_12:                                      ; preds = %for.
   cond_9
  %60 = bitcast double* %lsr.iv2 to i1*
  %61 = bitcast double* %lsr.iv9 to i1*
  %62 = add i64 %temp2.0, 1
  %scevgep4 = getelementptr i1* %60, i64 %35
  %63 = bitcast i1* %scevgep4 to double*
  %scevgep11 = getelementptr i1* %61, i64 %39
  %64 = bitcast i1* %scevgep11 to double*
  br label %for.cond_5
}
```

Listing 36: Compute Tiling Sizes based on [35]

```
import math

import numpy as np

def computerows(colsize, CS, N):
    cols_per_set = q1 = CS / N

    r1 = CS % N
    setdiff = N - r1
    cols_per_n = N / setdiff
    gap = N % setdiff

    if colsize == N:
        return cols_per_set
    elif colsize == r1 and colsize > setdiff:
        return cols_per_set + 1
    else:
        cols_per_setdiff = math.floor(setdiff / colsize)
        cols_per_gap = math.floor(gap / colsize)
        return int(cols_per_setdiff * cols_per_n * cols_per_set +
                   cols_per_gap * cols_per_set +
                   cols_per_setdiff * math.floor(r1 / setdiff) +
                      cols_per_gap)

def find_tile_sizes(arrays, cache_size):
    for array in arrays:
        colsize = N = max(array.strides) / array.dtype.itemsize
        rowsize = cache_size / N
        r = cache_size % colsize

        candidates = []
        if rowsize:
            candidates.append((rowsize, colsize))

        while colsize > 4 and r != 0:
            tmp = colsize
            colsize = r
            r = tmp % r
            rowsize = computerows(colsize, cache_size, N)
            candidates.append((rowsize, colsize))

        if array.strides[0] < array.strides[1]:
            candidates = [c[::-1] for c in candidates]

        print array.strides, candidates
        yield candidates

def maxarea(arrays, CS):
    list_of_candidate_sets = list(find_tile_sizes(arrays, CS))

    area = 0
    tile = None
```

```
for T1 in list_of_candidate_sets[0]:
    x0, y0 = T1
    for candidates in list_of_candidate_sets[1:]:
        curarea = 0
        for T2 in candidates:
            x1, y1 = T2
            if min(x0, x1) * min(y0, y1) > curarea:
                cur_x = min(x0, x1)
                cur_y = min(y0, y1)
                curarea = cur_x * cur_y

        x0, y0 = cur_x, cur_y

    if x0 * y0 > area:
        tile = x0, y0
        area = x0 * y0

return tile
```

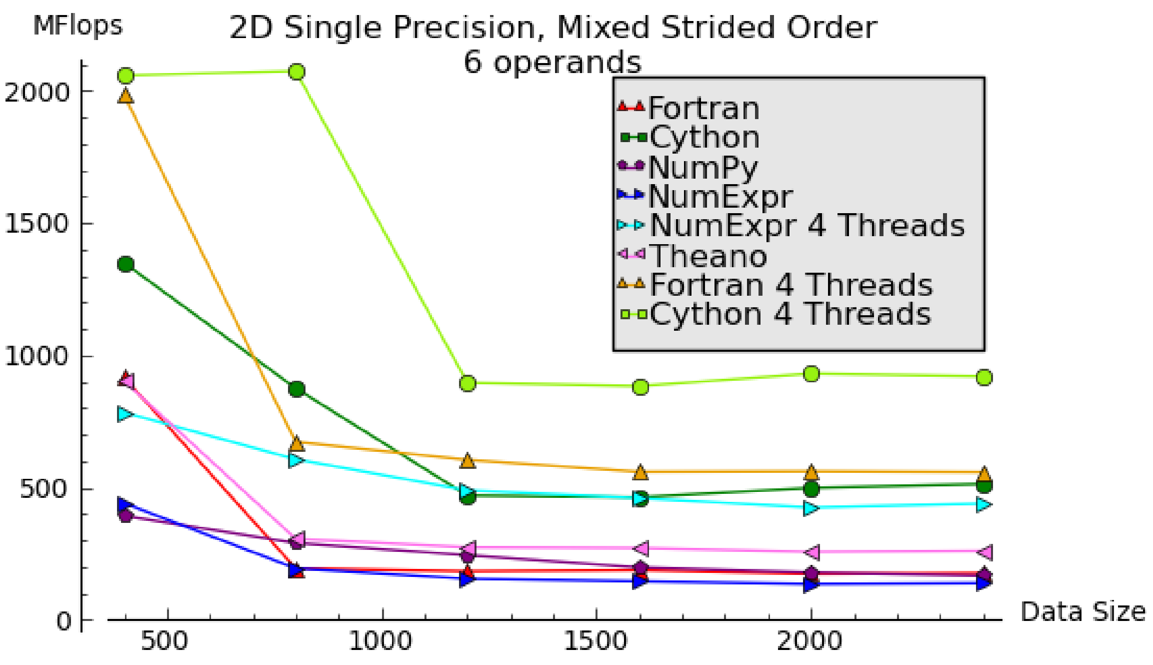

Figure 14: Benchmark with strided operands, mixed data order, GCC/GFortran 4.7.

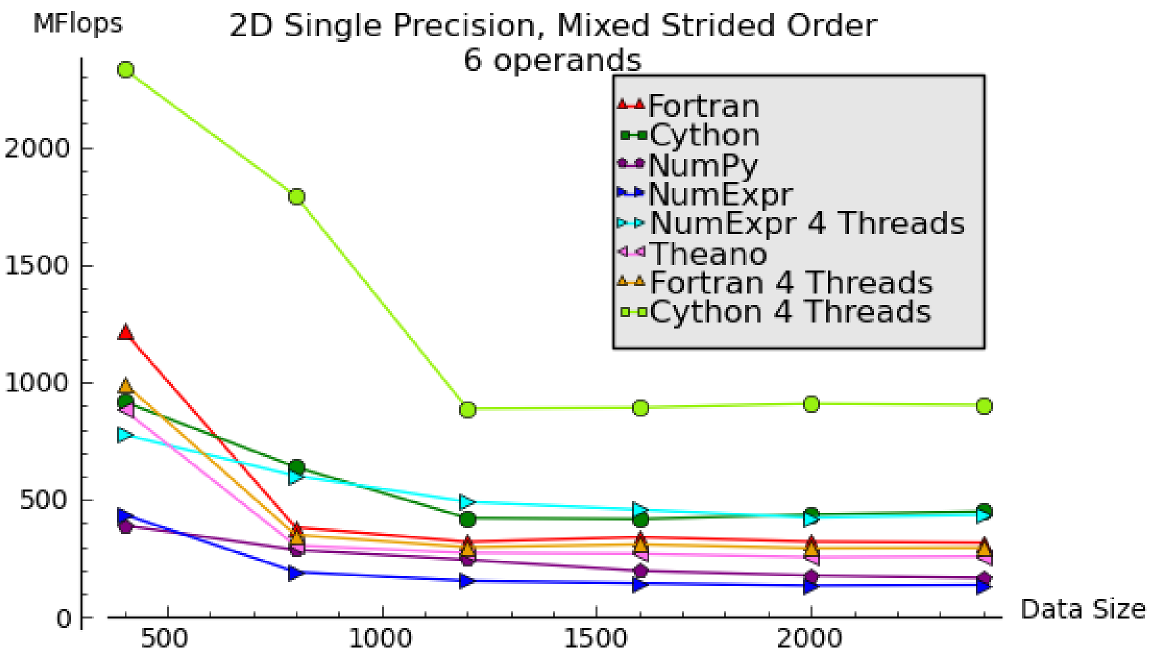

Figure 15: Benchmark with strided operands, mixed data order, Intel® C/Fortran version 12.1.5.

References

[1] K. E. Iverson. *A Programming Language*. Wiley, 1962. ISBN: 0-471-43014-5.

[2] *Eigen, A C++ Template Library for Linear Algebra*. URL: `http://eigen.tuxfamily.org/`.

[3] G. Winkler J. Walter M. Koch and D. Bellot. *Blitz*. URL: `http://www.oonumerics.org/blitz/`.

[4] *Boost, Basic Linear Algebra Library*. URL: `http://www.boost.org/doc/libs/1_50_0/libs/numeric/ublas/doc/index.htm`.

[5] *NumPy*. URL: `http://numpy.scipy.org/`.

[6] G. Varoquaux S. van der Walt S. C. Colbert. "The NumPy Array: A Structure for Efficient Numerical Computation". In: *Computing in Science & Engineering* 13.2 (2009).

[7] T. E. Oliphant. "Guide to NumPy". In: (2006).

[8] *NumPy Broadcasting*. URL: `http://docs.scipy.org/doc/numpy/user/basics.broadcasting.html`.

[9] V. Adve C. Lattner. "The LLVM Instruction Set and Compilation Strategy". In: ().

[10] AMD. "CPUID Specification". In: ().

[11] *Cython*. URL: `http://cython.org`.

[12] D. S. Seljebotn S. Behnel R. W. Bradshaw. "Cython Tutorial". In: Proceedings of the 8th Python in Science Conference (SciPy 2009), 2009.

[13] *Theano*. URL: `http://deeplearning.net/software/theano/`.

[14] *numexpr*. URL: `http://code.google.com/p/numexpr/`.

[15] *NumPy Indexing*. URL: `http://docs.scipy.org/doc/numpy/reference/arrays.indexing.html`.

[16] *Tentative NumPy Tutorial*. URL: `http://www.scipy.org/Tentative_NumPy_Tutorial`.

[17] *Iterating Over Arrays*. URL: `http://docs.scipy.org/doc/numpy/reference/arrays.nditer.html`.

[18] *NDIter NumPy Enhancement Proprosal*. URL: `https://github.com/numpy/numpy/blob/master/doc/neps/new-iterator-ufunc.rst`.

[19] James Bergstra et al. "Theano: a CPU and GPU Math Expression Compiler". In: *Proceedings of the Python for Scientific Computing Conference (SciPy)*. Oral Presentation. Austin, TX, June 2010. URL: `http://www.iro.umontreal.ca/~lisa/pointeurs/theano_scipy2010.pdf`.

[20] *Basic Tensor*. URL: `http://deeplearning.net/software/theano/library/tensor/basic.html`.

[21] *Theano, Using the GPU*. URL: `http://deeplearning.net/software/theano/tutorial/using_gpu.html`.

[22] *Theano, Using Different Compiling Modes*. URL: `http://deeplearning.net/software/theano/tutorial/modes.html`.

[23] *PyPy*. URL: `pypy.org`.

[24] *Jython*. URL: www.jython.org.

[25] *IronPython*. URL: ironpython.net.

[26] H. P. Langtangen X. Cai and H. Moe. *On the performance of the Python programming language for serial and parallel scientific computations*. 2005. URL: http://dl.acm.org/citation.cfm?id=1239665.

[27] D. S. Seljebotn. "Fast numerical computations with Cython". In: University of Oslo. Proceedings of the 8th Python in Science Conference (SciPy 2009), 2009.

[28] *F2py*. URL: http://www.scipy.org/F2py.

[29] *Typed Memoryviews*. URL: http://docs.cython.org/src/userguide/memoryviews.html.

[30] T. Oliphant and C. Banks. *PEP 3118 - Revising the buffer protocol*. URL: http://www.python.org/dev/peps/pep-3118/.

[31] *Python/C API Reference Manual*. URL: http://docs.python.org/c-api/.

[32] D. Ju G. Hwang J. K. Lee. "An Array Operation Synthesis Scheme to Optimize Fortran 90 Programs". In: ().

[33] E. E. Rothberg M. S. Lam and M. E. Wolf. "The Cache Performance and Optimizations of Blocked Algorithms". In: ().

[34] M. E. Wolf and M. S. Lam. "A Data Locality Optimizing Algorithm". In: ().

[35] K. S. McKinley S. Coleman. "Tile Size Selection Using Cache Organization and Data Layout". In: ().

[36] N. Anastopoulos E. Athanasaki K. Kourtis and N. Koziris. "Tuning Blocked Array Layouts to Exploit Memory Hierarchy in SMT Architectures". In: ().

[37] J. Chame and S. Moon. "A Tile Selection Algorithm for Data Locality and Cache Interference". In: ().

[38] C-H. Hsue and U. Kremer. "A Quantitative Analysis of Tile Size Selection Algorithms". In: ().

[39] M. E. Wolf. "Improving Locality and Parallelism in Nested Loops". In: ().

[40] *minivect*. URL: https://github.com/markflorisson88/minivect.

[41] A. Fog. "C++ vector class library". In: (). URL: http://www.agner.org/optimize/vectorclass.pdf.

[42] Martin von Löwis. *PEP 353 - Using ssize_t as the index type*. URL: http://www.python.org/dev/peps/pep-0353/.

[43] E. Rohou D. Nuzman S. Dyshel et al. "Vapor SIMD: Auto-Vectorize Once, Run Everywhere". In: (). URL: http://www.irisa.fr/alf/downloads/rohou/doc/Rohou_CGO11.pdf.

[44] A. Fog. "Optimizing Software in C++". In: (). URL: http://www.agner.org/optimize/optimizing_cpp.pdf.

[45] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures, A Dependence-Based Approach*. 2002. ISBN: 1-558-60286-0.

[46] R. D. Carmichael. *The Theory of Numbers*.

[47] R. D. Carmichael. *Diophantine Analysis*.

[48] Andrew Shao-Chun Huang. "Compilation Techniques for an XIMD Computer". In: (1992).

[49] W. Pugh. “On Fast Data Dependence Tests”. In: (1999).
[50] U. Banerjee. *Dependence Analysis for Supercomputing*. 1988. ISBN: 0-898-38289-0.
[51] Chau-Wen Tseng G. Goff K. Kennedy. “Practical Dependence Testing”. In: Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation.
[52] W. Pugh. “The Omega Test: a fast and practical integer programming algorithm for dependence analysis”. In: (1992).
[53] D. Klappholz X. Kong and K. Psarris. “The I Test: An Improved Dependence Test for Automatic Parallelization and Vectorization”. In: *IEEE Transactions on Parallel and Distributed Systems* 2.3 (1991).
[54] P. M. Petersen and D. A. Padua. “Static and Dynamic Evaluation of Data Dependence Analysis Techniques”. In: *IEEE Transactions on Parallel and Distributed Systems* 7.11 (1996).
[55] M. Wolfe. *Optimizing Supercompilers for Supercomputers*. 1989. ISBN: 0-262-73082-0.
[56] *Simple Dependence Tester*. URL: `https://github.com/markflorisson88/minivect/blob/master/demo/dependence_test.py`.
[57] “Cache Mapping”. In: (). URL: `http://williams.comp.ncat.edu/comp375/CacheMapping.pdf`.
[58] R. P. Grimaldi. *Discrete and Combinatorial Mathematics*. ISBN: 0-321-21103-0.
[59] M. Martonosi S. Ghosh and S. Malik. “Cache Miss Equations: A Compiler Framework for Analyzing and Tuning Memory Behavior”. In: (1999).
[60] *The libgomp ABI*. URL: `http://gcc.gnu.org/onlinedocs/libgomp/The-libgomp-ABI.html#The-libgomp-ABI`.
[61] *Intel® Math Kernel Library*. URL: `http://software.intel.com/en-us/articles/intel-mkl/`.
[62] *Numba*. URL: `https://github.com/numba/numba`.