APPy: Annotated Parallelism for Python on GPUs

Tong Zhou
Georgia Institute of Technology
USA
tz@gatech.edu

Jun Shirako
Georgia Institute of Technology
USA
shirako@gatech.edu

Vivek Sarkar
Georgia Institute of Technology
USA
vsarkar@gatech.edu

Abstract

GPUs are increasingly being used used to speed up Python applications in the scientific computing and machine learning domains. Currently, the two common approaches to leveraging GPU acceleration in Python are 1) create a custom native GPU kernel, and import it as a function that can be called from Python; 2) use libraries such as CuPy, which provides pre-defined GPU-implementation-backed tensor operators. The first approach is very flexible but requires tremendous manual effort to create a correct and high performance GPU kernel. While the second approach dramatically improves productivity, it is limited in its generality, as many applications cannot be expressed purely using CuPy's pre-defined tensor operators. Additionally, redundant memory access can often occur between adjacent tensor operators due to the materialization of intermediate results. In this work, we present APPy (Annotated Parallelism for Python), which enables users to parallelize generic Python loops and tensor expressions for execution on GPUs by adding simple compiler directives (annotations) to Python code. Empirical evaluation on 20 scientific computing kernels from the literature on a server with an AMD Ryzen 7 5800X 8-Core CPU and an NVIDIA RTX 3090 GPU demonstrates that with simple pragmas APPy is able to generate more efficient GPU code and achieves significant geometric mean speedup relative to CuPy (30× on average), and to three state-of-the-art Python compilers, Numba (8.3× on average), DaCe-GPU (3.1× on average) and JAX-GPU (18.8× on average).

CCS Concepts: • **Software and its engineering** → **Source code generation**.

Keywords: GPUs, Python, compilers, programming model, code generation

ACM Reference Format:
Tong Zhou, Jun Shirako, and Vivek Sarkar. 2024. APPy: Annotated Parallelism for Python on GPUs. In *Proceedings of the 33rd ACM SIGPLAN International Conference on Compiler Construction (CC '24), March 2–3, 2024, Edinburgh, United Kingdom.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3640537.3641575

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

CC '24, March 2–3, 2024, Edinburgh, United Kingdom
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0507-6/24/03
https://doi.org/10.1145/3640537.3641575

1 Introduction

Python has gained increasing popularity due to its concise and flexible language features and a rich ecosystem for various application domains, such as computational science, data science and machine learning. Programs in such domains naturally exhibit abundant data parallelism, making them amenable for GPU acceleration. Two common approaches to accelerate Python programs on the GPU are 1) create a custom native GPU kernel, and import it as a function that can be called from Python; 2) use libraries such as CuPy which provides predefined GPU-implementation-backed tensor operators. The first approach is very flexible, but programming the GPU is notoriously difficult [4, 14, 17], due to the inherent difficulty of parallel programming and the complex hierarchy of modern GPU's hardware parallelism and memory systems. It generally requires significant expertise and manual effort to create a correct high-performance GPU kernel. While the library operator-based approach dramatically simplifies programming and improves productivity, it is limited in its generality, as many applications cannot be expressed using only the pre-defined tensor operators (Listing 1 shows one such example). Additionally, data locality remains a major source of inefficiency for a sequence of operators, where the intermediate results will need to be stored back to the main memory and loaded again by the next operator, which can slow down the overall execution. There exist tensor operator compilers that are able to compile a sequence of operators and generate fused GPU kernels [5, 15, 18], however, these tools/techniques are often specific to machine learning, and are not applicable to general Python programs with combinations of explicit loops and generic tensor operators.

In this work, we present APPy: Annotated Parallelism for Python, a Python-embedded parallel programming model and JIT compiler that allows users to express their parallel program using loops, or tensor expressions or a mix of both, with annotated directives. Compared to existing GPU programming models, APPy has two distinct advantages:

- In APPy, the user writes sequential loops targeting at an abstract shared-memory multi-vector processor machine that exposes two layers of parallelism: synchronized vector processing (vectorization), and asynchronous multi-threading (parallelization). Users can

```
1 def group_by_sum(X, labels, centroids, M, N):
2     for i in range(M):
3         label = labels[i]
4         centroids[label, :N] += X[i, :N]
```

Listing 1. A kernel from the K-Means algorithm that groups rows from a matrix according to the label of each row. Such a kernel involves indirect memory access, and the same pattern cannot be expressed solely with tensor operators.

```
1 @appy.jit(auto_simd=True)
2 def group_by_sum(X, labels, centroids, M, N):
3     #pragma parallel for
4     for i in range(M):
5         label = labels[i]
6         #pragma atomic
7         centroids[label, :N] += X[i, :N]
```

Listing 2. Parallelized version of Listing 1. This implementation utilizes both loops (line 4-7) and tensor expressions (line 7). Besides parallelizing the for loop, the compiler also recognizes line 7 as a tensor expression, and will generate a loop automatically for it with auto-vectorization. Directive `#pragma atomic` indicates parallel reduction.

parallelize and vectorize a loop using OpenMP-like directives, `parallel for` and `simd`, respectively.
- In addition to loops, users can also write tensor/array expressions purely or combined with loops, and the compiler will automatically convert these array expressions into optimized loops.

Programming for the abstract machine model simplifies programming the GPU (the complex hierarchy of parallelism in modern GPUs is hidden), and makes the program portable for execution on any current or future hardware that fits with the abstract model. In addition, compiling both loops and tensor expressions makes APPy general enough to cover a diverse spectrum of scientific programs and maintain high productivity at the same time. Listing 2 shows the APPy parallelized version of kernel `group_by_sum`, shown in Listing 1.

In summary, we present the design of a loop-oriented programming model targeting an abstract multi-vector processor machine (section 3), as well as a tensor-oriented programming model (section 4) with array expressions. We then present an implementation of APPy and its code generation algorithms (section 5). Finally, we evaluate the performance of APPy on 20 scientific kernels (section 6), and show that by adding simple pragmas users can get significant performance improvement with APPy for scientific Python programs on the GPU, compared to other state-of-the-art programming tools and compilers.

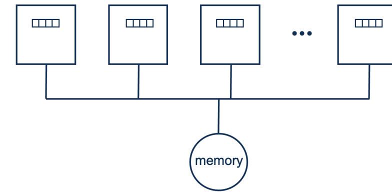

Figure 1. An illustration of the abstract machine model that exposes two layers of parallelism: multi-processor parallelization and vectorization within each processor.

2 Abstract Machine Model

In APPy, parallelism is specified against an abstract machine: a multi-vector processor. Such a system consists of one or more vector architecture processors [10] where each processor is able to process either a scalar or a vector of data in vector registers in a SIMD style, as illustrated in Fig. 1. The maximum vector length is represented by a built-in variable `appy.MVL`, and all vector operations must be of equal or smaller length. One can apply lane-wise operations to a vector register, such as basic mathematical unary and binary operations, and cross-lane operations such as reductions. It also provides an `appy.where(condition, x, y)` instruction to select values from two vectors based on a condition. Besides the SIMD-level parallelism, the system consists of multiple such vector processors and is capable of executing multiple vector instruction streams concurrently in a MIMD style. The processors may communicate via shared memory, e.g. updating a memory location atomically. In the paper, we refer to the SIMD level parallelism as *vectorization* and the MIMD level parallelism as *parallelization*. Maximum parallelism is achieved when both parallelization and vectorization are utilized.

The programmer only needs to think about this abstract machine model when programming with APPy, and the compiler automatically compiles user code to actual GPU code, and handles the underlying complexity. We note that when programming such abstract machine, one can use regular control flow structures and array indexings and slicings, just as in regular Python programming. The only thing that's specific to the abstract machine is about parallelism, e.g. specifying a loop as parallel, and processing data in a vector architecture style. The mathematical functions and operators applied on the data can be viewed as native instructions provided by the machine. Section 3 and 4 present the programming interface of the loop-oriented model and the tensor-oriented model respectively.

3 Loop-Oriented Programming Interface

3.1 Parallelization

A loop can be parallelized by being annotated with `#pragma parallel for`, where the end of the loop acts as a synchronization point. A vector addition example is shown in Listing 3. Each loop iteration is said to be assigned to a *worker*, and the number of workers launched is always equal to the number of loop iterations, unless directive `#pragma sequential for` is used, which launches only one worker that executes all iterations sequentially, e.g. due to loop-carried dependences. Each worker is scheduled to a single abstract processor, and executes its instructions sequentially. A parallel for-loop must be a for-range loop, and the number of loop iterations must be known at kernel launch time, i.e. no dynamic parallelism.

Tensors within the parallel region must already be a GPU tensor (data reside in the GPU memory), e.g. created using `cupy/torch/jax` with data on the device. Such libraries also provide APIs to create a GPU tensor from a NumPy[9] array.

```
1 @appy.jit
2 def vector_add(A, B, C, N):
3     #pragma parallel for
4     for i in range(N):
5         C[i] = A[i] + B[i]
```

Listing 3. Parallelize a for loop with APPy via `#pragma parallel for`. "`#pragma ...`" is a regular comment in Python, but will be parsed and treated as a directive by APPy.

3.2 Vectorization

Although `#pragma parallel for` parallelizes a loop, maximum parallelism is achieved when the loop body is also vectorized, when applicable. APPy provides three ways to achieve vectorization: 1) use tensor/array expressions (compiler generates a loop automatically, with more details in section 4); 2) annotate a loop with the `#pragma simd`, which divides the loop into smaller chunks, and assigns each chunk to a worker; 3) manually strip-mine a loop and operate on vectors of size `appy.MVL` or smaller. The last approach is considered "assembly-level programming" of the abstract machine, while the first two approaches rely on the compiler to generate "assembly code" (strip-mined loops). To handle arbitrary sized input, APPy provides a convenient built-in function `appy.vidx(i, step, bound=N)` to create a vector of indices that start from `i` and are bounded by `N`, with maximum length `step`.

Listing 4 shows the APPy implementations of vector addition and sparse matrix vector multiplication (SpMV) using `parallel for` and `simd` directive combined to achieve maximum parallelism. As with the `parallel for` directive, the `simd` directive relies on the programmer to check for dependences and guarantee correctness.

A loop that is not applicable for parallelization may be vectorizable. One example is the `j` loop in the SpMV example, where it has dynamic loop bounds.

```
1 @appy.jit
2 def vector_add(A, B, C, N):
3     #pragma parallel for simd
4     for i in range(N):
5         C[i] = A[i] + B[i]
6
7 @appy.jit
8 def spmv(A_row, A_col, A_val, x, y, N):
9     #pragma parallel for
10    for i in range(N - 1):
11        y[i] = 0.0
12        #pragma simd
13        for j in range(A_row[i], A_row[1+i]):
14            col = A_col[j]
15            y[i] += A_val[j] * x[col]
```

Listing 4. Best performance practice is to utilize both parallelization (via `parallel for`) and vectorization (via `simd`).

3.3 Data Sharing

APPy does not require the programmer to manually specify whether each variable is private or shared. Instead, it enforces syntactical difference between array and non-array variables, and use simple rules to infer the scope of the variables. Array variables are always followed by a square bracket, such as `A[vi]`, and the rest are non-array variables[1]. Array variables inside the parallel region are always considered shared (and their data reside in the global memory of the GPU). Non-array variables defined within the parallel region are considered private to each worker. Read-Only non-array variables are shared. APPy prohibits multiple reaching definitions from both inside and outside the parallel region of a non-array variable, which prevents writing into a shared non-array variable. To achieve such effects, the idiom is make the variable an array of size 1 (thus it has a global scope). An example is parallel reduction into a scalar, as shown in Listing 5.

3.4 Synchronization

The only synchronization across workers (loop iterations) supported is atomically updating a memory location. This can be achieved by either annotating a statement with compiler directive `#pragma atomic` or by using "assembly-level" programming of the abstract machine via built-in instruction `appy.atomic_<op>`. Note that within a worker, no synchronization is necessary even if it works on multiple elements.

[1]These non-array variables act like vector registers in vector architecture, so they are not called "scalars".

```python
@appy.jit
def vector_sum(A, s, N):
    #pragma parallel for simd
    for i in range(N):
        #pragma atomic
        s[0] += A[i]
```

Listing 5. Parallel reduction idiom in APPy. The reduction variable is an array of size 1. Directive simd divides the loop into smaller chunks. Each worker gets one chunk, and performs local reduction (SIMD capability is utlized). Finally each worker performs an atomic update to the final sum.

4 Tensor-Oriented Programming Interface

In addition to loops, APPy also allows users to use tensor/array expressions and the tensors can have arbitrary size. This often results in more natural and succinct program compared to the loop oriented version. Listing 6 shows a generic implementation of the softmax activation function[8], a fundamental building block of machine learning, using loops only and loops combined with tensor expressions respectively. As can be seen, to perform a max or a sum reduction on a row, the generic model must create a loop that works on a limited amount of elements at a time. In contrast, the tensor-oriented model allows the user to directly work with tensors of arbitrary size and dimensions, where the compiler automatically converts the tensor expressions into explicit loops. To disambiguate, we use the term "tensor/array assignment" to refer to operations performed on tensors of arbitrary size (as a high-level programming notation), and use the term " vector statement" to refer to operations that can be directly executed by the abstract machine model, i.e. the size of the vector is up to maximum vector length of the machine.

The tensor-oriented model works by programming in and annotating tensor expressions. However the tensor expression must be expressed in *sliced index notation* before it can be annotated.

4.1 Sliced Index Notation

We introduce sliced index notation as a form of tensor expression that simplifies compiler transformations while maintaining legitimacy as Python/NumPy syntax (no domain-specific language is used). In sliced index notation, each dimension of every tensor must be expressed as a slice with at least the upper bound explicitly specified. This notation draws inspiration from the Einstein summation convention [23], extended to support operations beyond multiplication and addition. Here are some examples:

- Matrix addition: `C[:M, :N] = A[:M, :N] + B[:M, :N]`
- Row-wise summation: `B[:M] = sum(A[:M, :N], axis=1)`
- Broadcast: `C[:M, :N] = A[:M, None] + B[None, :N]`
- Transpose: `B[:M, :N] = transpose(A[:N, :M])`

```python
@jit
def softmax_loop_oriented(a, b, M, N):
    #pragma parallel for
    for i in range(M):
        m = float('-inf')
        #pragma simd
        for j in range(N):
            m = maximum(m, a[i,j])

        s = 0.0
        #pragma simd
        for j in range(N):
            s += exp(a[i,j] - m)

        #pragma simd
        for j in range(N):
            b[i,j] = exp(a[i,j] - m) / s

@jit(auto_simd=True)
def softmax_tensor_oriented(a, b, M, N):
    #pragma parallel for
    for i in range(M):
        m = max(a[i,:N])
        s = sum(exp(a[i,:N] - m))
        b[i,:N] = exp(a[i,:N] - m) / s
```

Listing 6. Two equivalent implementations of generic softmax using loops only and loops combined with tensor expressions respectively. The APPy compiler automatically converts tensor expressions into explicit loops with vectorization.

- Stencil: `B[1:M-1, 1:N-1] = 0.2 * (A[1:M-1, 1:N-1] + A[1:M-1, :N-2] + A[1:M-1, 2:N] + ...)`

The sliced index notation requires the user to define a variable (referred to as a *dimension variable*) for each unique dimension within the expression, e.g. M and N. The expression is then rewritten to utilize slicings, such as ":M" to represent dimensions in every tensor. A scalar dimension can be denoted by a scalar index. It's crucial to note that distinct dimensions must be represented using different variables, even if their runtime values are identical, as in the case of a square matrix. Furthermore, tensor assignments can reference different slices of the same dimension, provided that the lengths of the slices are consistent, e.g. in stencil computations.

The tensor-oriented model supports only three classes of tensor operations: element-wise (both unary and binary), broadcast, and reduction operations (all examples above belong to these categories). Operations that don't belong to these categories, such as sorting, squeezing (removing zeros and compacting), matrix inversion etc are not supported.

4.2 Tensor-Oriented Pragmas

A tensor expression in sliced index notation can be annotated. The annotation process goes by enumerating every distinct dimension slice in the expression, and optionally specifying a

list of properties for it[2]. The syntax is `{slice}=>{properties}`, and the following properties are currently supported (multiple properties are comma-separated):

- `parallel`: the dimension can be parallelized, i.e. translated to a parallel for loop. The default value is `False`.
- `simd`: the dimension can be executed in a SIMD fashion. The default value is `False`.
- `reduction/reduce`: the dimension is reduced. The default value is `False`.
- `le(constant)`: indicate the dimension is less or equal than a small constant, which may enable more optimizations, such as caching a small tensor in registers.

The tensor oriented model also comes with a new compiler option `auto_simd`, which automatically adds a `simd` property to the last dimension. This transformation is always legal due to the vectorized semantics of tensor expressions. Listing 7 shows example pragmas used for `gesummv` and `jacobi_2d` from polybench. Note that the order in which the slices appear in the pragma (from left to right) determines the loop order of the generated nested loop. For instance, annotation `1:M-1=>parallel 1:N-1=>parallel` would imply that `1:M-1` corresponds to the outer loop and `1:N-1` corresponds to the inner loop.

```
1  @appy.jit(auto_simd=True)
2  def gesummv(alpha, beta, A, B, x, y, tmp, M, N):
3      #pragma :M=>parallel :N=>reduction(sum:y)
4      y[:M] = mv(alpha * A[:M, :N], x[:N])
5      #pragma :M=>parallel :N=>reduction(sum:tmp)
6      tmp[:M] = mv(beta * B[:M, :N], x[:N])
7      #pragma :M=>parallel
8      y[:M] += tmp[:M]
9
10 @appy.jit(auto_simd=True)
11 def jacobi_2d_one_iteration(A, B, M, N):
12     #pragma 1:M-1=>parallel 1:N-1=>parallel
13     B[1:M-1, 1:N-1] = 0.2 * (A[1:M-1, 1:N-1] + A[1:M-1, :N-2] +
14                 A[1:M-1, 2:N] + A[2:M, 1:N-1] + A[0:M-2, 1:N-1])
15     #pragma 1:M-1=>parallel 1:N-1=>parallel
16     A[1:M-1, 1:N-1] = 0.2 * (B[1:M-1, 1:N-1] + B[1:M-1, :N-2] +
17                 B[1:M-1, 2:N] + B[2:M, 1:N-1] + B[0:M-2, 1:N-1])
```

Listing 7. Annotated version of `gesummv` and `jacobi_2d`, a linear algebra kernel and a stencil kernel from polybench. Within each tensor assignment statement, operator fusion will be performed automatically during code generation to improve locality.

5 Implementation Overview

The APPy compiler first performs a sequence of high-level analysis and transformations that rewrite the input program to "assembly-level" APPy code targeting the abstract multi-vector processor machine, where the program loads and processes data in strip-mined loops and "vector registers". Then it compiles this "assembly code" to actual GPU code that can run on the GPU. Specifically, APPy generates a Triton kernel[24] and its corresponding host code for each top-level parallel for-loop. The overall compilation flow is described in Alg. 1. For simplicity, we use code examples below to highlight the key points related to these transformations.

Algorithm 1 Top-level compilation process. $func$ is the function object to be compiled. A new function is returned.

```
1:  function CODEGEN(func)
2:      ast ← getAST(func)
3:      m ← newModule()
4:      ast ← HIGHLEVELTRANSFORM(ast)
5:      for node in depth-first-traversal of ast do
6:          if node is a top-level parallel for-loop then
7:              devFunc ← GENDEVICECODE(node)
8:              hostCode ← GENHOSTCODE(node)
9:              m.body.append(devFunc)
10:             ▷ This replacement modifies ast in-place
11:             replace node with hostCode
12:     append ast to m, dynamically load module m and
        return the new function from m.
```

5.1 High-Level APPy To "Assembly" APPy Transformation

In the interest of brevity, we highlight a few key transformations in this section using code examples.

5.1.1 Creating Temporary Arrays. In a tensor assignment, if the tensor to be stored also appears on the right hand side with non-identical slices, APPy creates a new temporary array and splits the original statement into two assignments, where the computed values are first stored in a temporary array (first assignment) and then stored in the final destination (second assignment) to ensure the correctness when generating a loop from the tensor expression[3]. Listing 2b shows the result of the transformation from 2a.

5.1.2 Inserting Synchronizations. APPy maps each vector statement to a thread block for flexibility and performance consideration. However, a thread block does not naturally have a vectorized execution semantic and may contain threads/warps that execute asynchronously. Therefore, APPy inserts synchronization statement after certain memory operations to ensure the sequential execution of the vector statements.

When inserting synchronizations, APPy performs an optimization specific to its programming model: *synchronizations*

[2]When the statement has only one dimension, the annotation can be skipped if all properties have their default values.

[3]This approach could be overly conservative, and some of the rewrites can be skipped with more advanced dependence analysis available.

```
1 #pragma parallel for
2 for i in range(M):
3     #pragma 1:N=>simd
4     A[i, 1:N] = 0.5 * (A[i, :N-1] + B[i, :N-1])
```

(a) Original input program using a combination of loop and tensor expressions.

```
1 tmp = empty_like(A)
2 #pragma parallel for
3 for i in range(M):
4     #pragma 1:N=>simd
5     tmp[i, 1:N] = 0.5 * (A[i, :N-1] + B[i, :N-1])
6     #pragma 1:N=>simd
7     A[i, 1:N] = tmp[i, 1:N]
```

(b) Rewrite and split the array assignment if in the original assignment, the tensor to be stored also appears on the right hand side with a different slice. This ensures correct code generation when the array assignment is converted to a loop.

```
1 tmp = empty_like(A)
2 #pragma parallel for
3 for i in range(M):
4     #pragma 1:N=>simd
5     tmp[i, 1:N] = 0.5 * (A[i, :N-1] + B[i, :N-1])
6     appy.syncthreads()
7     #pragma 1:N=>simd
8     A[i, 1:N] = tmp[i, 1:N]
9     appy.syncthreads()
```

(c) Insert thread synchronizations after memory operations to ensure correct execution when vector statements are mapped to multiple asynchronously executing threads. Line 6 and 9 will be translated to a call equivalent to `__syncthreads` in CUDA during the backend code generation.

```
1 tmp = empty_like(A)
2 #pragma parallel for
3 for i in range(M):
4     for _i0 in range(1, N, appy.MVL):
5         _v0 = appy.vidx(_i0, appy.MVL, N)
6         tmp[i, _v0] = 0.5 * (A[i, _v0-1] + B[i, _v0-1])
7     appy.syncthreads()
8     for _i1 in range(1, N, appy.MVL):
9         _v1 = appy.vidx(_i1, appy.MVL, N)
10        A[i, _v1] = tmp[i, _v1]
11    appy.syncthreads()
```

(d) Each tensor assignment is converted to a loop by the compiler. This code is ready to be converted to the backend code.

Figure 2. Some key steps of the high-level transformations using the same example.

within loops generated from element-wise and broadcast tensor assignments are unnecessary. Therefore, at code generation time, synchronizations are inserted before tensor assignments are converted to loops. Listing 2c shows the code state after inserting synchronizations.

5.1.3 Lowering Tensor Assignments To Loops.

A tensor assignment statement is converted to a loop nest by the compiler. An index variable and a loop is created for each unique dimension, and the slicings in the original statement are replaced by these new index variables. The depth of the loop nest equals the number of unique dimensions in the tensor assignment, with the original statement placed in the innermost loop. Listing 2d shows the transformation from 2c.

5.1.4 Reduction-Bounded Operator Fusion.

When converting a tensor expression into loops, the compiler automatically performs what we call *reduction-bounded operator fusion*. This type of fusion fuses a sequence of element-wise operators and stops when a reduction operator is encountered, which creates a fusion boundary. The reduction operator itself will be the last operator in this fusible group. Listing 8 shows the compiler-generated code for a generic matrix vector multiplication with operator fusion.

```
1 ## Before transformation
2 @appy.jit(auto_simd=True)
3 def kernel(alpha, A, x):
4     M, N = A.shape
5     #pragma :M=>parallel :N=>reduction(sum:y)
6     y[:M] = mv(alpha * A[:M, :N], x[:N])
7
8 ## After transformation
9 @appy.jit
10 def _generated(alpha, A, x):
11     M, N = A.shape
12     #pragma parallel for
13     for _i0 in range(0, M, 1):
14         y[_i0] = 0.0
15         for _i1 in range(0, N, appy.MVL):
16             _v1 = appy.vidx(_i1, appy.MVL, N)
17             y[_i0] += sum(alpha * A[_i0, _v1] * x[_v1])
```

Listing 8. User code and compiler generated code for matrix vector multiplication. The last line indicates the operator fusion (the intermediate results are never materialized).

Reduction-Bounded fusion is illustrated in Fig. 3, where each circle represents a data element, and each arrow signifies an operation as well as a data dependence. A horizontal arrow represents an element-wise operation, while a gather arrow represents a reduction operation, and a scatter arrow represents a broadcast operation. The same operation is applied vertically to different elements. op1 and op4 are element-wise operations, while op2 and op3 are reduction and broadcast operations, respectively. Different colors represent the worker assignments of the data elements.

In this figure, op1 and op2 can be safely fused for each worker, as can op3 and op4. Fusing op1 and op2 means that the same data element is immediately being applied op2 after op1, without waiting for all data elements to have undergone op1. However, fusing op3 together with op1 and op2 would not be legal since op3 requires op2 to be completed on all data elements first.

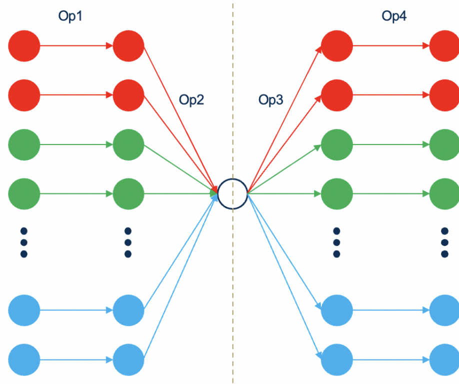

Figure 3. Reduction-Bounded fusion illustration. Vertically, workers (denoted by different colors) can work on their own data elements concurrently, and horizontally a sequence of operations (such as op1 and op2) can be fused, until a reduction operation like op2 is encountered, which ends a fusible group (itself included) and demands thread synchronizations (the dotted line).

Figure 3 also implies that when a worker is mapped to asynchronously executing hardware threads, the threads can work on their data elements concurrently, and only need to synchronize when a reduction operator is encountered.

5.2 Triton Background

After all the passes in the high-level transformation phase are performed, the code is now at "assembly" level and is ready to be lowered to the backend code. In our implementation we choose to generate Triton code instead of CUDA or OpenCL code because its programming model matches well with our abstract machine model, and simplifies our implementation. Nonetheless it's totally possible to generate CUDA or OpenCL from APPy.

First, we provide a brief introduction to the Triton language and compiler before getting into generating Triton code. Triton [24] is a GPU programming language and compiler that aims to make GPU programming more productive than CUDA. Based on Python, Triton provides a single-program-multiple-data (SPMD) programming interface, where the user specifies the behavior for a thread block as a whole, instead of individual threads, and the Triton compiler automatically distributes work across threads and generates SIMT-style code from it. This paradigm simplifies thread-block scope collective operations such as reduction, where the programmer just makes a high-level API call. Listing 9 illustrates a Triton kernel that performs a dot product of two vectors. The function `kernel` is the launch function that specifies the block size and grid size, and invokes `_kernel` (lines 15-16). Function `_kernel` is the Triton GPU kernel decorated with `triton.jit`, which the Triton compiler will JIT-compile to native code on the target machine, e.g. PTX code for NVIDIA hardware. Within the kernel, `tl.load` loads a block of data from global memory to registers that will be processed by the thread block. The block size `BN` is user-determined. Line 10 performs a local dot product (data are in registers) to get the partial results, and line 11 performs a global reduction via `tl.atomic_add`. It's worth noting that Triton (at present) still requires the programmer to insert thread synchronizations[4] if two statements might have cross-thread data dependencies.

```
1 import triton
2 import triton.language as tl
3
4 @triton.jit
5 def _kernel(a, b, c, N, BN: tl.constexpr):
6     i = tl.program_id(0) * BN
7     offsets = i + tl.arange(BN)
8     a_block = tl.load(a + offsets, mask=offsets<N)
9     b_block = tl.load(b + offsets, mask=offsets<N)
10    c_val = tl.sum(a_block * b_block)
11    tl.atomic_add(c, c_val)
12
13 def kernel(a, b, c, N):
14    BN = 512  # User-determined data block size
15    grid = (N + BN - 1) // BN
16    _kernel[grid](a, b, c, N, BN)
```

Listing 9. A Triton kernel that performs a dot product of vector `a` and `b`. The result is stored in `c`. `_kernel` is the Triton GPU kernel, launched at line 16.

5.3 "Assembly" APPy to Triton Code Generation

When generating the backend code, for each top-level parallel for-loop, a Triton kernel function is generated, together with its corresponding host code. The parallel for-loop can be a nested parallel loop, in which case a multi-dimensional thread block will be launched. The host code part includes creating an N-dimensional grid, where "N" is the nesting level of `#pragma parallel for`, and launching the kernel by passing the actual arguments, including the block size (`appy.MVL`), which is automatically determined by APPy given a hardware platform and a kernel. For our implementation and evaluation on an NVIDIA GPU, APPy's backend code includes some auto-tuning code that automatically selects the best `appy.MVL` value (that lead to the best runtime) from 4 possible options, 128, 256, 512 and 1024, for all applicable kernels. The best block size is cached for later invocations of the kernel. To generate a device function, an empty function decorated with *@triton.jit* is first created. The code generator first inserts the *program_id* statements into the function body, and then it traverses the loop body of the loop being compiled, converts the statements into corresponding Triton statements, and insert these Triton statements into

[4] Analogous to `__syncthreads()` in CUDA.

the kernel function body. This includes converting array indexings into Triton load and store statements etc.

6 Performance Evaluation

We compare APPy's performance with five other popular frameworks: NumPy (CPU baseline), Numba (CPU compiler) [13], CuPy (GPU baseline) [16], JAX-GPU (GPU compiler) [6] and DaCe-GPU (GPU compiler) [26]. We evaluate the performance of APPy using 20 kernels from the NPBench benchmarking framework [25]. Specifically, our evaluation encompasses all benchmarks in NPBench that are under 40 lines of code, have parallelism present, do not use any data types/functions unsupported by APPy, and pass the validation test, which resulted in a set of 20 kernels.

NPBench comprises a set of NumPy code samples representing a wide variety of HPC applications and includes benchmarking tools to compare the performance of different NumPy-accelerating compilers (e.g. Numba etc). For each application, NPBench has a baseline NumPy implementation written in idiomatic Python/NumPy, which may use explicit loops, tensor operators, or both. Additionally, it includes implementations derived from the NumPy version using various compiler frameworks. As of now, it already includes compiler frameworks such as Numba and DaCe [26] (with both CPU and GPU backends), and we added APPy and JAX as additional frameworks.

6.1 Code Adaptation

Table 1 uses source lines of code (SLOC) as a metric to measure how much code adaptation is required for each framework. The SLOC count is obtained using the `pygount` tool. The code adaptation process for each framework is then summarized in the following sections.

6.1.1 Code Adaptation for APPy. The APPy versions of these kernels are derived from the baseline NumPy version with three modifications in most cases: 1) the tensor expressions are updated to use sliced index notation (e.g. `A[:] + B[:]` becomes `A[:N] + B[:N]`); 2) each parallel loop and tensor expression is annotated with pragmas, and the `le` clause (small tensor optimizations) is used when applicable; 3) the kernel function is decorated with `@appy.jit`. We do not annotate matmul operators used in the kernels and delegate them to library implementations.

For three kernels (`softmax`, `spmv` and `azimint_naive`), we also rewrite some tensor operators using loops due to limitations of our tensor-oriented model (e.g. multiple reduction operators will not be fused with our current operator fusion approach but a loop-based implementation would fuse them). Additionally, reduction sub-expressions are extracted out to form separate statements due to current implementation limitations, which caused a slight increase in additional lines of code for benchmarks such as `gesummv` and `cholesky` in table 1.

6.1.2 Code Adaptation for JAX. Due to the language design constraints of JAX, creating the JAX versions of the kernels requires more significant changes to the original NumPy versions compared to other frameworks. These changes arise from three major JAX restrictions: 1) in-place array updates must use the `at` idiom; 2) control flows, such as for loops, must be rewritten to structured forms using `jax.lax.fori_loop`. Otherwise, the compilation would take too long (e.g., hours for large loops) because JAX unrolls Python loops by default; 3) array slices with dynamic sizes are not supported. When creating the JAX versions of the kernels, we attempt to convert as many Python loops as possible to JAX structured loops and rewrite array slices with dynamic sizes using `where` if the slice size depends on the loop index variable.

6.1.3 Code Adaptation for Other Frameworks. Code adaptation for CuPy is the most straightforward, as it serves as an almost seamless drop-in replacement for NumPy. The only necessary change is to replace `import numpy as np` with `import cupy as np`. For Numba, two changes are required: 1) the kernel function should be decorated with `@numba.jit`; 2) if a loop can be parallelized, modify it to use `numba.prange` instead of `range`. Similar to Numba, DaCe kernels are also decorated with `@dace.program` for JIT compilation. Additionally, DaCe mandates that function arguments be type-annotated, including their shapes, which must be either integer constants or symbols, e.g. `dace.float64[N, M]`. These shape variables are defined similarly to APPy dimension variables. Notably, DaCe will automatically discover for loops that can be parallelized.

6.2 Performance Results

We evaluate the performance using the NPBench's benchmarking functionality, with the same input sizes as in the NPBench paper [25], such that each benchmark is run for approximately 1 second. By default, NPBench runs each benchmark 10 times and reports the median runtime[5]. The test machine used for results presented in this subsection is a Ryzen 7 5800X 8-Core CPU and a RTX 3090 GPU[6]. The operating system is Ubuntu 18.04.6 LTS and the CUDA version is 12.1. We use CPython version 3.9.16 as part of an Anaconda 3 environment. The NumPy version is 1.26.2, Numba version is 0.58.1, CuPy version is 12.2.0, JAX version is 0.4.23, DaCe version is 0.14.4 and Triton version is 2.1.0 (installed via torch 2.1.2).

Fig. 4 shows the performance results. The NumPy column on the right shows the absolute execution time, while the columns for the other frameworks show their speedup (up arrow) or slowdown (down arrow) relative to NumPy. On average (using geometric means), APPy achieves 3.1× speedup over DaCe-GPU (up to 11×), 18.8× speedup over JAX (with

[5]One exception is that kernel `gemver` is run only 8 times with JAX due to an out-of-memory error when attempting to run it 10 times.

[6]Additional results using different hardware are provided in section 6.3.

Benchmark	Contains Loops	Contains Tensor Expressions	APPy	DaCe	JAX	CuPy	Numba	NumPy
azimnaiv	Yes (P)	Yes	+8	+8	+3	+0	+2	11
cholesky	Yes	Yes	+7	+3	+7	+0	+2	10
covariance	Yes (P)	Yes	+7	+4	+8	+0	+2	8
fdtd_2d	Yes	Yes	+5	+4	+1	+0	+2	9
floyd_warshall	Yes	Yes	+3	+3	+5	+0	+3	5
gemm	No	Yes	+4	+4	+1	+1	+2	4
gemver	No	Yes	+7	+6	+2	+0	+2	6
gesummv	No	Yes	+10	+4	+1	+0	+2	3
go_fast	Yes (P)	No	+4	+3	+1	+0	+2	6
gramschmidt	Yes (P)	Yes	+5	+3	+15	+0	+2	12
hdiff	No	Yes	+8	+3	+2	+0	+2	22
heat_3d	Yes	Yes	+4	+3	+1	+0	+2	18
jacobi_2d	Yes	Yes	+4	+3	+1	+0	+2	8
softmax	No	Yes	+11	+9	+2	+0	+5	6
spmv	Yes (P)	Yes	+7	+8	+9	+0	+2	8
symm	Yes (P)	Yes	+5	+3	+16	+0	+1	10
syr2k	Yes (P)	Yes	+6	+4	+4	+0	+2	8
syrk	Yes (P)	Yes	+6	+4	+4	+0	+2	7
trisolv	Yes	Yes	+5	+3	+5	+0	+2	5
trmm	Yes (P)	Yes	+5	+3	+12	+0	+2	7

Table 1. Measure the amount of code adaptation using source lines of code (SLOC) as a metric. The column labeled "Contains Loops" indicates whether the benchmark contains explicit loops in its NumPy reference implementation, and "(P)" denotes that at least one loop is parallelizable. The column "Contains Tensor Expressions" indicates whether the NumPy reference implementation employs any (non-scalar) tensor expressions. The "NumPy" column displays the number of lines of code for the benchmark, while all other columns indicate their additional SLOC relative to the NumPy version. Note that pragma lines are included in the SLOC count for APPy.

JIT) and 30× speedup over CuPy. Comparing to the CPU frameworks, APPy achieves 30× speedup over NumPy, and 8.3× speedup over Numba.

6.2.1 Comparison with DaCe-GPU. Upon inspecting the generated CUDA code from DaCe, we identified two potential inefficiencies: 1) a sequential loop is generated when writing to an array slice, for instance, `C[i,:i+1] = ...;` 2) parallel reduction is not supported, which is utilized in the kernels `go_fast` and `azimint_naive`. APPy, with appropriate annotations, effectively parallelizes both cases. For a subset of benchmarks exclusively using operators, such as `gesummv`, `floyd_warshall`, and `gemver`, DaCe is also slower than APPy. This difference may be due to the fact that APPy is more effective in fusing these operator patterns. Moreover, in benchmarks like `softmax` and `cholesky`, their APPy implementations benefit from the `le` pragma, leveraging prior knowledge of small dimensions to enhance data locality. On the other hand, DaCe and APPy exhibit comparable performance for stencil kernels, including `jacobi_2d`, `heat_3d`, and `hdiff`.

6.2.2 Comparison with JAX-GPU. JAX is significantly slower compared to APPy or DaCe for kernels written with explicit loops that can be parallelized, such as `covariance`, `npgofast`, `spmv`, `symm`, and `syrk`, among others. In JAX, a `fori_loop` is compiled but not parallelized, even if it is parallelizable. For a number of benchmarks written solely using operators, JAX is even slower than CuPy, despite the fact that it compiles and fuses operators. This could be due to sub-optimal code generation of the fused operators in JAX. In the case of `spmv`, JAX is > 100× slower than the NumPy baseline because the original NumPy reference implementation uses arrays of dynamic shapes (depending on the row), which has to be rewritten to an implementation using `where` operators in JAX since dynamic shapes are not supported by JAX. Further, the `where` based implementation has to load the entire data and column arrays in every iteration and then select only a few elements. This approach is highly inefficient and fails to leverage the advantages of sparse representations. However, JAX demonstrates significant performance improvement over all other frameworks for `azimnaiv`. This could be attributed to the fact

Benchmarks	appy	dace_gpu	jax	cupy	numba	numpy
Total	↑30.0	↑9.8	↑1.6	↑1.1	↑3.6	
azimnaiv	↑4.5 (1)	↑7.4	↑17.3 (2)	↑2.6	↑4.2 (10)	0.11 s
cholesky	↓1.4	↓14.7	↓20.4 (1)	↓50.6	↑17.7	0.45 s
covarian	↑1.6 (8)	↑1.2	↓1.1 (14)	↓2.1	↓1.1	49.24 ms
fdtd_2d	↑39.5 (1)	↑38.5	↑27.2 (2)	↑14.5	↑5.2 (1)	2.43 s
floydwar	↑56.5	↑18.3	↑24.9 (2)	↑16.8 (1)	↑12.9 (3)	1.60 s
gemm	↑2.0	↑2.0	↑1.9	↑2.0 (15)	↓1.1 (1)	90.60 ms
gemver	↑97.0	↑56.1 (1)	↑11.7	↑38.0	↓1.5	0.85 s
gesummv	↑100.0	↑41.2	↑6.8	↑42.8	↓1.1	0.31 s
gramschm	↑9.4	↑5.7	↓8.2 (2)	↓24.6 (7)	↑1.8	0.44 s
hdiff	↑98.7	↑113.0	↑34.2 (4)	↑35.4	↑1.2 (2)	0.36 s
heat3d	↑362.0	↑352.0	↑332.0 (6)	↑40.4	↑28.9 (1)	5.47 s
jacobi2d	↑210.0	↑176.0	↑160.0	↑28.4	↑3.0	3.14 s
npgofast	↑38.3	↑2.8	↑1.4	↑1.0	↑1.2 (2)	0.15 s
softmax	↑214.0	↑43.6	↑14.5	↑61.2 (3)	↓1.0	0.70 s
spmv	↑207.0 (8)	↓30.5	↓160.0	↓51.0 (2)	↑32.4 (1)	0.32 s
symm	↑37.5	↑19.9	↓11.7	↓33.5 (2)	↑15.7	3.76 s
syr2k	↑127.0 (15)	↑30.5	↓6.9 (1)	↓14.1 (9)	↑6.0 (1)	6.18 s
syrk	↑100.0 (1)	↑25.6	↓17.7 (1)	↓18.2 (3)	↑3.7 (2)	2.36 s
trisolv	↓3.1	↓5.1	↓3.8	↓17.3	↑1.7	57.29 ms
trmm	↑78.2 (1)	↑63.3	↓28.3	↓46.9 (6)	↑14.3 (1)	1.59 s

Figure 4. Performance results on a set of NPBench benchmarks. All frameworks show their speedup relative to NumPy, except NumPy itself shows its absolute execution time in seconds. Up arrows indicate performance improvement, and down arrows indicate performance slowdown.

that `azimnaiv` presents additional optimization opportunities when the outer loop is unrolled[7], for instance, allowing for the saving of re-computation of a sub-expression.

6.2.3 Comparison with CuPy.

The CuPy versions of the kernels often still require explicit loops, which executes sequentially in the Python interpreter, while with APPy such loops are parallelized when possible. One might wonder why CuPy is even slower than NumPy, which has identical code but uses CPU-based implementations instead of GPU-based implementations. This is because each loop iteration typically processes relatively small sized data in our input setting. Due to the small data size, processing them on the CPU isn't necessarily slower than on the GPU. When the program can be entirely expressed using tensor operators, such as in case of `gemm`, `gemver`, `gesummv`, and `softmax`, CuPy typically achieves significant speedup over NumPy.

6.2.4 Comparison with NumPy and Numba.

GPUs are known to be more efficient than CPUs when it comes to data parallelism. NumPy runs the code sequentially in the Python interpreter, although each operator is backed by an optimized native code implementation. Still it's limited by the sequential execution and the interpreter overhead. Compared to NumPy, Numba applies Python-to-native-code compilation, automatic optimizations and parallelization (with `prange`), but it parallelizes the code only on the CPU. In contrast, APPy compiles the kernel to native GPU execution.

However, we note that for `trisolv`, APPy is significantly slower than NumPy (3.1× slower) and Numba. In fact, all the frameworks employ the same parallelization strategy for `trisolv`, i.e. the outer loop is sequentialized, and the loop body contains a dot product that exhibits parallelism and gets parallelized. In case of APPy, the outer loop is annotated with `#pragma sequential for` to ensure that only one worker thread is launched. This limited parallelism within one worker thread could be one of the reasons why APPy is 3.1× slower than NumPy baseline for `trisolv`.

6.3 Additional Performance Results

For completeness, we performed an additional performance evaluation on a different machine consisting of an AMD EPYC 7502 32-Core CPU and an NVIDIA A100-PCIE-40GB GPU. The package versions and benchmarking methodologies are identical with the previous evaluation reported in Section 6.2. DaCe-GPU is excluded from this additional evaluation due to compilation errors on this platform. There is also one data point missing for Numba due to a runtime error that was encountered in that case.

Fig. 5 shows the performance results. In general, the relative speedup f the GPU frameworks over the CPU frameworks roughly doubles compared to the results in Fig. 4. Notably, JAX has improved performance over APPy for `floydwar`, `fdtd_2d` and `heat3d`. This could be due to the fact that JAX's code generation is better optimized for the A100 hardware, compared to RTX 3090. However, the general performance trends are still similar to that of Fig. 4. On average APPy achieves 18.8× speedup over JAX (with JIT) and 34.6× speedup over CuPy. Comparing to the CPU frameworks, APPy achieves 65.7× speedup over NumPy, and 16.8× speedup over Numba.

[7]JAX unrolls Python loops by default.

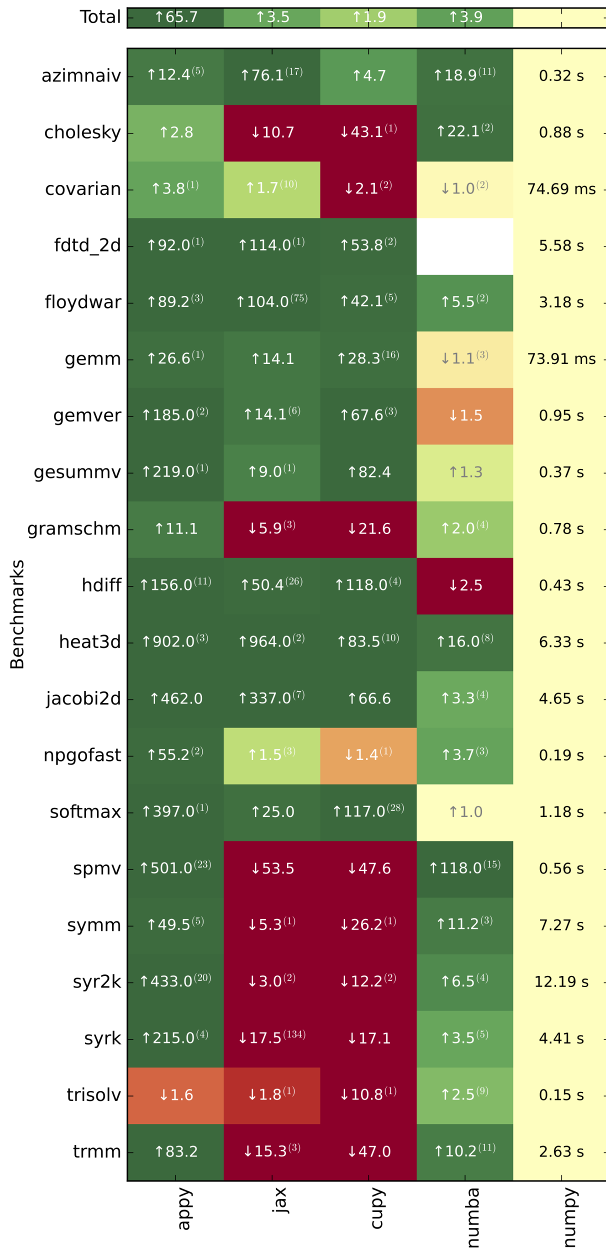

Figure 5. Additional performance results on the same set of benchmarks as Fig. 4 but using a different server with an AMD EPYC 7502 32-Core CPU and an NVIDIA A100-PCIE-40GB GPU. All frameworks show their speedup relative to NumPy, except NumPy itself shows its absolute execution time in seconds. Up arrows indicate performance improvement, and down arrows indicate performance slowdown.

7 Related Work

7.1 SIMT/SPMD-Based GPU Programming

General purpose GPU programming can be classified into 1) explicit parallel programming and 2) compiler directive-based programming. The former traditionally employs an extended version of C/C++ language that enables expressing parallelism in the style of single program multiple data (SPMD), and two prominent programming languages are CUDA [14] for NVIDIA hardware and OpenCL[22] which is more portable and widely supported among vendors. Combined with hardware-managed SIMD execution, this programming paradigm is referred to as single instruction multiple thread (SIMT)[14], where the same piece of user code is executed by multiple threads in parallel and threads within a thread block can communicate via on-chip shared memory. SPMD combined with hardware SIMD enables highly flexible programming interface and high performance at the same time, but this low-level GPU programming is inherently challenging, due to the complexity of managing thread synchronization and memory hierarchy optimization.

Numba [13] supports GPU programming by directly exposing the parallel execution model of the hardware in a way similar to CUDA-C and OpenCL, which still requires explicit parallel programming, although embedded in Python. Triton [24] language and compiler is a more recent effort in the realm of general purpose GPU programming. Its goal is to provide a simplified programming interface compared to CUDA or OpenCL, by letting the compiler automatically manages intra-block thread scheduling, memory coalescing and shared memory management. However, Triton still requires the programmer to manually partition the computation onto different program instances, selecting block sizes and launching the GPU kernels. In addition, optimizations such as operator fusion are still manual.

None of these approaches parallelize a sequential Python loop directly and support array/tensor operations as APPy does.

7.2 Directive-Based GPU Programming

Two popular directive-based programming models for GPUs are OpenMP [3] and OpenACC[21], which are based on C/C++/Fortran. OpenMP originally only targets at multicore CPUs, but has introduced new pragmas to target at GPUs since OpenMP 4 while OpenACC was originally designed for GPUs but has support for CPUs as well. They simplify the programming effort compared to CUDA and OpenCL by allowing annotating sequential loops with pragmas and the compiler generates the final parallel code. However, they still directly expose GPU's complex hierarchy of parallelism, and as a result, one has to grasp the GPU hardware execution model and use a complex set of directives to achieve maximum parallelism. [19] proposes a unified SIMD primitive

coupled with the Kokkos ecosystem that allows intrinsics-based vectorization on CPUs and GPUs. A feature called logical vector length (LVL) is also proposed to write code without considering underlying physical vector length. This is similar to the maximal vector length (MVL) in APPy. But the SIMD primitive in [19] is still built upon Kokkos' existing complex parallel programming concepts and requires the programmer to manage teams, threads etc. In contrast, APPy's directives are much higher level and let the programmer write just regular sequential code.

7.3 Library-Based GPU Acceleration

CuPy [16] is a NumPy/SciPy-compatible array library for GPU-accelerated computing with Python. Its interface is highly compatible with NumPy and SciPy and in most cases it can be used as a drop-in replacement. In addition to providing NumPy-compatible operators, CuPy also provides simple interface to quickly write custom element-wise and reduction CUDA kernels. Custom generic CUDA kernels can also be imported using CuPy's `RawKernel` module, which compiles and wraps CUDA C++ code to a Python function. PyTorch [18], TensorFlow [1] and JAX [6] are among the most popular machine learning frameworks that provide CuPy-like generic tensor computation APIs as well as machine learning building blocks. Legate [2] (cuNumeric) is another framework that aims to provide a distributed and accelerated drop-in replacement for the NumPy. It supports both single node CPU/GPU and transparent distributed acceleration across multi-node machines.

7.4 High-Level Language to GPU Compiler

`jit4GPU` (implemented under `unPython`) [7] is an early effort (2010) that just-in-time compiles Python/NumPy code to both CPU code (with OpenMP) and GPU code (AMD hardware-only). It also takes an annotation-based approach and requires the programmer to annotate parallel loops with `prange`. However, the paper does not document how it performs thread-level parallelization and parallel reduction does not appear supported as well. In terms of memory hierarchy optimization, `jit4GPU` does not generally perform operator fusion, due to lack of dependence analysis. `Loo.py` [12] employs a domain-specific language similar to the sliced index notation to express parallel kernels, and also provides schedules to specify code transformations. However it does not provide general programming interface to express more general programs, such as parallel reduction with indirect memory access. ALPyNA [11] proposes an automatic Python loop parallelization approach that splits the compilation and dependence analysis into an ahead-of-time (AOT) stage and a just-in-time (JIT) stage. However, its performance evaluation does not demonstrate consistent performance improvement over Numba (can be even 3× slower than Numba). In addition, it targets at only loops, not tensor expressions. DaCe[26] compiles and parallelizes Python code to both CPU and GPU execution, using a stateful dataflow multiGraphs (SDFG) data-centric intermediate representation, on which automatic or manual transformations can be performed. Parallel loops are either manually specified as `dace.map` iterator, or automatically inferred by a transformation pass via dependence analysis. However we have observed that certain parallelizable patterns are not parallelized by DaCe which could lead to sub-optimal performance. AutoMPHC [20] proposes an approach to automatic ahead-of-time (AOT) parallelization and optimization of sequential Python programs for execution on distributed heterogeneous platforms, based on extensions to the polyhedral framework that unify user-written loops and tensor operators. However its intra-node parallelization strategy of explicit loops is limited in that an equivalent pre-defined tensor operator must exist in order for the loop to be parallelized, nor can it parallelize loops with indirect memory accesses.

8 Conclusion

In this work, we present APPy, which allows users to parallelize their sequential Python loops and tensor assignments on GPUs using compiler directives. We present the design and implementation of APPy, including code generation and automatic compiler optimizations. We evaluate the performance of APPy using 20 kernels from scientific computing and demonstrate significant speedup over a state-of-the-art library CuPy (30× on average) and three Python compilers Numba (8.3× on average), DaCe-GPU (3.1× on average) and JAX-GPU (18.8× on average).

9 Acknowledgments

This work is partly based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Agreement Nos. HR0011-17-S-0055 and HR0011-20-9-0020. This work is also partly based upon work supported by the Office of the Director of National Intelligence (ODNI), Intelligence Advanced Research Projects Activity (IARPA), through the Advanced Graphical Intelligence Logical Computing Environment (AGILE) research program, under Army Research Office (ARO) contract number W911NF22C0083. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the ODNI, IARPA, or the U.S. Government. We thank the reviewers for their helpful feedback and suggestions to improve the paper. We thank Alexandros Ziogas for using his script when creating the performance results graph, and his help with getting familiar with NPBench and DaCe.

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. https://www.tensorflow.org/ Software available from tensorflow.org.

[2] Michael Bauer and Michael Garland. 2019. Legate NumPy: Accelerated and Distributed Array Computing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '19)*. Association for Computing Machinery, New York, NY, USA, Article 23, 23 pages. https://doi.org/10.1145/3295500.3356175

[3] OpenMP Architecture Review Board. [n. d.]. OpenMP Programming API 5.2. https://www.openmp.org/spec-html/5.2/openmp.html

[4] André R Brodtkorb, Trond R Hagen, and Martin L Sætra. 2013. Graphics processing unit (GPU) programming strategies and trends in GPU computing. *J. Parallel and Distrib. Comput.* 73, 1 (2013), 4–13.

[5] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. {TVM}: An Automated {End-to-End} Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 578–594.

[6] Roy Frostig, Matthew Johnson, and Chris Leary. 2018. Compiling machine learning programs via high-level tracing. https://mlsys.org/Conferences/doc/2018/146.pdf

[7] Rahul Garg and José Nelson Amaral. 2010. Compiling Python to a Hybrid Execution Environment. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units* (Pittsburgh, Pennsylvania, USA) *(GPGPU-3)*. Association for Computing Machinery, New York, NY, USA, 19–30. https://doi.org/10.1145/1735688.1735695

[8] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. MIT Press. http://www.deeplearningbook.org.

[9] Charles R Harris, K Jarrod Millman, Stéfan J Van Der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J Smith, et al. 2020. Array programming with NumPy. *Nature* 585, 7825 (2020), 357–362.

[10] John L. Hennessy and David A. Patterson. 2017. *Computer Architecture, Sixth Edition: A Quantitative Approach* (6th ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[11] Dejice Jacob, Phil Trinder, and Jeremy Singer. 2019. Python programmers have GPUs too: automatic Python loop parallelization with staged dependence analysis. In *Proceedings of the 15th ACM SIGPLAN International Symposium on Dynamic Languages* (Athens, Greece) *(DLS 2019)*. Association for Computing Machinery, New York, NY, USA, 42–54. https://doi.org/10.1145/3359619.3359743

[12] Andreas Klöckner. 2014. Loo.Py: Transformation-Based Code Generation for GPUs and CPUs. In *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming* (Edinburgh, United Kingdom) *(ARRAY'14)*. Association for Computing Machinery, New York, NY, USA, 82–87. https://doi.org/10.1145/2627373.2627387

[13] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. 2015. Numba: A llvm-based python jit compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*. 1–6.

[14] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. 2008. Scalable Parallel Programming with CUDA: Is CUDA the Parallel Programming Model That Application Developers Have Been Waiting For? *Queue* 6, 2 (mar 2008), 40–53. https://doi.org/10.1145/1365490.1365500

[15] Wei Niu, Jiexiong Guan, Yanzhi Wang, Gagan Agrawal, and Bin Ren. 2021. DNNFusion: Accelerating Deep Neural Networks Execution with Advanced Operator Fusion. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 883–898. https://doi.org/10.1145/3453483.3454083

[16] Ryosuke Okuta, Yuya Unno, Daisuke Nishino, Shohei Hido, and Crissman Loomis. 2017. CuPy: A NumPy-Compatible Library for NVIDIA GPU Calculations. In *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Thirty-first Annual Conference on Neural Information Processing Systems (NIPS)*. http://learningsys.org/nips17/assets/papers/paper_16.pdf

[17] John D. Owens, Mike Houston, David Luebke, Simon Green, John E. Stone, and James C. Phillips. 2008. GPU Computing. *Proc. IEEE* 96, 5 (2008), 879–899. https://doi.org/10.1109/JPROC.2008.917757

[18] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019).

[19] Damodar Sahasrabudhe, Eric T. Phipps, Sivasankaran Rajamanickam, and Martin Berzins. 2020. A Portable SIMD Primitive Using Kokkos for Heterogeneous Architectures. In *Accelerator Programming Using Directives*, Sandra Wienke and Sridutt Bhalachandra (Eds.). Springer International Publishing, Cham, 140–163.

[20] Jun Shirako, Akihiro Hayashi, Sri Raj Paul, Alexey Tumanov, and Vivek Sarkar. 2022. Automatic Parallelization of Python Programs for Distributed Heterogeneous Computing. In *Euro-Par 2022: Parallel Processing*, José Cano and Phil Trinder (Eds.). Springer International Publishing, Cham, 350–366.

[21] OpenACC specification authors. [n. d.]. OpenACC Programming API 3.2. https://www.openacc.org/sites/default/files/inline-images/Specification/OpenACC-3.2-final.pdf

[22] John E. Stone, David Gohara, and Guochun Shi. 2010. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *Computing in Science and Engineering* 12, 3 (2010), 66–73. https://doi.org/10.1109/MCSE.2010.69

[23] Christopher Stover and Eric W. Weisstein. [n. d.]. Einstein Summation. https://mathworld.wolfram.com/EinsteinSummation.html

[24] Philippe Tillet, H. T. Kung, and David Cox. 2019. Triton: An Intermediate Language and Compiler for Tiled Neural Network Computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages* (Phoenix, AZ, USA) *(MAPL 2019)*. Association for Computing Machinery, New York, NY, USA, 10–19. https://doi.org/10.1145/3315508.3329973

[25] Alexandros Nikolaos Ziogas, Tal Ben-Nun, Timo Schneider, and Torsten Hoefler. 2021. NPBench: A Benchmarking Suite for High-Performance NumPy. In *Proceedings of the ACM International Conference on Supercomputing* (Virtual Event, USA) *(ICS '21)*. Association for Computing Machinery, New York, NY, USA, 63–74. https://doi.org/10.1145/3447818.3460360

[26] Alexandros Nikolaos Ziogas, Timo Schneider, Tal Ben-Nun, Alexandru Calotoiu, Tiziano De Matteis, Johannes de Fine Licht, Luca Lavarini, and Torsten Hoefler. 2021. Productivity, Portability, Performance: Data-Centric Python. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (St. Louis, Missouri) *(SC '21)*. Association for Computing Machinery, New York, NY, USA, Article 95, 13 pages. https://doi.org/10.1145/3458817.3476176

A Implementation Details

This section gives more details that are not covered in the implementation overview. Fig. 6 shows the end-to-end workflow using the `group-by-sum` as an example.

A.1 Algorithm Notation

All of the code generation and analysis algorithms are based on AST tree traversals [8], so in the pseudocode we often use the term *node* to represent an AST node. We assume the existence of utility functions that create an AST object of a given type and accept either strings or AST objects as arguments. These utility functions are named $new\langle Kind\rangle$, e.g. *newCall*, *newAssign* etc. We also assume that an AST object has the associated attributes for its kind. For example an assignment object will have two attributes *target* and *value*, to represent the storing target and the value to be stored respectively.

Algorithm 2 Top-level compilation process. $func$ is the function object to be compiled. A new function is returned.

```
1: function CODEGEN(func)
2:     ast ← getAST(func)
3:     m ← newModule()
4:     ast ← HIGHLEVELTRANSFORM(ast)
5:     for node in depth-first-traversal of ast do
6:         if node is a top-level parallel for-loop then
7:             devFunc ← GENDEVICECODE(node)
8:             hostCode ← GENHOSTCODE(node)
9:             m.body.append(devFunc)
10:            replace node with hostCode
11:    append ast to m, dynamically load module m and
       return the new function from m.
```

A.2 High-Level APPy To "Assembly" APPy Transformation

Alg. 3 describes the high-level analysis and transformations. The first pass is to transform the AST of the input function to a normalized form, paving the way for downstream analysis and transformations. This includes rewriting `+=` operator into a regular assignment, rewriting `range(N)` to `range(0, N, 1)` etc. The next pass is to link user annotated pragmas with the corresponding statement or loop, such that later passes can check if a loop/statement is annotated, e.g. parallel or simd etc. Line 14 inserts synchronization statements to guarantee a worker executes its code sequentially even when mapped to multiple asynchronous threads, and is described in more detail in section A.3. Line 11 guarantees correct code generation for array assignments such as `A[1:N] = A[0:N-1]`, by splitting the assignment, first storing to a temporary array, and then store to `A[1:N]` in another statement.

[8]The actual implementations are based on the Python `ast.NodeTransformer` module.

Line 17 converts loops with the `simd` directive to a strip-mined form, as shown in Alg. 6; and line 19 converts tensor expressions into strip-mined loops and is covered in detail in section A.4. Each of these steps has a time complexity of $O(V)$ where V is the number of nodes in the AST. After this sequence of high level transformations, the code is now ready to be converted to Triton code.

Algorithm 3 High-Level transformations performed to the input APPy function. The resulting function is a "assembly APPy" code.

```
1: function HIGHLEVELTRANSFORM(ast)
2:     ast ← NORMALIZE(ast)
       ▷ Link a pragma to the node following it
3:     pragma ← None
4:     for node in depth-first-traversal of ast do
5:         if node is a pragma comment then
6:             pragma ← node.value
7:         else
8:             if pragma ≠ None then
9:                 node.pragma ← pragma
10:                pragma ← None
       ▷ Ensures correct codegen for like "A[1:N] = A[0:N-1]"
11:    CREATETEMPARRAY(ast)
12:    for node in depth-first-traversal of ast do
13:        if node is a parallel for loop then
14:            INSERTSYNC(node)
15:    for node in depth-first-traversal of ast do
16:        if node is a for-loop annotated with simd then
17:            CONVERTSIMDLOOP(node)
18:        if node is a tensor expression assignment then
19:            CONVERTTENSORASSIGN(node)
20:    return ast
```

Algorithm 4 Algorithm to insert synchronization statements in the worker code.

```
1: function INSERTSYNC(node)
2:     used ← new set
3:     for child in depth-first-traversal of node do
4:         if child is an array load then
5:             used.add(child.tensorName)
6:     for child in depth-first-traversal of node do
7:         if child is an array store then
8:             if child.tensorName in used then
9:                 s1 ← new synchronization call
10:                s2 ← new synchronization call
11:                replace child with (s1, child, s2)
```

A.3 Parallelize A Worker

A vector statement is executed by a worker in a vectorized fashion in lockstep. There are multiple possible approaches

```python
1 @appy.jit(auto_simd=True)
2 def group_by_sum(data, ls, cs, M, N):
3     #pragma parallel for
4     for i in range(M):
5         l = ls[i]
6         #pragma atomic
7         cs[l, :N] += data[i, :N]
```

(a) Original input program.

```python
1 @appy.jit(auto_simd=True)
2 def group_by_sum(data, ls, cs, M, N):
3     #pragma parallel for
4     for i in range(0, M, 1):
5         l = ls[i]
6         for _var0 in range(0, N, MVL):
7             _var1 = vidx(_var0, MVL, N)
8             #pragma atomic
9             cs[l, _var1] += data[i, _var1]
```

(b) After lowering to "assembly" APPy. The tensor/array assignment is converted to a strip-mined loop.

```python
1 import triton
2 import triton.language as tl
3
4 @triton.jit
5 def _kernel(data, ls, cs, M, N, data_stride0, \
6               ls_stride0, cs_stride0, MVL: tl.constexpr):
7     i = tl.program_id(0) * 1
8     l = tl.load(ls, i)
9     for _var0 in range(0, N, MVL):
10        _var1 = _var0 + tl.arange(MVL)
11        tl.atomic_add(cs + l*cs_stride0 + _var1, \
12                      tl.load(data + i*data_stride0 + _var1, \
13                              mask=_var1 < N), \
14                      mask=_var1 < N)
15
16 def group_by_sum(data, ls, cs, M, N):
17     grid = M // 1
18     MVL = 128
19     _kernel[grid](data, ls, cs, M, N, data.stride(0), \
20                   ls.stride(0), cs.stride(0), MVL)
```

(c) The original parallel for-loop is replaced by the generated host code, and a Triton kernel function `_kernel`.

Figure 6. Compilation workflow using `group-by-sum` as an example. Section A.2-A.4 and A.5 cover the transformations from (6a) to (6b), and from (6b) to (6c), respectively.

Algorithm 5 Algorithm to allocate temporary arrays for array assignments that may have data dependences.

```
1: function CREATETEMPARRAY(ast)
2:     for node in depth-first-traversal of ast do
3:         if node is a tensor assignment then
4:             tar ← node.target
5:             if tar is used in node.value then
                   ▷ Allocate a new temporary array "tmp"
6:                 tmp ← getNewVar()
7:                 alloc ← newCall("empty_like", tar)
8:                 s1 ← newAssign(tmp, alloc)
9:                 s2 ← deepcopy(node)
                   ▷ Update assignment target to "tmp"
10:                node.target.tensorName ← tmp
                   ▷ Store "tmp" to the original target
11:                s2.value = node.target
12:                replace node with (s1, node, s2)
```

Algorithm 6 Code generation algorithm to handle `simd` directive.

```
1: function CONVERTSIMDLOOP(loop)
2:     i ← loop.index
3:     N ← loop.upperBound
4:     vi ← newCall("vidx", "MVL", N)
5:     s ← newAssign(i, vi)
6:     loop.body.insert(0, s)
       ▷ Rewrite "s = s + ..." to "s = s + sum(...)" etc
7:     for child in depth-first-traversal of loop do
8:         if child is a reduction assignment then
9:             op ← getReduceOpType(child)
10:            val ← newCall(op, getReduceVal(child))
11:            tar ← child.target
12:            newVal ← newBinOp(op, tar, val)
13:            newChild ← newAssign(tar, newVal)
14:            replace child with newChild
```

to implement a worker on an actual hardware like a GPU. The first approach is to map a worker to a single hardware SIMD execution unit, e.g. a warp (or wavefront). The synchronized execution of a warp naturally maps to the vectorized execution of tensor expressions. However, a warp typically has limited parallelism (lanes), e.g. 32 and 64. These SIMD units are not designed to have very long lanes to achieve efficiency when there are branch divergences. The downside of mapping a worker to a warp is that synchronization is limited to warp scope, which could in turn limit parallelism, e.g. for parallel reductions. A more flexible approach is to map a worker to a thread block, which support larger synchronization scope by allowing inter-warp communication. However, this flexibility comes with its own complexity: different warps may execute asynchronously, which may violate the vectorized semantics. An easy solution is to insert synchronization barrier after each original APPy statement that involves a memory read or write, to enforce synchronized execution (note that such synchronization is automatically implied if a worker is mapped to a single warp). An alternative approach that we implement which could be more efficient when the program has more memory loads than writes is to insert a synchronization before and after every memory store, shown in Alg. 4. This makes sure that 1) before a store operation, all previous memory loads and stores have completed; 2) and the store operation itself must

have completed before future memory loads or stores are executed. Therefore the vectorized execution semantics are preserved. In addition, we can apply a simple optimization: if the array being stored is never used in any other statements, the synchronizations before and after it can be skipped, since this write cannot possibly have data dependencies with other reads, since APPy disallows pointer aliasing.

Algorithm 7 Generate a loop nest from a tensor assignment.

```
1: function convertTensorAssign(node)
2:     m ← newModule()
3:     p ← m
4:     slice2var ← new map
       ▷ Generate the loop nest structure
5:     for dim in getSlicings(node.pragma) do
6:         loop ← genOneLoop(dim)
7:         p.body.append(loop)
8:         p ← loop                         ▷ loop is new parent
       ▷ Update the assignment to use new index vars
9:     for child in depth-first-traversal of node do
10:        if child is a slicing then
11:            if child in slice2var then
12:                idx ← slice2var[child]
13:            else
                   ▷ Deal with shifted index
14:                for s in slice2var.keys() do
                       ▷ Requires symbolic computation
15:                    off ← symbolicSubtract(child, s)
16:                    if off is a constant then
17:                        idx ← newAdd(s, off)
18:            replace child with idx
       ▷ Update the value for reductions
19:    if node has reduction then
20:        op ← getReduceOp(node)
21:        val ← newBinOp(op, node.target, node.value)
22:        node.value ← val
       ▷ Insert updated statement into the innermost loop
23:    p.body.append(node)
24:    return m.body
```

A.4 Lowering Tensor Assignments To Loops

Alg. 7 shows the code generation algorithm to generate a loop from a tensor assignment. An index variable and a loop is created for each unique dimension, and the slicings in the original statement is replaced by these new index variables. The depth of the loop nest equals the number of unique dimensions in the tensor assignment, with the original statement sits in the innermost loop. During the code generation process, a form of operator fusion, which we call *reduction-bounded operator fusion* is automatically performed, as described in more detail in section 5.1.4. Another optimization is that thread synchronization is not necessary within loops generated from tensor assignments. This optimization is reflected in Alg. 3 that *insertSync* is performed before *convertTensorAssign*.

A.5 "Assembly" APPy to Triton Code Generation

The Triton code generation algorithm generates a Triton kernel function (Alg. 9), together with its corresponding host code (Alg. 10), from a parallel APPy for-loop. The parallel for-loop can be a nested parallel loop, in which case a multi-dimensional thread block will be launched (determined in sub-procedure *inspectSubLoops*). To generate a device function, Alg. 9 first creates an empty function decorated with *@triton.jit*, inserts the *program_id* statements, and then it converts the loop body into a Triton kernel and insert the transformed statements (Alg. 12) into the new function. To generate host code, Alg. 10 creates an N-dimensional grid, where "N" is determined in *inspectSubLoops*, and launches the kernel by passing the actual arguments, including the block size (`appy.MVL`). A good block size depends on the hardware and the kernel. For our implementation and evaluation on an NVIDIA GPU, we generated some auto-tuning code that automatically selects the best block size (that lead to the best runtime) from 4 possible options, 128, 256, 512

Algorithm 8 Generate a single loop layer or a statement for a dimension.

```
1:  function genOneLoop(dim)
2:      start, end ← getRangeFromSlice(dim)
3:      i ← getNewVar()
        ▷ Skip generating a loop for small dimensions
4:      if dim.le ≠ None then
5:          vi ← newCall("vidx", i, dim.le, end)
6:          m ← newModule()
7:          m.body.append(vi)
8:          return m
9:      else
10:         if dim.simd == true then
11:             vi ← newCall("vidx", i, "MVL", end)
12:             loop ← newLoop(i, start, end, "MVL")
13:             loop.body.append(vi)
14:         else
15:             loop ← newLoop(i, start, end, "1")
16:         if dim.parallel == true then
17:             loop.pragma ← "parallel for"
18:         else
19:             return loop
```

and 1024. The best block size is cached for later invocations of the kernel.

Algorithm 9 Generate a Triton GPU kernel from a parallel loop. It calls Alg. 12 as a sub-procedure.

```
1:  function genDeviceCode(node)
2:      params ← extractKernelParams(node)
3:      devFunc ← newFunction(params)
4:      addDecorator(devFunc, "triton.jit")
5:      inspectSubLoops(node)
6:      axis ← 0
7:      for ploop in node.parloops do
8:          pid ← newCall("tl.program_id", axis)
9:          i ← ploop.target
10:         s ← newAssign(i, newMul(pid, ploop.step))
            ▷ Example: i = tl.program_id(0) * step
11:         devFunc.body.append(s)
12:         axis ← axis + 1
13:     for child in node do
14:         newChild ← toTritonStmts(child)
15:         devFunc.body.append(newChild)
16:     return devFunc
```

Algorithm 10 Generate host code to launch the Triton kernel. This includes determining the dimension of the grid and passing the arguments etc.

```
1:  function genHostCode(node)
2:      args ← extractKernelArgs(node)
3:      args ← setMVL(args)
4:      grid ← empty list
5:      for ploop in node.parloops do
6:          up ← ploop.upperBound
7:          lo ← ploop.lowerBound
8:          st ← ploop.step
9:          nblocks ← newCall("ceil_div", newSub(up, lo), st)
10:         grid.append(nblocks)
11:     return newKernelCall(grid, args)
```

Algorithm 11 Inspect the parallel sub-loops for a top-level parallel loop.

```
1:  function inspectSubLoops(node)
2:      parloops ← empty set
3:      for child in depth-first-traversal of node do
4:          if child is a parallel loop then
5:              parloops.add(child)
6:      node.parloops ← parloops
```

B Limitations

APPy has a number of limitations that we hope to address in the future work.

B.1 Language Rules

- APPy uses syntactical difference to denote the dimensionality of the variables. This can be viewed as a form of type annotations. For example, `a` or `A[0]` denotes a scalar variable while `A[:]` and `B[:, :]` would denote a 1D and 2D tensor, respectively.
- To make alias of an array or an array slice, one must use a built-in function, e.g. `a = appy.alias(A[:])`, since `a = A[:]` is a memory load statement in APPy.
- When loading a slice of an array into a scalar, the size of the slice must be specified with a `le(CONST)` property.
- Arrays cannot have overlap in their memory regions.

B.2 Unsupported Functions

APPy supports slicing an array, memory loads and stores, basic mathematical functions on scalars or tensors, and broadcasting a dimension. Operations unsupported include reshaping, concatenating, sorting arrays, and linear algebra routines. One can either use these functions just in the Python interpreter (no compilation), or create their custom implementations using the basic operations recognizable by APPy.

Algorithm 12 Convert APPy statements to Triton statements.

```
1: function toTritonStmts(node)
2:   if node is an assignment then
3:     node.value ← toTritonStmts(node.value)
4:     node.target.parent ← node
5:     if node.target is an array store then
6:       node ← toTritonStmts(node.target)
7:     else
8:       node.target ← toTritonStmts(node.target)
9:   else if node is an array subscript then
10:    offset ← generate offsets from node.indices
11:    mask ← generate mask from node.indices
12:    ptrs ← newAdd(node.tensorName, offset)
13:    if node.ctx is a load then
14:      node ← newCall("tl.load", ptrs, mask)
15:    else
16:      val ← node.parent.value
17:      op ← "tl.store"
18:      if node is an atomic update then
           ▷ op will become "tl.atomic_add" etc
19:        op ← getReduceOpType(node.parent)
20:        val ← getReduceVal(node.parent)
21:      node ← newCall(op, ptrs, val, mask)
22:  else if node is a function call then
23:    update node.funcName to the Triton version
24:  else
25:    for child in iterChildNodes(node) do
26:      newChild ← toTritonStmts(child)
27:      node.⟨childFieldName⟩ ← newChild
28:  return node
```

B.3 Unsupported Annotations

A tensor dimension cannot have conflicting properties in different sub-expressions. For example, for tensor expression `A[:M, :N] / sum(A[:M, :N], axis=1)[:, None]`, dimension `:N` is a reduction dimension in the `sum` sub-expression, but a non-reduction dimension for the divide operator. The statement can be separated into multiple new statements (with their own annotations) to resolve the conflict.

In addition, APPy currently requires the result of a reduction operator not be used as a sub-expression, and must be assigned to a temporary tensor. For example, to sum up two matrix vector multiplications (mv), as in kernel `gesummv` in Listing 7, one must store the results of both mvs to two different temporary tensors, and then create a third statement to add these two temporary tensors [9].

Received 12-NOV-2023; accepted 2023-12-23

[9]This requirement is due to the complexity of splitting an expression and its pragmas automatically into sub-expressions and their annotations.