C2TACO: Lifting Tensor Code to TACO

José Wesley de Souza Magalhães
jwesley.magalhaes@ed.ac.uk
University of Edinburgh
UK

Jackson Woodruff
J.C.Woodruff@sms.ed.ac.uk
University of Edinburgh
UK

Elizabeth Polgreen
elizabeth.polgreen@ed.ac.uk
University of Edinburgh
UK

Michael F. P. O'Boyle
mob@inf.ed.ac.uk
University of Edinburgh
UK

Abstract

Domain-specific languages (DSLs) promise a significant performance and portability advantage over traditional languages. DSLs are designed to be high-level and platform-independent, allowing an optimizing compiler significant leeway when targeting a particular device. Such languages are particularly popular with emerging tensor algebra workloads. However, DSLs present their own challenge: they require programmers to learn new programming languages and put in significant effort to migrate legacy code.

We present C2TACO, a synthesis tool for synthesizing TACO, a well-known tensor DSL, from C code. We develop a guided enumerative synthesizer that uses automatically generated IO examples and source-code analysis to efficiently generate dense tensor algebra code. C2TACO is able to synthesize 95% benchmarks from a tensor benchmark suite, outperforming an alternative neural machine translation technique, and demonstrates substantially higher levels of accuracy when evaluated against two state-of-the-art existing schemes, TF-Coder and ChatGPT. Our synthesized TACO programs are, by design, portable achieving significant performance improvement when evaluated on a multi-core and GPU platform.

CCS Concepts: • **Software and its engineering** → **Source code generation**; **Domain specific languages**.

Keywords: Program Lifting, Synthesis, TACO, Tensor Algebra

ACM Reference Format:
José Wesley de Souza Magalhães, Jackson Woodruff, Elizabeth Polgreen, and Michael F. P. O'Boyle. 2023. C2TACO: Lifting Tensor Code to TACO. In *Proceedings of the 22nd ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '23), October 22–23, 2023, Cascais, Portugal.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3624007.3624053

1 Introduction

In the last decade, we have witnessed a dramatic increase in machine learning (ML) use in applications ranging from cloud computing to edge devices [45]. ML workloads are dominated by tensor code [60], leading to large-scale efforts aimed at improving its performance [67]. Dense tensor algebra is highly parallel, allowing efficient hardware exploitation across platforms. However, extracting effective parallelism from existing languages is difficult with current compiler technology. This language/compiler failure has led to the growth of domain-specific languages (DSLs) aimed at efficient linear algebra e.g., Diesel [27], TACO [37]. These DSLs deliver excellent cross-platform performance outperforming existing approaches [38].

Accessing such performance is straightforward for new applications: just write your program in the appropriate DSL. However, for legacy programs, it is more problematic with the programmer responsible for both rewriting sections in the new DSL and reintegration with the existing application. As DSLs continuously evolve, this rewriting must be repeated several times throughout the lifetime of the application. This is costly and error-prone, presenting a serious barrier to existing applications to harness hardware performance.

1.1 Existing Techniques

Rewriting is a significant issue and there are a number of different approaches aimed at automatically porting programs to access hardware performance without programmer effort.

API Matching: Rather than translate programs into high level-DSLs, some techniques aim to match and replace sections of user code with fast libraries. For example, Ginsbach et al. [30], De Carvalho et al. [24], and Martínez et al. [44] propose schemes to discover specific code patterns, such as matrix multiplication, and replace them with accelerator calls. However, these matching tools are often brittle and cannot be extended. They require retooling whenever the target API changes, which makes such approaches non-portable.

Program Lifting via Synthesis: There are several lifting approaches based on program synthesis, i.e., algorithms for generating programs from specifications. Synthesis is used directly to lift legacy code in the work by Kamil et al. [34], where the user defines the region of code to lift. However, a compiler from the program source to the internal format and a decompiler to the high-level DSL have to be provided, limiting the applicability of this approach to new DSLs and legacy software. The synthesis used in this lifting is reliant on SMT solvers to guarantee correctness and drive search. This means these techniques cannot be easily applied to the benchmarks we tackle in our paper, which, owing to pointers, and unbounded tensors and loops, are too complex for the state-of-the-art SMT-solver driven software verification tools to reason about. We attempt to verify bounded correctness for some of our synthesized code, but even for simple benchmarks, we cannot verify correctness for tensors of size more than $10 \times 10 \times 10$ within a timeout of 1 hour. This makes this verification impossible to embed into a synthesis loop where we check thousands of candidates. Our synthesis must use alternatives like observational equivalence [22] in order to achieve the necessary scalability.

Neural Machine Translation (NMT): Language models have proved useful in translation/transpilation tasks. In the work by Roziere et al. [55], an unsupervised Java to C# model is learned using a sequence-to-sequence transformer. It is shown to be reasonably accurate, however, like most NMT techniques, it requires a large corpus of source and target code which is not available for emerging DSLs where most source programs do not have a corresponding domain-specific representation.

1.2 Our Approach

This paper presents C2TACO, a synthesis tool for lifting dense tensor code written in C to TACO. We propose a guided enumerative synthesis method to generate TACO programs based on automatically generated IO examples. We use source code analysis to retrieve features from the original programs and use them as search aids during synthesis.

We compared the performance of C2TACO against a neural machine translation approach and two state-of-the-art existing schemes, TF-Coder [59] and ChatGPT [48]. When evaluated on a suite of tensor benchmarks, C2TACO is able to synthesize 95% of the programs, demonstrating considerably higher accuracy than the other techniques (10%, 32% and 24% respectively). Because they are portable, our lifted TACO programs achieve significant performance improvements over the original implementation when evaluated on a multi-core (geo-mean 1.79x) and GPU (geo-mean 24.1x) platform.

```
for (i= 0; i<N; i++){
   for (k = 0; k<N; k++){
      sum =  sum + X[i][k]*b[k];
    }
    a[i] = sum + c[i];
}
```

Figure 1. C implementation of matrix vector product and summation.

This paper makes the following contributions:

- A guided program synthesis technique that discovers and lifts legacy C code to TACO, a domain-specific tensor language based on behavioral equivalence and on the original program structure.
- An extensive evaluation against existing synthesizer and neural machine translation models, showing that our approach has higher coverage and is more accurate than existing approaches.

2 Motivation

In this section we briefly introduce TACO and describe how and why we lift C to TACO.

2.1 TACO

TACO [38] is a high-level programming language for tensor contractions. A tensor is a generalization of a matrix (order 2) to higher orders. It supports tensor expressions of unbounded length and supports tensors of unbounded order. The core TACO language is based on Einstein summation notation (Einsum) allowing concise representation of tensor computation using tensor index notation. It has been used in other frameworks including TVM [19].

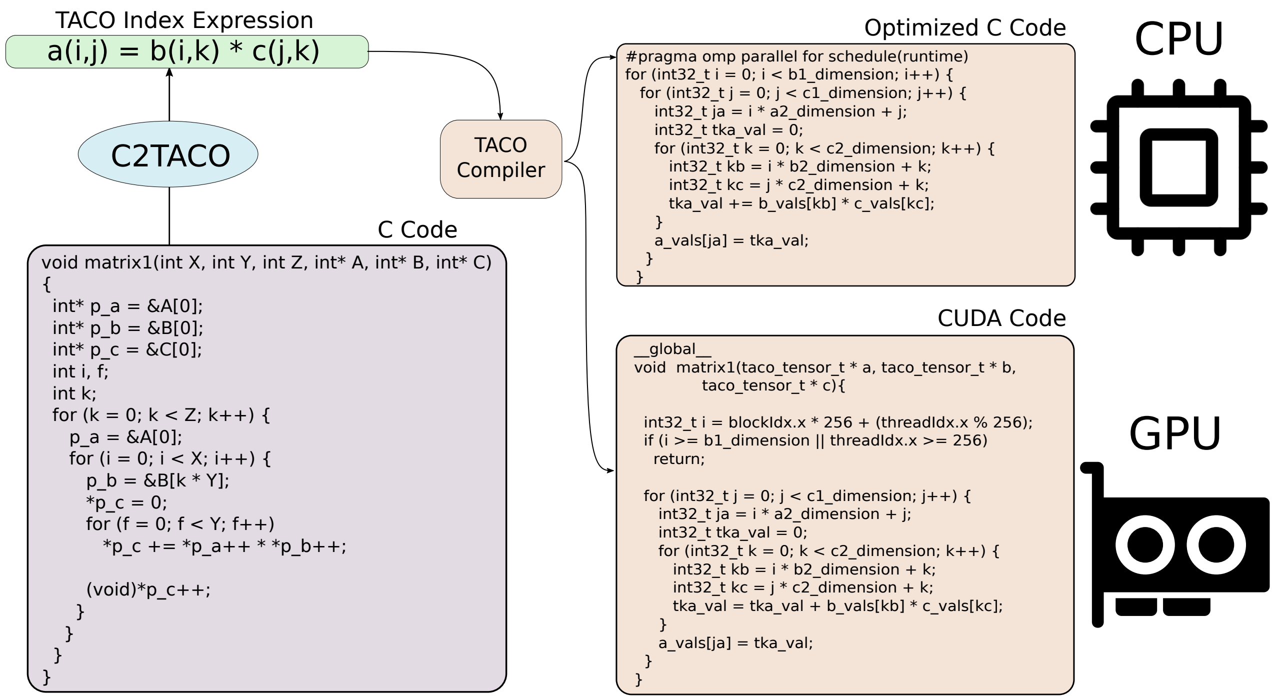
Consider the matrix-vector product and summation example: $a_i = \Sigma_k X_{i,k} b_k + c_i . \forall i$. In C a simple sequential implementation would result in the code shown in Figure 1. While straightforward, targeting this code for different platforms such as multi-cores or GPUs would require significant code restructuring. Writing the example in TACO gives:

$$a(i) = X(i,k) * b(k) + c(i).$$

This is nearer the original formulation and, crucially, does not include any assumptions about whether the platform is sequential or parallel. The TACO compiler takes this program as input and generates platform-specific optimized code.

2.2 Example

We take existing legacy C code, lift it to TACO, and then use TACO's code generation abilities to target diverse, high-performance platforms. Consider the program in Figure 2. This is a C function from the DSPStone benchmark suite [72]



```
void matrix1(int X, int Y, int Z, int* A, int* B, int* C)
{
  int* p_a = &A[0];
  int* p_b = &B[0];
  int* p_c = &C[0];
  int i, f;
  int k;
  for (k = 0; k < Z; k++) {
    p_a = &A[0];
    for (i = 0; i < X; i++) {
      p_b = &B[k * Y];
      *p_c = 0;
      for (f = 0; f < Y; f++)
        *p_c += *p_a++ * *p_b++;

      (void)*p_c++;
    }
  }
 }
}
```

```
#pragma omp parallel for schedule(runtime)
for (int32_t i = 0; i < b1_dimension; i++) {
  for (int32_t j = 0; j < c1_dimension; j++) {
    int32_t ja = i * a2_dimension + j;
    int32_t tka_val = 0;
    for (int32_t k = 0; k < c2_dimension; k++) {
      int32_t kb = i * b2_dimension + k;
      int32_t kc = j * c2_dimension + k;
      tka_val += b_vals[kb] * c_vals[kc];
    }
    a_vals[ja] = tka_val;
  }
}
```

```
__global__
void  matrix1(taco_tensor_t * a, taco_tensor_t * b,
            taco_tensor_t * c){

  int32_t i = blockIdx.x * 256 + (threadIdx.x % 256);
  if (i >= b1_dimension || threadIdx.x >= 256)
    return;

  for (int32_t j = 0; j < c1_dimension; j++) {
    int32_t ja = i * a2_dimension + j;
    int32_t tka_val = 0;
    for (int32_t k = 0; k < c2_dimension; k++) {
      int32_t kb = i * b2_dimension + k;
      int32_t kc = j * c2_dimension + k;
      tka_val = tka_val + b_vals[kb] * c_vals[kc];
    }
    a_vals[ja] = tka_val;
  }
}
```

Figure 2. Lifting C code to TACO using C2TACO. Given a program implemented in C, C2TACO generates a equivalent program written in TACO tensor index notation which the TACO compiler can use to produce high-performance code targeting a variety of hardware platforms.

which makes use of post-increment pointer arithmetic to target the addressing modes found in DSP processors. Although the pointers are a hindrance to understanding, this program is in fact matrix multiplication.

C2TACO uses automatically-generated input/output examples as a specification for an enumerative synthesis algorithm. C2TACO uses information about the C program to guide a search through the TACO grammar in a type-directed template-based enumerative fashion and produces the TACO code shown in Figure 2. As well as being higher-level and easier to read than the original C code, the synthesized TACO program can be optimized and targeted at different platforms.

Figure 2 shows the code generated from tensor index notation for a multi-core CPU and an NVIDIA GPU. For the CPU, the TACO compiler generates OpenMP code with a dynamic runtime schedule policy. So in effect, lifting is an automatic parallelization method for certain C programs. For the NVIDIA GPU, the TACO compiler generates CUDA code (also shown in Figure 2). Although, the code is syntactically distinct from the OpenMP version, the TACO compiler again exploits parallelism with implicit concurrence across all of the threads executing the shown kernel.

2.3 Validity

Our synthesized TACO programs are demonstrated to have observational equivalence with the original programs in C. We also manually inspect the synthesized code. Proving these programs are equivalent is a challenging task due to the unbounded loops and data structures and pointers present in the code. Using CBMC [39], a model checker for C programs, we are able to verify three representative benchmarks using very small loop bounds and tensors (in one case, we can only verify up to a loop bound of 10 within a timeout of one hour). Full verification of synthesized code is an open challenge and out of the scope of this paper.

3 Overview

Figure 3 shows our overall approach. We summarize the pipeline of C2TACO and describe the key components in sections 4 and 5 followed by an extensive evaluation (Section 7).

Given a program P written in C, we first detect the program sections K that are suitable for lifting using neural program classification. Once we have extracted the candidate regions, we generate input-output (IO) examples which are then used as a specification for our synthesis scheme. Our system performs a series of static code analysis to extract relevant features from K. We then search the TACO grammar for equivalent programs that satisfy the IO specification using the features of K to prune the program space. Once we have identified a suitable equivalent TACO program T, we

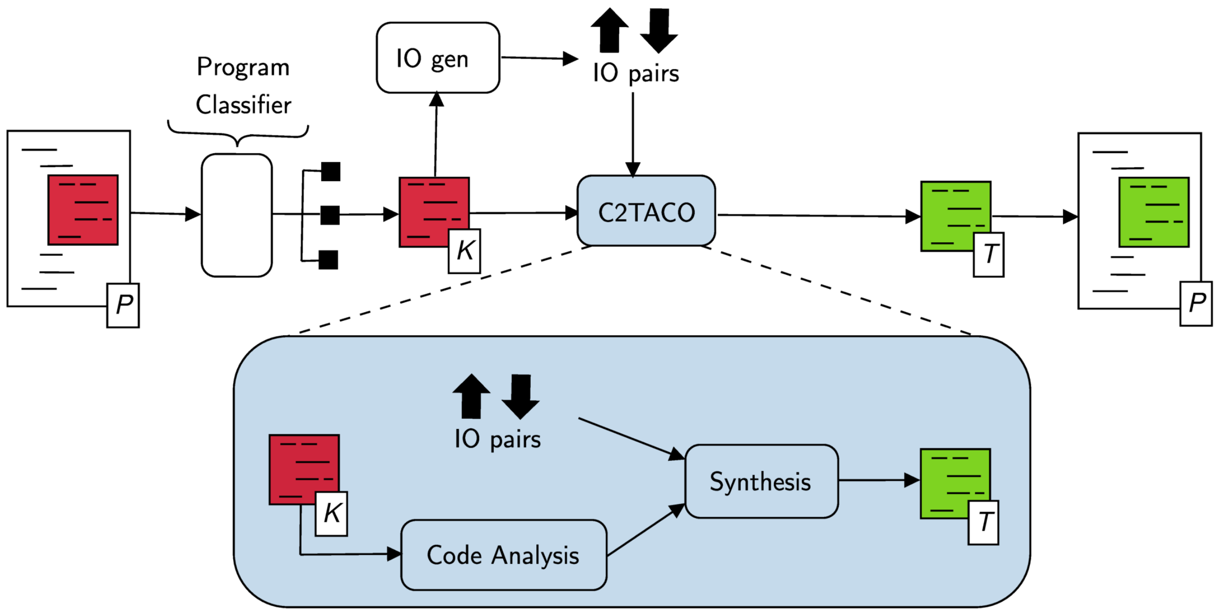


Figure 3. Architecture of C2TACO.

lower it to the target platform and insert it into the original program for execution.

3.1 Classification

We take as input general-purpose programs that perform varied computations and perform lifting to a domain-specific language for tensor contractions. Because we cannot express general computation in TACO, there is a need to identify the code regions that can be lifted and accelerated. We use prior work in neural program classification [69] to determine which parts of the program represent tensor operations.

3.2 IO Generation

Our synthesizer is driven by a specification of observational equivalence (i.e., randomly generated input-output examples). We generate 10 input-output examples. Whilst this means that we cannot *guarantee* absolute equivalence of the synthesized and source code, it allows our synthesis to scale to programs too complex to be reasoned about by the SMT solvers that drive other lifting techniques [17].

3.3 Lifting via Synthesis

Once we have the IO examples of the code to lift, we explore the space of TACO programs using enumeration of templates over TACO's grammar to generate programs that may be equivalent to the original C program. We execute each candidate on the IO samples to see if it is equivalent. The Enumerative Template Synthesis algorithm is described in Section 4. Given the unbounded size of the TACO program space, this can lead to excessive synthesis time. We, therefore, introduce a compiler tool that extracts a set of features from the original C program and use it to guide search, as described in Section 5.

3.4 Lowering

Once we have a suitable candidate TACO program, we then compile it to the target platform using TACO's platform-specific optimizing compilation. In this paper, we investigate multi-core and GPU targets. The generated code is then patched into the original calling program and evaluated on the target platform.

4 Enumerative Template Synthesis

The task of automatically lifting C to TACO can be defined as a formal program synthesis problem. That is, given a source program $P_C : \vec{x} \longrightarrow \vec{y}$, which is written in C, we wish to find an equivalent program $P_T : \vec{x} \longrightarrow \vec{y}$, written in TACO, such that the specification $\forall I \in \vec{x}.P_C(I) = P_T(I)$, i.e., the TACO program behaves identically to the C program on all possible inputs.

We use a bottom-up enumerative synthesis algorithm to enumerate *template* TACO programs, i.e., TACO programs that use symbolic variables in place of all tensors and constants. We then check whether there is a valid substitution of inputs and constant literals for these symbolic variables that satisfies the specification. The enumeration of our algorithm is based on classic algorithms in the literature [9, 66], while the use of a sub-procedure to instantiate concrete variable names and constant literals is based on CEGIS(T) [3].

4.1 The Grammar

Our synthesis algorithm enumerates through a grammar G, shown in Figure 4, which defines a search space of possible template TACO programs. The grammar G is defined as a set of nonterminal symbols NT, terminal symbols, and production rules R. For each rule $r \in R$, $|NT|$ indicates the number of non-terminal symbols in the rule. We refer to the nonterminal symbols on the right-hand side of a rule in the order they appear as $NT_0, NT_1, \ldots$. For example, for the production rule $\langle PROGRAM \rangle$ ::= $\langle TENSOR \rangle = \langle EXPR \rangle$, the nontermimals are $NT_0 = \langle TENSOR \rangle$ and $NT_1 = \langle EXPR \rangle$, and $|NT| = 2$.

The grammar includes symbolic constants and symbolic tensor IDs. When we test the program, we substitute these IDs and symbolic constants with input variables and constants from the source program and test all valid substitutions until we find a program that satisfies the specification. We limit our grammar to 4 index variables, which limits the number of tensor dimensions we can reason about to 4.

$\langle PROGRAM \rangle$ ::= $\langle TENSOR \rangle$ = $\langle EXPR \rangle$

$\langle TENSOR \rangle$ ::= $\langle ID \rangle$ **(** $\langle INDEX\text{-}EXPR \rangle$ **)** | $\langle ID \rangle$

$\langle INDEX\text{-}EXPR \rangle$ ::= $\langle INDEX\text{-}VAR \rangle$
  | $\langle INDEX\text{-}VAR \rangle$**,** $\langle INDEX\text{-}EXPR \rangle$

$\langle INDEX\text{-}VAR \rangle$ ::= $i \mid j \mid k \mid l$

$\langle EXPR \rangle$ ::= $\langle EXPR \rangle$ **+** $\langle EXPR \rangle$
  | $\langle EXPR \rangle$ **-** $\langle EXPR \rangle$
  | $\langle EXPR \rangle$ ***** $\langle EXPR \rangle$
  | $\langle EXPR \rangle$ **/** $\langle EXPR \rangle$
  | $\langle CONSTANT \rangle$
  | $\langle TENSOR \rangle$

$\langle ID \rangle$ ::= $T_0 \mid T_1 \mid T_2 \mid \ldots$

$\langle CONSTANT \rangle$ ::= $C_0 \mid C_1 \mid C_2 \mid \ldots$

Figure 4. TACO grammar.

4.2 Specification

Given a source function $P_C : \vec{x} \longrightarrow \vec{y}$, we wish to find an equivalent TACO function $P_T : \vec{x} \longrightarrow \vec{y}$ such that $\forall I \in \vec{x}.P_C(I) = P_T(I)$. Checking this equivalence is undecidable in general, however, due to the lack of data-dependent control-flow in TACO programs, it is sufficient in almost all cases to check observational equivalence.

We extend the method set out in FACC [69], where inputs are randomly generated according to manually given constraints dictating the length of arrays and favoring smaller values to make evaluation faster. We constrain arrays to be of size 4096, and fix tensor-dimensions to be equal (e.g., a 2-dimensional tensor is of size 64×64).

A single input-output example I, O consists of a set of randomly generated arguments $I = (i_1, \ldots i_m)$, corresponding to the input parameters $\vec{x} = (x_1, \ldots, x_m)$, and an output $O = P_C(i_1, \ldots, i_m)$, We generate 10 input-output examples which form a specification: $\phi_{IO} = \{(I,O)_1, \ldots, (I,O)_{10}\}$ A program P_T satisfies the specification ϕ_{IO} iff $\forall (I,O) \in \phi_{IO}.P_T(I) = O$. To determine this in practice, we run P_T using the TACO Python API, checking if the behavior matches the corresponding outputs.

4.3 Template Enumeration

We implement bottom-up enumeration i.e., we enumerate templates starting with the shortest first. We define the length of a template as the number of references to tensors or constants in the template, e.g., the template $T_0[i] = T_1[i] + 2$ has length 3 because it refers to T_0, T_1 and 2.

We enumerate templates as shown in Algorithm 2, by iterating through production rules until we have found all possible complete templates of length 1 in the grammar. We then increase the length and repeat the process, using the previously enumerated templates as building blocks, until we have hit the maximum user-given length. Each time the length increases, we add a new tensor ID and a new symbolic constant to the set of candidate templates. This is shown in Algorithm 1.

We discard any invalid candidates during enumeration, i.e., templates that do not type check or are unsupported by TACO. More specifically we discard: any candidate that iterates over two different dimensions with the same index variable (e.g., $T_0(i,i)$); any candidate where the same tensor appears more than once in a program with different orders (e.g.: $T_0(i) = T_1(i) * T_1(i,j)$); and any candidate where the same tensor appears on both sides of an assignment (e.g.: $T_0(i,j) = T_0(i,j) + T_1(j,k)$).

4.4 Instantiating Templates

After we have generated all templates of length L, we check whether any of these templates generate programs that satisfy the specification, ϕ_{IO} (see Section 4.2). To do this, we enumerate through all substitutions that map all symbolic constants in the candidate program to concrete values, and all tensor IDs to inputs in the specification, until we find a substitution that gives us a TACO program that satisfies the specification. This is shown in Algorithm 2. We limit the concrete constant values to constants present in the source program.

We check all possible substitutions until we find a substitution that results in a complete TACO program that satisfies the specification, which is checked by the *check* procedure. Although checking all possible substitutions has $L!$ complexity for a template of length L, L is typically small (< 5). We check the templates of length MAX before any shorter templates, as this is the likely length of the target program.

Algorithm 1: Enumerative Template Synthesis. The subprocedures *instantiate* and *completeRule* are shown in Algorithm 2.

```
input  : source code $P_C$, grammar $G$, max length $L$
output : candidate program, or no solution
Algorithm synthesize($P_C, G, L$)
    $short \leftarrow \emptyset$;      // set of short candidates
    $long \leftarrow \emptyset$;       // set of long candidates
    $\phi_{IO} \leftarrow generateSpec(P_C)$;
    for $l$ in $1 \dots L$ do
        $short \leftarrow short \cup newTensor() \cup newCons()$;
        while $true$ do
            $nS \leftarrow \emptyset$;   // new short candidates
            $nL \leftarrow \emptyset$;   // new long candidates
            for $Rule \in G$ do
                for $p \in completeRule(short, Rule)$ do
                    if $Length(p) = L \land valid(p)$ then
                        $nL \leftarrow nL \cup p$;
                    else if $Length(p) < L \land valid(p)$ then
                        $nS \leftarrow nS \cup p$;
            if $nS \subseteq short \land nL \subseteq long$ then break;
            if $l = L$ then
                $long \leftarrow long \cup nL$;
            else
                $short \leftarrow short \cup nS \cup nL$;
    for $p \in long, short$ do
        $P_T, result \leftarrow instantiate(p, \phi_{IO})$;
        if $result$ then return $P_T$;
    return no solution
```

Algorithm 2: Subprocedures. Note $e.\{x \mapsto y\}$ denotes the result of the proper substitution of the expression x by the expression y in the expression e.

```
Procedure completeRule(short, Rule)
    $completions \leftarrow \emptyset$;
    for $p \in short$ do
        $candidate \leftarrow Rule.\{NT_0 \mapsto p\}$;
        if $|NT| \in Rule = 2$ then
            for $q \in short$ do
                $candidate \leftarrow Rule.\{NT_1 \mapsto q\}$;
        $completions \leftarrow completions \cup candidate$;
    return completions
Procedure instantiate($p, \phi_{IO}, P_C$)
    $X \leftarrow getInputParams(P_C)$;
    $K \leftarrow getConstants(P_C)$;
    $T \leftarrow getTensors(p)$;
    $C \leftarrow getConstantSymbols(p)$;
    for $x, t \in cartesianProduct(X, T)$ do
        for $k, c \in cartesianProduct(K, C)$ do
            $result \leftarrow check(p.\{t \mapsto x\}.\{c \mapsto k\}, \phi_{IO})$
            if $result$ then return
            $result, p.\{t \mapsto x\}.\{c \mapsto k\}$;
```

5 Synthesis Guided by Code Analysis

The search space of possible TACO templates is large, and so, in C2TACO, we use program analysis to focus the scope of the synthesis search, prioritizing candidates that are more likely to be correct. In particular, we use heuristics to estimate the correct TACO template length (section 5.1), the correct dimensions (section 5.2) and the operators (section 5.3).

5.1 TACO Program Length

The length of a TACO program is related to the number of array/pointer references and constants in the original C code. However, temporary variables to capture common subexpressions and mutable arrays mean that there is no direct correspondence. Fixing the size of the target TACO program reduces the search space because we only have to enumerate candidates once.

To determine the range of sizes C2TACO explores, we focus on the definition of the *output* array and examine the number of input arrays, or *uses* [23]. At each definition, we iteratively build a set of variables used by that definition. We use reaching analysis to disambiguate between different references to the same (mutable) variables. We then reduce the constructed set in the presence of summations or reductions. In C, when writing a reduction or summation, a variable appears on both sides of an assignment but only once in the TACO program. For this reason, we apply simple data dependence analysis to check if there is a recurrence. If there is, we do not count it twice.

For example, in Figure 1, we have the use set `sum, X, b, c` for the the output array `a`. This is reduced to `X, b, c` after detecting the reduction on `sum` to give 4 (`a, X, b, c`) as the predicted number of tensors in the TACO program.

5.2 Tensor Dimensions

C programs frequently contain linearized arrays: where a single pointer is used to represent a multi-dimensional tensor. However, in TACO dimensions are explicit, and searching over all possible dimensions is costly. To address this, we apply the dataflow analysis defined in [29] to recover arrays from pointer structures, and then apply delinearization [47] and determine the highest dimension array.

As an example, consider the program in Figure 2. After applying dataflow analysis to `*p_c`, we get `p_c`$[Z * k + i]$. Let n be the dimensionality of the recovered C array and m be the dimension of the enclosing loop nest J. Here $n = 1, m = 3$. The loop iterators are represented by a column vector $J = [k, i, f]^T$, and UJ is the affine expression for the array access $[Z * k + i]$:

$$UJ = [Z, 1, 0] \begin{bmatrix} k \\ i \\ f \end{bmatrix} = [Z * k + i]$$

We now delinearize by constructing a transformation S, such that SU gives a matrix of 1's and 0's. For our example $S = [()/Z, ()\%Z]$. For details of this step, we refer the reader to the paper [47]. We apply the transformation to give:

$$SUJ = \begin{bmatrix} ()/Z \\ ()\%Z \end{bmatrix} [Z, 1, 0]J = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} J$$

This gives us a 2D delinearized array access $p\_c[k, i]$, so we begin our search for TACO programs using 2D tensors.

5.3 Operator Analysis

Finally, we use the source code to predict which operators are likely to be included in the target program. We do this based on a straightforward analysis of the Abstract Syntax Tree of the source code, which counts the number of appearances of each operator type. This effectively reduces the search space of possible TACO programs by eliminating unlikely combinations of operators.

6 Experimental Methodology

To evaluate C2TACO we compared its performance against other techniques. We implemented a simple version of the synthesis process described in Section 4 and an alternative approach based on neural machine translation. In addition, we consider an existing large language model ChatGPT and IO-based synthesizer, TF-Coder.

6.1 Alternative Approaches

ETS. C2TACO uses the synthesis algorithm described in Section 4 combined with the heuristics described in Section 5. To evaluate the contribution of the heuristics in C2TACO, we compare to the most basic enumerative template synthesis algorithm described in Section 4 (without any heuristics), which we refer to as ETS.

Neural Machine Translation. NMT converts text sequences from one language to another by means of a deep neural network and has shown positive results on code tasks. We therefore frame the task of lifting C to TACO as a neural machine translation problem. We train a Transformer [68] that given a C input sequence minimizes the edit distance between the predicted and ground-truth TACO. Once trained, then, given an unseen C program, the model will generate the most likely equivalent TACO program.

The main challenge for any new DSL is the availability of training data. To overcome this, we generate a synthetic dataset based on the TACO grammar shown in Figure 4. We compile the synthetically generated TACO programs to generate the equivalent C programs. We limit our synthetic dataset to programs that contain a maximum of 5 tensors of no more than 4 dimensions, and where all datatypes are integers.

We enumerate this space in a bottom-up manner, similar to the enumeration performed by our synthesis algorithm, and use testing to eliminate semantically equivalence programs. Since TACO-generated programs contain details that are unlikely to be present in real-world tensor kernels such as memory allocation, we modify the `clang` compiler to extract only the kernel signature and computation of the program for our equivalent C program.

We generate 800K pairs of C program and TACO expressions of which we separated 5K for validation, 5K for test, and the remaining were used for training. The trained model is a Transformer with 6 encoders and 6 decoders with 16 attention heads and an embedding size fixed at 1024.

6.2 Existing Approaches

TF-Coder. TF-Coder [59] is an open-source publicly available program synthesizer. It takes a single input-output example as source and generates a corresponding TensorFlow program. Although the search space of TF-Coder is not defined by the same grammar we considered in our synthesis methods, we compare C2TACO against TF-Coder because both synthesize programs from IO examples and operate on the domain of tensor computations. We use one of the IO examples automatically generated by our synthesis scheme, but limit it to less than 100 elements as required by TF-Coder.

ChatGPT. ChatGPT [48] is large-scale language model based on GPT 3.5. It has been used for a wide number of tasks including code generation. We used version 3.5 in our experiments. As its accuracy depends on the quality of its prompts, we experimented with various formats and found the following to be the most effective, followed by the original source code:

```
"Translate the following C code to an expression in
the TACO tensor index notation. The expression must
be valid as input to the taco compiler. Return the
expression and only the expression, no explanations."
```

6.3 Setup

Benchmarks. To evaluate C2TACO, we designed two different suites of tensor algebra benchmarks. The first contains C programs generated by the TACO compiler a distinct subset of those used to train the NMT model. The second contains programs from existing software libraries. We refer to these suites as *artificial* and *real-world* respectively.

The real-world benchmarks originate from different applications. We selected a subset of the programs used by previous synthesis work [22]:

Table 1. Synthesis coverage of different approaches on the artificial dataset.

	Correct				
TACO Program	TF-Coder	ChatGPT	NMT	ETS	C2TACO
a(i) = b(i) + c(i) - d(i)	✓	✗	✗	✓	✓
a(i,j) = b(i,j) + c(i,j)	✓	✓	✓	✓	✓
a(i) = b(i) * c(i)	✓	✗	✓	✓	✓
a(i) = b(i) + c(i) + d(i) + e(i)	✓	✗	✗	✗	✓
a(i,j) = b(i,j) * c(j)	✗	✗	✓	✓	✓
a(i,j) = b(i,k) * c(k,j)	✗	✓	✓	✓	✓
a(i,j) = b(i)	✗	✓	✓	✓	✓
a(i,j) = b(i) * c(i,j)	✗	✗	✓	✓	✓
a(i,j) = b(i,j,k) * c(k)	✗	✓	✓	✓	✓
a(i,j) = b(i,k,l) * c(l,j) * d(k,j)	✗	✓	✗	✗	✓

- **blas**: baseline implementation of functions from the BLAS [18] linear algebra library as synthesized by Collie et al. [20].
- **DSP**: signal processing functions adapted from the TI [2] library.
- **makespeare**: programs that manipulate arrays of integers. Originally from Rosin [54].
- **mathfu**: mathematical functions from the Mathfu [1] library.
- **simpl_array**: problems performing different computation on arrays of integers. Originally from the work by So and Oh [62].

In addition to those, we extracted benchmarks from other suites that contain tensor manipulations:

- **darknet**: neural network operations from the Darknet [53] deep learning framework.
- **DSPStone** and **UTDSP**: kernels targeting digital signal architectures from the DSPStone [72] and UTDSP[56] suites.

We gathered 71 benchmarks in total, of which 10 are artificial and 61 come from real-world code.

Software. ETS and C2TACO are implemented in Python version 3.8.10. The NMT Transformer model is implemented using Fairseq [49] 0-12.2 with Google's SentencePiece [40] as the tokenizer. The analyses described on Section 5 are implemented as plugins for the `clang` compiler version 14.0.0. Operating system is Ubuntu 20.04.6 LTS.

Hardware. We evaluate on a multi-core CPU and GPU platform. The targeted CPU is an 8-core Intel i5-1135G7 at 2.40GHz with 16 GB of RAM (LPDDR4) at 4267 MT/s. The GPU is an NVidia GeForce GTX 1080 Ti using driver version 535.54.03 and CUDA runtime version 12.2.

Metrics. We evaluated the performance of each approach by executing its generated code 10 times and recording the median. In our experiments, we saw little execution time variance. We measure speedup as the ratio of the running time of lifted programs over the original version. Programs are compiled with `gcc -O3` version 9.4. We also recorded the time to produce a lifted TACO program with a timeout of 90 minutes for all approaches in all the experiments conducted.

7 Evaluation

In this section, we evaluate against four criteria: coverage (Section 7.1), error rate (Section 7.2), synthesis time (Section 7.3), and speedup (Section 7.4).

7.1 Synthesis Coverage

Figure 5 shows the lifting coverage of each of the five schemes described in section 6 across the two benchmark suites: artificial and real-world.

Benchmark Suite: Artificial. As described in section 6, these are C kernels generated by the TACO compiler guaranteed to have an equivalent in the TACO language. The coverage of each scheme is shown in Table 1 and Figure 5.

C2TACO is most effective, lifting all benchmarks correctly. ETS lifts 8 out of 10. In two cases it could not find the correct program in time as the space of possible grows too large. C2TACO overcomes this by using the code analysis information to focus the search on parts of the grammar where the programs are most likely to be the solution. TF-Coder is able to synthesize 4 out of 10 benchmarks but is unable to match the coverage of the other synthesis approaches. Like ETS scheme, it times out for the more complex programs.

NMT achieves higher accuracy, translating seven of the benchmarks. The Transformer model was trained using TACO-generated kernels, which have a similar structure to the synthetic programs. Unlike synthesis methods, it always produces a result even though it may be inaccurate and does not timeout. In the three cases where NMT fails, it correctly guesses the number of tensors but misorders them in the resulting programs.

ChatGPT is able to correctly predict 5 of the 10 benchmarks, hallucinating the remainder. In four cases it produces syntactic invalid programs. The syntax errors include wrong

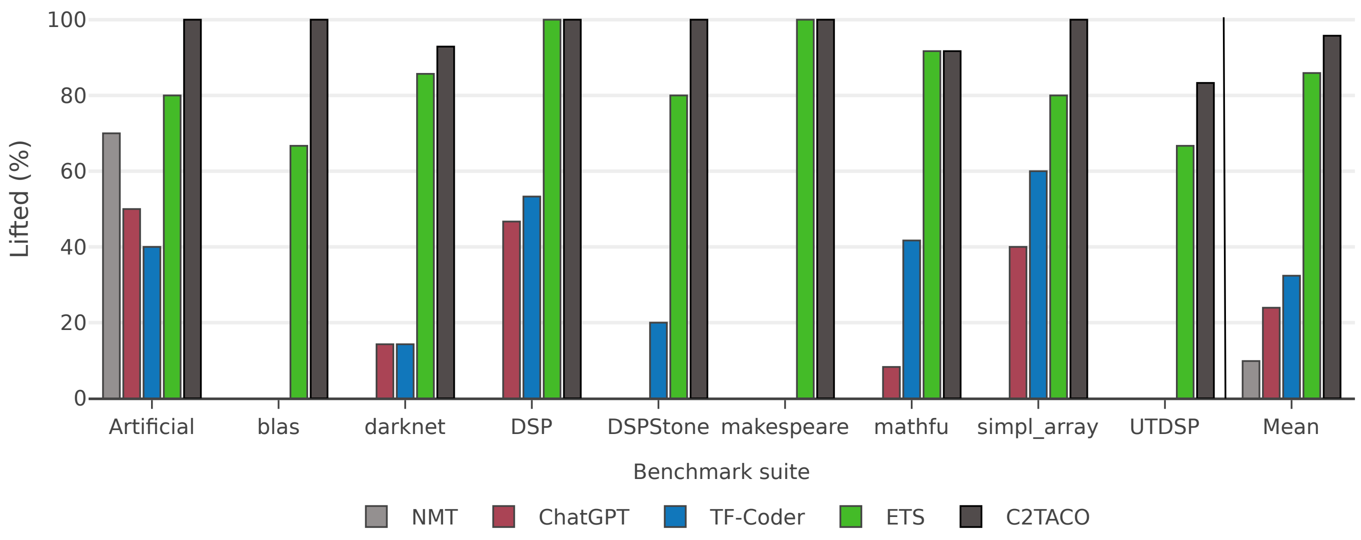


Figure 5. Overall lifting coverage across benchmarks suites.

indices, multiple assignments, and duplication. In one case a tensor was treated as having different orders in the same program. In another, ChatGPT produces a program that is syntactically valid but incorrectly refers to the same tensor twice.

Benchmark Suite: Real-World. Real-world benchmarks are more challenging as shown in Figure 5. Both ETS and C2TACO are able to achieve high coverage of 85% and 95% respectively. ETS times out on 5 out of 61 while the sole instance of failure for C2TACO is the presence of program features not contemplated in our implementation of the grammar 4. TF-Coder manages to correctly synthesize 31% of the benchmarks. Along with timeouts, TF-Coder also produces programs that are semantically incorrect. We further discuss these in Section 7.2.

Real-world programs impose a harder challenge to neural machine translation due to the diversity of their implementation. While artificial programs have a syntactic structure identical to TACO-generated C programs, real-world ones are written in several different fashions, which makes it difficult for sequence-to-sequence methods to recognize patterns. NMT performs particularly poorly compared to the artificial case, generating no correct programs. This reinforces the view that it may be over-specific to a particular style of programming due to its training sample. ChatGPT also has a weak performance, only translating 20% of the benchmarks correctly. As well as in the artificial case, both approaches produce varied hallucinations as we detail below.

7.2 Error Analysis

We identify several different reasons for failure: a large search space causing time out; syntactic and semantically wrong solutions. Figure 6 depicts a summary.

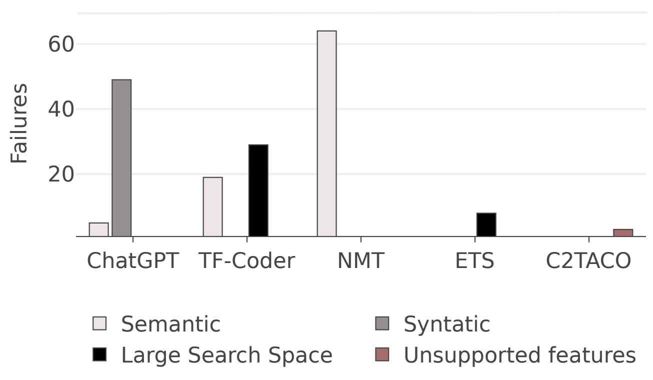


Figure 6. Distribution of failure causes for the different approaches evaluated.

Large Search Space. Enumerative synthesis techniques explore a large search space, which grows as program length increases. This causes 60.42% of TF-Coder's failures and all failures for ETS. Neural translation approaches, ChatGPT and NMT, always find a solution in time due as they translate a program in a sequence-to-sequence fashion and do not perform an extensive search. Although C2TACO is also based on enumeration, it never times out as program analysis restricts the search space sufficiently.

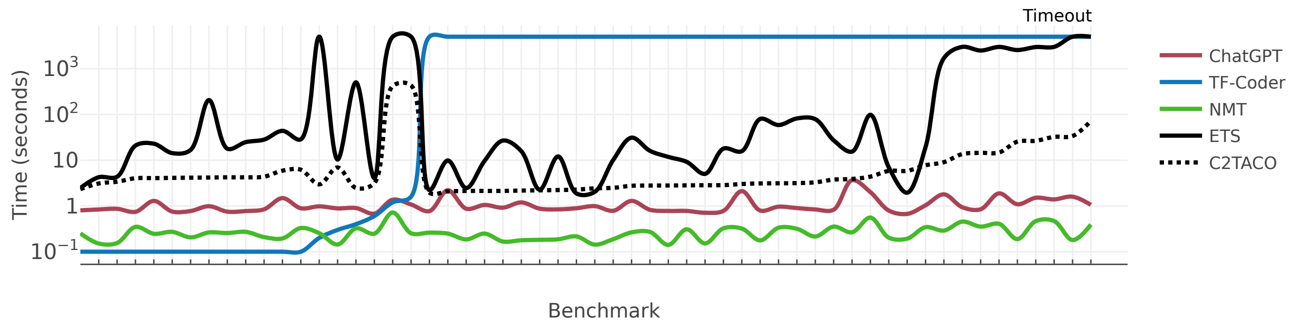


Figure 7. Lifting time on real-world benchmarks. Y-axis is on logarithmic scale.

Syntactic. TF-Coder, ETS, and C2TACO always produce programs that are syntactically correct. On the other hand, neural approaches frequently generate incorrect translations or hallucinations. In addition, 90% of the wrong translations produced by ChatGPT are syntactically incorrect. These hallucinations often include explanations of the ranges of index variables and use braces instead of parenthesis, which is the symbol used for indexation in the TACO tensor index notation language. Example 1 shows a syntactic hallucination produced by ChatGPT.

Example 1. When given as input a program that computes a dot product of two arrays b and c, the expected solution expressed in TACO is

$$a = b(i) * c(i)$$

However, ChatGPT produced the string below which is not a valid TACO program.

$$sum(a[i] * b[i] \; for \; i \; in \; 0..< n)$$

Although NMT is also neural-based it always produces well-formed programs. The difference is that NMT is trained on a domain-specific dataset containing only programs generated by the TACO compiler while ChatGPT is trained on more diverse data.

Semantic. These are programs that are syntactically correct, but produce the wrong output when executed. Almost 40% of TF-Coder failures are programs that are semantically wrong. TF-Coder relies on just one IO example and often fails to generalize. The majority of false positives produced by TF-Coder include manipulations on the shape of tensors, which is not present in any of the original benchmarks. Semantic hallucinations also correspond to 9.26% of the incorrect answers produced by ChatGPT. Example 2 shows an example of a hallucination produced by ChatGPT and Example 3 depicts one generated by TF-Coder.

Example 2. For a program that performs general matrix multiplication, the solution can be expressed in TACO as

$$C(i, j) = A(i, k) * B(k, j)$$

ChatGPT generates a program that includes an extra summation and reference to the resulting matrix on the right-hand side. Although that is equivalent according to C semantics, the same is not true in TACO.

$$C(i, j) = C(i, j) + ALPHA * A(i, k) * B(k, j)$$

Example 3. Given a program that computes the product of an array arr with a scalar value v, the correct TACO implementation is:

$$arr(i) = arr(i) * v.$$

TF-Coder synthesizes a solution that, although syntactically valid in TensorFlow, adds arr to itself, which is not semantically equivalent to the original program:

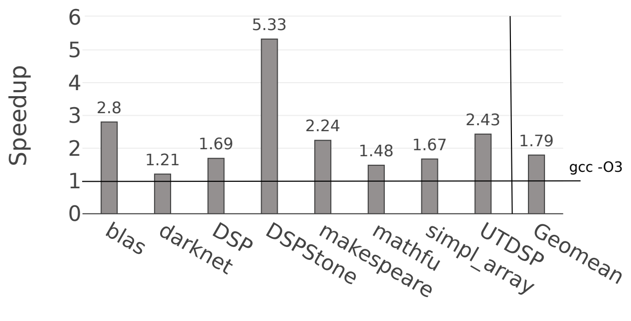
```
tf.add(arr, arr)
```

TACO-generated programs have a particular code structure that does not reflect real-world programming styles, which is why NMT fails to generalize. Semantic hallucinations are the cause of all of NMT's failures.

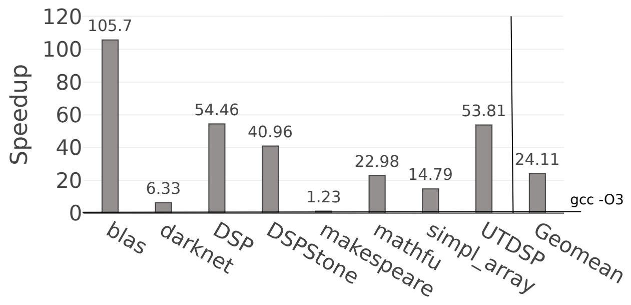
7.3 Generation Time

Artificial. NMT is by far the fastest approach with a geometric mean of 0.36 seconds. NMT is faster because it does not involve an extensive search and it does not check whether the program is correct using IO examples, which represents the largest part of the synthesis time for the program synthesis approaches. ChatGPT is also fast for the same reasons and translated artificial benchmarks within 1.14 seconds on average.

Despite performing a search, TF-Coder is fast, taking an average of 1.18 seconds to find the solution. Nevertheless, TF-Coder is only able to correctly lift 40% of the artificial benchmarks (Section 7.1). ETS is the slowest method with an average of 238 seconds to find the solution. In contrast, C2TACO takes an average of 21 seconds. That result shows the impacts of the program features obtained by syntactic analyses in guiding the synthesizer to find the correct answer.



(a) CPU



(b) GPU

Figure 8. Speedup obtained by the synthesized TACO programs on different hardware platforms. The baseline is the average running time of the original implementations when compiled with `gcc -O3`.

Real-World. Figure 7 shows the synthesis times for each of the five approaches across the real-world collection. Numbers are on a logarithmic scale. As expected both the neural approaches NMT and ChatGPT are fast and stable across all programs. NMT always returns a program in less than 1 second and ChatGPT takes a maximum of 4 seconds to find a solution. However, as shown in Section 7.1, this speed comes at the expense of frequently generating wrong code.

TF-Coder performs well on the simpler program. It synthesized a solution even faster than neural approaches in 15 cases. Nevertheless, the generation time of TF-Coder rose sharply as the programs became less trivial and it timed out in 42 out of 61 instances. ETS is slower on average however it only times out on 13% of the benchmarks. We observed that ETS particularly struggles with instances of length N ≥ 3 and programs involving multiple multidimensional tensors, where the number of possible index expressions increases exponentially for each tensor. C2TACO is considerably faster with an average synthesis time of 5.6 seconds and a maximum of 7 minutes. The only cases where C2TACO was slower than ETS involve very simple programs that only perform initialization of arrays with a constant value. For all the other programs C2TACO was able to find a solution faster than ETS and it kept stable across the whole suite.

7.4 Performance of Lifted Code

The main reason we wish to lift code to TACO is to exploit its portable performance. We generated C and CUDA versions of the programs generated by C2TACO and measured their performance on a multi-core CPU and GPU respectively. Figure 8 shows the speedup across the benchmark suite achieved running lifted programs. Baseline is the original implementation compiled with `gcc -O3`. Only the real-world benchmarks are considered as the artificial ones are directly derived from the TACO compiler and the synthesized version corresponds to the original.

Lifted programs are faster than their original counterparts in both devices. On a multi-core device, the benchmarks are on average 1.79x faster when lifted to TACO. That speedup varies over different benchmark sources. The highest speedup is 5.33x on DSPStone benchmarks and the lowest is 1.21x for the darknet programs. The main reason for the better performance is that the kernels generated by TACO optimize array access by linearizing index expressions and exploit the parallel nature of a multi-core CPU by inserting OpenMP pragmas on loops.

Speedup is even higher on the GPU. The lowest value was 1.23x on the makespeare set. However, it is worth emphasizing that makespeare contains only 1 program. We noticed high speedups on the digital signal processing benchmarks: DSP, DSPStone, and UTDSP, on which lifted programs are 54.46x, 40.96x and 53.81x faster than the original version. The highest value occurs on the BLAS benchmarks, which run 105.7x faster when lifted. The overall speedup achieved on GPU was 24.11x. Similarly to the multi-core kernels, TACO-generated CUDA kernels are designed to leverage high-level parallelism on GPU accelerators and are optimized aiming to divide the workload uniformly among threads.

Speedup by Program Complexity. We further evaluated the impact of lifting on the performance of programs when such programs become more complex. In our domain, we consider programs more complex as they manipulate tensors with higher orders. We define the concept of dominant order as the highest order among the tensors in a program. For example, the program shown in Figure 1, manipulates tensors of 3 different orders: vectors (order 1), a matrix (order 2) and a scalar variables (order 0). The dominant order for that program is therefore 2.

Table 2 shows the overall speedup obtained on programs with different dominant orders. We observed two categories of dominant orders in the real-world benchmarks, 1 and 2. Programs that handle two-dimensional tensors benefit more

Table 2. Speedup obtained given different tensor dominant orders. We consider the highest order among the tensors in a program as dominant.

Dominant order	Multi-core Speedup	GPU Speedup
1	1.41	20.19
2	3.20	36.97

from being lifted than the ones operating on one-dimensional ones. The speedup goes from 1.41x to 3.20x on the multi-core and from 20.19x to 36.97x on the GPU. These results show that the impact of lifting is even higher for programs that are more complex in the sense that they manipulate multi-dimensional tensors.

7.5 Summary

Overall C2TACO was the most effective method in our evaluation, lifting 95% with an average time of 21 seconds on the artificial suite and 5.6 seconds on the real-world programs. C2TACO was considerably faster than its ETS counterpart, which illustrates that the program analysis used by C2TACO to guide the search shown have a large impact on its generation time. We shown that we obtain performance gains by lifting programs to TACO, achieving an average speedup of 1.79x on a multi-core platform and 24.1x on a GPU.

8 Related Work

In this section we discuss how our work relates to the area of program synthesis and other techniques to automatically construct code.

8.1 Program Synthesis

Program synthesis is a well-studied area where programs are generated based on an external specification. It is the form of specification and the methodology used to generate programs that characterize the different approaches.

Logic. In Syntax-Guided Synthesis(SyGuS) [10] approaches, the program specification is provided in the form of first-order logic. This type of specification allows SMT solvers such as Z3 [25] to be used in a CounterExample Guided Inductive Synthesis(CEGIS) [63] loop to rapidly synthesize candidate programs. Recent work allows extension beyond first-order logic [51], but SyGuS is not well-suited to tensor computations due to the complexity of checking the correctness of a tensor computation using an SMT solver. Due to this limitation, our work uses a testing-based procedure to validate candidates. Our synthesis approach is similar in style to CEGIS(T) [3], in that we enumerate programs with symbolic constants and tensors, and then find the bindings for these constants as part of the correctness check.

IO Examples. IO-based synthesis is part of the programming by example style of synthesis, in which input/output examples are used as the specification. Early work looked at generating Excel commands from a few examples [31]. The same concept and has been used for other tasks [21, 73], including generating PyTorch or TensorFlow code from tensor inputs [46, 59]. TF-Coder [59] takes as input a single user-provided example to generate equivalent TensorFlow code using type constraints and bottom-up enumerative synthesis. Alternative schemes [16, 46] use deep learning models trained on IO samples to guide the generation of code.

Verified Lifting. Using program synthesis to generate programs from a specification is a long-studied area [28, 61]. Using a low-level program as the specification and a high level-one as the target was tacked by Kamil et al. [34]. Here appropriate stencil-like loops in FORTRAN are lifted to their equivalent in Halide [52]. This has been extended to a more generic LLVM framework [4] based on a common IR. While this has the potential to allow lifting to multiple targets [5–7], it requires the compiler writer to provide a compiler and decompiler from each potential source and target into the IR which is not scalable. Their technique also relies on being able to formally verify the equivalence of the target and source programs in order to give counterexamples to the synthesis algorithm, which we have found is not possible for programs in our benchmark suite. In contrast, whilst it gives weaker guarantees of correctness, our approach is able to synthesize programs based on observational equivalence, and the scalability of our approach is not dependent on the tractability of the equivalence checking problem.

8.2 Other Approaches

Neural Machine Translation. Since the advent of sequence to sequence models [64], neural machine translation has been applied to programming language translation tasks [11, 12, 26], including unsupervised settings [13, 55, 65]. Training data is often extracted from coding websites [42].

Other tasks range from code style detection [50], generating accurate variable names [41], correcting syntax errors and bugs [32, 57] code completion [36] and program synthesis [14] to API recommendation [35], and specification synthesis [43]. While powerful, such approaches are inaccurate and are not mature enough for precise lifting.

API Migration/Matching. Replacing matched code/IR to a fixed API call is a limited form of raising. KernelFaRer [24] works at the program level and restricts its attention to just GEMM API targets, but is more robust than IDL [30] matching significantly more user code. This robustness is extended further by Martínez et al. [44] which uses behavioral equivalence to match code. Such approaches, however, are intrinsically limited as they focus on fixed APIs rather than the open-ended nature of DSLs and their IRs.

Compiling TACO. TACO [37] is a popular DSL for expressing tensor computations. In addition to generating high-performance CPU code [38], it has been extended to compile to GPUs [58], CGRAs [33], high-performance libraries [15] and distributed systems [70]. In addition to these target-specific optimizations, work has been done for sparse tensors [8, 71].

9 Conclusion

This paper presents C2TACO, a synthesis tool for lifting C tensor code to TACO. C2TACO uses equivalence behavior and program analysis to generate code and it is shown to lift more programs in a shorter time with greater accuracy when compared to an alternative NMT and simpler synthesis approaches. C2TACO also outperforms existing techniques, lifting 95% of the benchmarks, against 32% for TF-Coder and 24% for ChatGPT. We demonstrate that the synthesis of equivalent TACO programs is feasible for a range of C programs taken from software libraries and benchmark suites. We also show that we can obtain significant performance improvement over the original source. Using C2TACO we are able to synthesize TACO programs that are 1.79x faster when evaluated on a multi-core CPU and 24.1x when ported to a GPU platform. Future work will explore methods to further improve lifting applicability, by handling sparse tensor algebra, and efficiency using neural-guided synthesis to perform search.

References

[1] [n. d.]. Mathfu. https://github.com/google/mathfu.

[2] [n. d.]. Texas Instrument Digital Signal Processing (DSP) Library for MSP430 Microcontrollers. https://www.ti.com/tool/MSP-DSPLIB.

[3] Alessandro Abate, Cristina David, Pascal Kesseli, Daniel Kroening, and Elizabeth Polgreen. 2018. Counterexample Guided Inductive Synthesis Modulo Theories. In *CAV (1) (Lecture Notes in Computer Science, Vol. 10981).* Springer, 270–288.

[4] Maaz Bin Safeer Ahmad and Alvin Cheung. 2016. Leveraging parallel data processing frameworks with verified lifting. *arXiv preprint arXiv:1611.07623* (2016).

[5] Maaz Bin Safeer Ahmad and Alvin Cheung. 2017. Optimizing Data-Intensive Applications Automatically By Leveraging Parallel Data Processing Frameworks. In *Proceedings of the 2017 ACM International Conference on Management of Data.* 1675–1678.

[6] Maaz Bin Safeer Ahmad and Alvin Cheung. 2018. Automatically leveraging mapreduce frameworks for data-intensive applications. In *Proceedings of the 2018 International Conference on Management of Data.* 1205–1220.

[7] Maaz Bin Safeer Ahmad, Jonathan Ragan-Kelley, Alvin Cheung, and Shoaib Kamil. 2019. Automatically translating image processing libraries to halide. *ACM Transactions on Graphics (TOG)* 38, 6 (2019), 1–13.

[8] Peter Ahrens, Fredrik Kulstad, and Saman Amarasinghe. 2022. Autoscheduling for Sparse Tensor Algebra with an Asymptotic Cost Model. *PLDI* (2022).

[9] Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. 2013. Recursive Program Synthesis. In *CAV (Lecture Notes in Computer Science, Vol. 8044).* Springer, 934–950.

[10] Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-guided synthesis. In *FMCAD.* IEEE, 1–8.

[11] Jordi Armengol-Estapé and Michael O'Boyle. 2021. Learning C to x86 Translation: An Experiment in Neural Compilation. In *Advances in Programming Languages and Neurosymbolic Systems Workshop.*

[12] Jordi Armengol-Estapé, Jackson Woodruff, Chris Cummins, and Michael FP O'Boyle. 2023. SLaDe: A Portable Small Language Model Decompiler for Optimized Assembler. *arXiv preprint arXiv:2305.12520* (2023).

[13] Mikel Artetxe, Gorka Labaka, and Eneko Agirre. 2019. An Effective Approach to Unsupervised Machine Translation. In *ACL (1).* Association for Computational Linguistics, 194–203.

[14] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Terry Michael, Quoc Le, and Charles Sutton. 2021. Program Synthesis with Large Language Models. *CoRR* (2021).

[15] Manya Bansal, Olivia Hsu, Kunle Olukotun, and Fredrik Kjolstad. 2023. Mosaic: An Interoperable Compiler for Tensor Algebra. *PLDI* (2023).

[16] Rohan Bavishi, Caroline Lemieux, Roy Fox, Koushik Sen, and Ion Stoica. 2019. AutoPandas: neural-backed generators for program synthesis. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–27.

[17] Sahil Bhatia, Sumer Kohli, Sanjit A. Seshia, and Alvin Cheung. 2023. Building Code Transpilers for Domain-Specific Languages Using Program Synthesis (Experience Paper). In *ECOOP (LIPIcs, Vol. 263).* Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 38:1–38:30.

[18] L Susan Blackford, Antoine Petitet, Roldan Pozo, Karin Remington, R Clint Whaley, James Demmel, Jack Dongarra, Iain Duff, Sven Hammarling, Greg Henry, et al. 2002. An updated set of basic linear algebra subprograms (BLAS). *ACM Trans. Math. Software* 28, 2 (2002), 135–151.

[19] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. TVM: an automated end-to-end optimizing compiler for deep learning. In *Proceedings of the 13th USENIX conference on Operating Systems Design and Implementation.* 579–594.

[20] Bruce Collie, Philip Ginsbach, and Michael FP O'Boyle. 2019. Type-Directed Program Synthesis and Constraint Generation for Library Portability. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT).* IEEE, 55–67.

[21] Bruce Collie and Michael FP O'Boyle. 2021. Program lifting using gray-box behavior. In *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT).* IEEE, 60–74.

[22] Bruce Collie, Jackson Woodruff, and Michael FP O'Boyle. 2020. Modeling black-box components with probabilistic synthesis. In *Proceedings of the 19th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences.* 1–14.

[23] Keith D Cooper and Linda Torczon. 2011. *Engineering a compiler.* Elsevier.

[24] Joao PL De Carvalho, Braedy Kuzma, Ivan Korostelev, José Nelson Amaral, Christopher Barton, José Moreira, and Guido Araujo. 2021. KernelFaRer: replacing native-code idioms with high-performance library calls. *ACM Transactions On Architecture And Code Optimization (TACO)* 18, 3 (2021), 1–22.

[25] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS (Lecture Notes in Computer Science, Vol. 4963).* Springer, 337–340.

[26] Mehdi Drissi, Olivia Watkins, Aditya Khant, Vivaswat Ojha, Pedro Sandoval Segura, Rakia Segev, Eric Weiner, and Robert Keller. 2018. Program Language Translation Using a Grammar-Driven Tree-to-Tree Model. *CoRR* abs/1807.01784 (2018).

[27] Venmugil Elango, Norm Rubin, Mahesh Ravishankar, Hariharan Sandanagobalane, and Vinod Grover. 2018. Diesel: DSL for linear algebra

and neural net computations on GPUs. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. 42–51.

[28] Grigory Fedyukovich, Maaz Bin Safeer Ahmad, and Rastislav Bodik. 2017. Gradual synthesis for static parallelization of single-pass array-processing programs. *ACM SIGPLAN Notices* 52, 6 (2017), 572–585.

[29] Björn Franke and Michael O'Boyle. 2003. Array recovery and high-level transformations for DSP applications. *ACM Transactions on Embedded Computing Systems (TECS)* 2, 2 (2003), 132–162.

[30] Philip Ginsbach, Toomas Remmelg, Michel Steuwer, Bruno Bodin, Christophe Dubach, and Michael FP O'Boyle. 2018. Automatic matching of legacy code to heterogeneous APIs: An idiomatic approach. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. 139–153.

[31] Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. *ACM Sigplan Notices* 46, 1 (2011), 317–330.

[32] Yining Hong, Kaichun Mo, Li Yi, Leonidas J. Guibas, Antonio Torralba, Joshua B. Tenenbaum, and Chuang Gan. 2022. Fixing Malfunctional Objects With Learned Physical Simulation and Functional Prediction. In *CVPR*. IEEE, 1403–1413.

[33] Olivia Hsu, Alexander Rucker, Tian Zhao, Kule Olukotun, and Fredrik Kjolstad. 2022. Stardust: Compiling Sparse Tensor Algebra to a Reconfigurable Dataflow Architecture. *CoRR* (2022). Available at https://arxiv.org/abs/2211.03251.

[34] Shoaib Kamil, Alvin Cheung, Shachar Itzhaky, and Armando Solar-Lezama. 2016. Verified lifting of stencil computations. *ACM SIGPLAN Notices* 51, 6 (2016), 711–726.

[35] Yuning Kang, Zan Wang, Hongyu Zhang, Junjie Chen, and Hanmo You. 2021. APIRecX: Cross-Library API Recommendation via Pre-Trained Language Model. In *EMNLP (1)*. Association for Computational Linguistics, 3425–3436.

[36] Omer Katz, Yuval Olshaker, Yoav Goldberg, and Eran Yahav. 2019. Towards Neural Decompilation. *CoRR* abs/1905.08325 (2019).

[37] Fredrik Kjolstad, Stephen Chou, David Lugato, Shoaib Kamil, and Saman Amarasinghe. 2017. taco: A Tool to Generate Tensor Algebra Kernels. *ASE* (2017).

[38] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The Tensor Algebra Compiler. *OOPSLA* (2017).

[39] Daniel Kroening and Michael Tautschnig. 2014. CBMC - C Bounded Model Checker - (Competition Contribution). In *TACAS (Lecture Notes in Computer Science, Vol. 8413)*. Springer, 389–391.

[40] Taku Kudo and John Richardson. 2018. SentencePiece: A simple and language independent subword tokenizer and detokenizer for Neural Text Processing. In *EMNLP (Demonstration)*. Association for Computational Linguistics, 66–71.

[41] Jeremy Lacomis, Pengcheng Yin, Edward J. Schwartz, Miltiadis Allamanis, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. 2019. DIRE: A Neural Approach to Decompiled Identifier Naming. In *ASE*. IEEE, 628–639.

[42] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, and Shengyu Fu. 2021. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. *CoRR* (2021). Available at https://arxiv.org/pdf/2102.04664.pdf.

[43] Shantanu Mandal, Adhkri Chethan, Vahid Janfaza, S M Farabi Mahmud, Todd A Anderson, Javier Turek, Jesmin Jahan Tithi, and Abdullah Muzahid. 2023. Large Language Models Based Automatic Synthesis of Software Specifications. *CoRR* (2023). Available at https://arxiv.org/pdf/2304.09181.pdf.

[44] Pablo Antonio Martínez, Jackson Woodruff, Jordi Armengol-Estapé, Gregorio Bernabé, José Manuel García, and Michael FP O'Boyle. 2023. Matching linear algebra and tensor code to specialized hardware accelerators. In *Proceedings of the 32nd ACM SIGPLAN International Conference on Compiler Construction*. 85–97.

[45] MG Sarwar Murshed, Christopher Murphy, Daqing Hou, Nazar Khan, Ganesh Ananthanarayanan, and Faraz Hussain. 2021. Machine learning at the network edge: A survey. *ACM Computing Surveys (CSUR)* 54, 8 (2021), 1–37.

[46] Daye Nam, Baishakhi Ray, Seohyun Kim, Xianshan Qu, and Satish Chandra. 2022. Predictive synthesis of API-centric code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*. 40–49.

[47] Michael F. P. O'Boyle and Peter M. W. Knijnenburg. 2002. Integrating Loop and Data Transformations for Global Optimization. *J. Parallel Distributed Comput.* 62, 4 (2002), 563–590.

[48] OpenAI. [n. d.]. ChatGPT. https://openai.com/chatgpt.

[49] Myle Ott, Sergey Edunov, Alexei Baevski, Angela Fan, Sam Gross, Nathan Ng, David Grangier, and Michael Auli. 2019. fairseq: A fast, extensible toolkit for sequence modeling. *arXiv preprint arXiv:1904.01038* (2019).

[50] Davide Pizzolotto and Katsuro Inoue. 2021. Identifying Compiler and Optimization Level in Binary Code from Multipler Architectures. *IEEE Access* (2021).

[51] Elizabeth Polgreen, Andrew Reynolds, and Sanjit A. Seshia. 2022. Satisfiability and Synthesis Modulo Oracles. In *VMCAI (Lecture Notes in Computer Science, Vol. 13182)*. Springer, 263–284.

[52] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *Acm Sigplan Notices* 48, 6 (2013), 519–530.

[53] Joseph Redmon. 2013–2016. Darknet: Open Source Neural Networks in C. http://pjreddie.com/darknet/.

[54] Christopher D Rosin. 2019. Stepping stones to inductive synthesis of low-level looping programs. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 33. 2362–2370.

[55] Baptiste Roziere, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised Translation of Programming Languages. In *Proceedings of the 34th International Conference on Neural Information Processing Systems* (Vancouver, BC, Canada) *(NIPS'20)*. Curran Associates Inc., Red Hook, NY, USA, Article 1730, 11 pages.

[56] Mazen AR Saghir. 1998. *Application-specific instruction-set architectures for embedded DSP applications*. Citeseer.

[57] Eddie Antonio Santos, Joshua Charles Campbell, Dhvani Patel, Abram Hindle, and José Nelson Amaral. 2018. Syntax and sensibility: Using language models to detect and correct syntax errors. In *SANER*. IEEE Computer Society, 311–322.

[58] Ryan Senanayake, Changwan Hong, Siheng Wang, Wilson Amalee, Stephen Chou, Shoaib Kamil, Saman Amarasinghe, and Fredrik Kjolstad. 2020. A Sparse Iteration Space Transformation Framework for Sparse Tensor Algebra. *OOPSLA* (2020).

[59] Kensen Shi, David Bieber, and Rishabh Singh. 2022. TF-Coder: Program synthesis for tensor manipulations. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 44, 2 (2022), 1–36.

[60] Nicholas D Sidiropoulos, Lieven De Lathauwer, Xiao Fu, Kejun Huang, Evangelos E Papalexakis, and Christos Faloutsos. 2017. Tensor decomposition for signal processing and machine learning. *IEEE Transactions on signal processing* 65, 13 (2017), 3551–3582.

[61] Rohit Singh, Rishabh Singh, Zhilei Xu, Rebecca Krosnick, and Armando Solar-Lezama. 2014. Modular synthesis of sketches using models. In *Verification, Model Checking, and Abstract Interpretation: 15th International Conference, VMCAI 2014, San Diego, CA, USA, January 19-21, 2014, Proceedings 15*. Springer, 395–414.

[62] Sunbeom So and Hakjoo Oh. 2017. Synthesizing imperative programs from examples guided by static analysis. In *International Static Analysis Symposium*. Springer, 364–381.

[63] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. 2006. Combinatorial sketching for finite programs. In *ASPLOS*. ACM, 404–415.

[64] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. 2014. Sequence to Sequence Learning with Neural Networks. In *NIPS*. 3104–3112.

[65] Marc Szafraniec, Baptiste Roziere, Hugh Leather Francois Charton, Patrick Labatut, and Gabriel Synnaeve. 2022. Code translation with compiler representations. *arXiv preprint arXiv:2207.03578* (2022).

[66] Abhishek Udupa, Arun Raghavan, Jyotirmoy V. Deshmukh, Sela Mador-Haim, Milo M. K. Martin, and Rajeev Alur. 2013. TRANSIT: specifying protocols with concolic snippets. In *PLDI*. ACM, 287–296.

[67] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *arXiv preprint arXiv:1802.04730* (2018).

[68] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *NIPS*. 5998–6008.

[69] Jackson Woodruff, Jordi Armengol-Estapé, Sam Ainsworth, and Michael F. P. O'Boyle. 2022. Bind the gap: compiling real software to hardware FFT accelerators. In *PLDI*. ACM, 687–702.

[70] Rohan Yadav, Alex Aiken, and Fredrik Kjolstad. 2022. DISTAL: The Distributed Tensor Algebra Compiler. *PLDI* (2022).

[71] Rohan Yadav, Alex Aiken, and Fredrik Kjolstad. 2022. SpDISTAL: Compiling Distributed Sparse Tensor Computations. *International Conference for High Performance Computing, Networking, Storage and Analysis* (2022).

[72] Vojin Zivojnovic. 1994. DSPstone: A DSP-oriented benchmarking methodology. *Proc. Signal Processing Applications & Technology, Dallas, TX, 1994* (1994), 715–720.

[73] Amit Zohar and Lior Wolf. 2018. Automatic program synthesis of long programs with a learned garbage collector. *Advances in neural information processing systems* 31 (2018).

Received 2023-07-14; accepted 2023-09-03