Guess, Measure & Edit: Using Lowering to Lift Tensor Code

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

Jackson Woodruff
University of Edinburgh
United Kingdom
jackson.woodruff@ed.ac.uk

Jordi Armengol-Estapé
University of Edinburgh
United Kingdom
jordi.armegol.estape@ed.ac.uk

Alexander Brauckmann
University of Edinburgh
United Kingdom
alexander.brauckmann@ed.ac.uk

Luc Jaulmes
University of Edinburgh
United Kingdom
ljaulmes@ed.ac.uk

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

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

***Abstract*—Recently, we have observed a steady growth in specialized hardware accelerators. These accelerators are typically programmed in high-level domain-specific languages (DSLs), enabling compilers to generate efficient code for rapidly evolving heterogeneous hardware. However, rewriting existing code to exploit DSL compiler performance is an onerous programmer task. This has led to recent interest in automatically translating or *lifting* code to DSLs. Current lifting techniques use language models or program synthesis to translate code. Although language models have proved remarkably successful in related translation tasks, they are prone to hallucinations. Program synthesis approaches are accurate but do not scale to complex tensor DSLs.**

This paper presents a novel approach, Guess, Measure & Edit; that exploits both language models and compiler technology to lift existing code to high-level DSLs. Given a source program, it uses a language model to *guess* an initial equivalent target program. It then compiles or *lowers* the guess and the original program, and *measures* the low-level distance between them using program similarity metrics. It iteratively uses these low-level metrics to guide high-level *edits* to the guess until it is correct.

To validate this approach, we develop KONRUL which correctly lifts existing tensor algebra C code to einsum notation, the basis of tensor contraction DSLs. Our evaluation shows that KONRUL is fast and accurate, lifting 98% of an extensive benchmark suite and significantly outperforming 4 state-of-the-art lifting schemes. KONRUL is scalable and is the only approach to correctly lift higher-dimensional tensor contraction code. Our lifted programs result in geomean speedups of $4.07\times$ and $38.30\times$ when ported to a multi-core CPU and GPU respectively.

***Index Terms*—Lifting, Language Models, Program Similarity, Guided Edit**

I. Introduction

Recent years have seen a significant increase in heterogeneous parallel architectures. This is driven by the continued need to deliver application performance, particularly for ML workloads, as architecture scaling slows. Frequently, these architectures are accessed using specialized platform-specific APIs or domain-specific languages (DSL). As ML models can be expressed in terms of tensor algebra, this has lead to a wide range of tensor-based DSLs [36], [37], [53], [22], [62], [67], [66], based on Einstein summation (Einsum) notation [24].

Porting legacy code to emerging hardware therefore requires rewriting sections of code in a suitable DSL. Here, the programmer writes their application once, and relies on vendor-supported compilers to generate efficient code for each new platform. Although a one-off activity, this task remains a significant programming cost and deters code migration. This cost has led to recent interest in exploring automated techniques to rewrite or *lift* such code into a higher DSL form [45], [32].

Lifting to Reduce Porting Effort. Existing lifting approaches primarily use different forms of program synthesis [51], [59], [44]. Such approaches have the benefit of being portable and able to lift programs to different target languages. However, they are not scalable in terms of program complexity. As the size of the source or target program increases, the time to synthesize a solution grows exponentially. To overcome this fundamental problem, synthesis approaches rely on strong hard-coded heuristics to prune the candidate space; in some cases requiring an externally provided sketch of the target program [51]. Alternative bottom-up enumerative, input-output synthesis techniques also fail to scale, requiring an excessively large number of candidates [59] and occasionally producing incorrect results [44]. In practice, these approaches cannot synthesize programs with tensors of greater than 2 dimensions.

Coincidentally, there has been increased use of neural machine translation and language models for program translation tasks [63], [17], [23]. In principle, such models could be used for lifting and are highly attractive as their training and deployment can be fully automated. Unfortunately, they require a large amount of training data and are ill-suited to new, low-resource DSLs. More fundamentally, while language models are powerful, they are also inaccurate, hallucinating outputs [54], making them unreliable. Ideally, we would like to exploit the power of language models, while ensuring correct translation.

Our approach. This paper proposes a novel, fast, program lifting methodology which harnesses the power of language models and critically uses existing compilation to

979-8-3315-8295-1/25/$31.00 ©2025 IEEE
DOI 10.1109/PACT65351.2025.00029

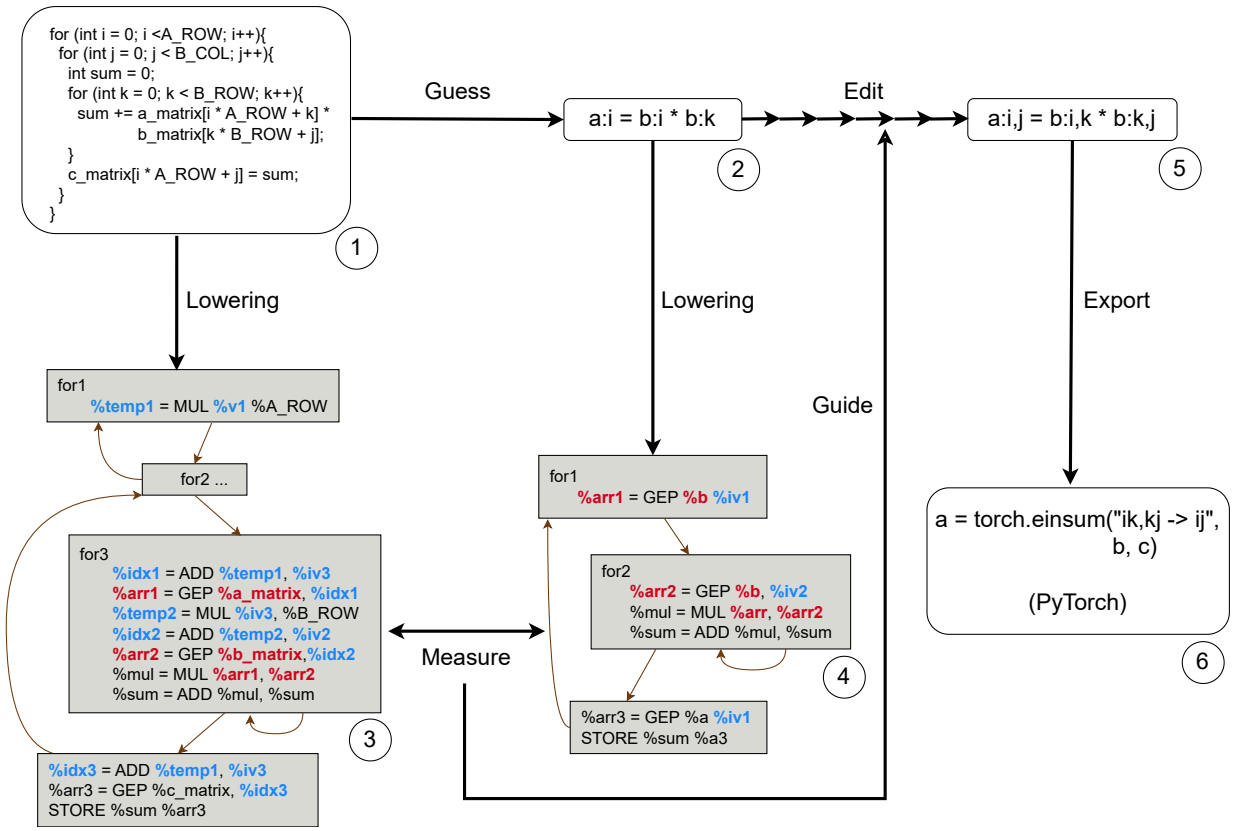


Fig. 1: KONRUL takes as input a C program ① and tries to guess an equivalent einsum program ②. It lowers both to LLVM IR ③, ④ and checks similarity between them. KONRUL uses the result of this check to iteratively edit the guess until it finds an einsum program that is equivalent to the input ⑤.

overcome their limitations.

Specifically, our framework, *Guess, Measure & Edit*, uses a language model to *guess* a high-level candidate solution; uses compiler lowering to enable *measurement* of the low-level distance between the guess and the original program; and then predicts high-level *edits* to apply to the guess to reduce the low-level distance. Finally, the behavior of the new program is formally *verified* to fully match that of the original program.

Our key insight is that while it is difficult to lift a program in low-level language L to a program in high-level language H, we frequently have compiler infra-structure that lowers H to L. As programs that are near each other in H are likely to be compiled to be near each other in L, if can reduce the distance between two low-level programs then it is likely that we also reduce the distance between the high-level ones. This novel insight enables us to use similarity metrics on the low-level language to guide our choice of edit rules to apply to the high-level program. We use this guided editing process to repair incorrect guesses from a language model.

KONRUL. Based on this methodology, we developed KONRUL, a code lifter that translates code written in C to Tensor DSLs, a problem tackled by a wide variety of recent work [44], [51]. This task is both important, as tensor contractions are a fundamental building block of ML workloads, and challenging, as the high-dimensionality of tensors exposes the scalability issues of program synthesis schemes. Tensor DSLs are primarily based on einsum notation [24]. KONRUL therefore targets tensor algebra C programs and lifts them to an einsum program. KONRUL then exports the einsum program to PyTorch [29], which supports this format. We test the lifted programs using automatically generated input-output (I/O) examples to ensure observational equivalence. Furthermore, we perform formal verification of a candidate that passes all tests using the bounded model checker CBMC [38].

We evaluate against 4 state-of-the-art approaches, Tenspiler [51], GPT-4 [4], TF-Coder [59] and C2TACO [44]. We show that KONRUL significantly outperforms existing approaches in terms of the number of programs lifted and scalability. On an extended benchmark suite of 81 programs we are able to lift 98% of them in 23 secs (on average), achieving a 26% to 51% improvement over the baselines.

KONRUL and Tenspiler are found to be the only schemes that consistently generate correctly lifted code. Furthermore, KONRUL is able to scale and is the only approach to lift all higher dimensional tensor contraction code correctly. Overall, the code lifted by KONRUL is able to achieve, on average, a $4.07\times$ to $38.3\times$ speedup over the original implementation, respectively when ported to a multi-core CPU and GPU platform.

This paper makes the following contributions:

- The first verified lifting framework to exploit lowering and source measurement to edit language model hallucinations.
- KONRUL, a fast, scalable and correct C to einsum lifter based on this framework.
- An extensive evaluation of KONRUL against alternative approaches and the state-of-the-art.

II. Guess, Measure & Edit

Formally, given an input program in C, denoted P, our aim is to find an equivalent expression in einsum notation, denoted E. An expression E is a valid solution iff $\forall x.P(x) = E(x)$, where x is a list of tensor inputs.

Our framework relies on the key insight that we can use existing compiler technology to lower P and E to one common intermediate representation, giving a lowered specification $\forall x.IR_P(x) = IR_E(x)$. Since both programs are now represented in a common language, if we obtain an invalid guess $\hat{E}$, i.e., $\exists x.\hat{E}(x) \neq P(x)$, we can use program similarity metrics to syntactically compare $IR_{\hat{E}}$ to IR_P.

We thus execute the following, until we find a valid solution:

1) *Guess* an einsum expression, $\hat{E}$. If $\forall x.\hat{E}(x) = E(x)$ is true, we have a valid solution. If not, proceed to the next step.
2) Lower $\hat{E}$ and E to $IR_{\hat{E}}$ and IR_E and, using a similarity metric Sim, *measure* the syntactic difference between $IR_{\hat{E}}$ and IR_E.
3) If $|Sim(IR_{\hat{E}}) - Sim(IR_P)| \geq \delta$, *edit* $\hat{E}$ and return to step 2. If $|Sim(IR_{\hat{E}}) - Sim(IR_P)| < \delta$, check if $\hat{E}$ is a valid solution, i.e., $\forall x.\hat{E}(x) = E(x)$. If $\hat{E}$ is invalid, *edit* $\hat{E}$ and return to step 2.

Figure 1 illustrates this on an example C program, lifting to einsum notation:

- The input is a C program ① (after it has been checked to see if it is suitable for lifting) that is given to a trained language model.The model outputs the best einsum prediction for this C input as shown in ② (the *guess*).
- This program is lowered using an einsum compiler to generate C code from the guess. As coding styles vary, both the original C program and the guess are lowered further to LLVM `-Oz` IR, which normalizes code, for comparison. The LLVM IR of the original program is shown in ③, while the LLVM of the lowered guess is shown in ④.
- We then *measure* the difference between the two LLVM IR programs using program similarity metrics. In this case, the LLVM programs are different, as highlighted by the fragments shown in ③ and ④. ③ contains three loop nests, while ④ has two. The memory locations in ③ are accessed by indexes that are computed based on different loop induction variables, `iv1`, `iv2` and `iv3` whereas in ④ memory is accessed directly using the value of loop iterators as shown in blue. Furthermore, ③ multiplies values that are references to two different arrays `a_matrix` and `b_matrix` while ④ multiplies two references to the same variable `b`, as highlighted in red.
- The program similarity metrics are then used to predict an *edit* to the einsum notation. This process is repeated to eventually give the correct lifted notation shown in ⑤.
- This einsum program is then executed and checked for I/O equivalence. If successful, it is then verified by model checking before being exported into PyTorch as shown in ⑥.

III. Applying Guess, Measure & Edit to Tensor Code

We implemented the proposed lifting technique in a tool called KONRUL, which lifts C code into einsum notation. KONRUL's architecture is depicted on Figure 2 and the search process described in Algorithm 1. KONRUL uses a Transformer model [64] to *guess* a solution, compiler lowering to infer a specification for the guess (section III-A), program similarity metrics to *measure* the difference between the specifications (section III-B) and guide a search of edit rules (section III-C), and finally testing and model checking to verify the solution (section III-D).

Target language. We use extended einsum notation as our target language. Einstein summation (einsum) notation [24] is a high-level language to express tensor contractions. An einsum program contains an indexing term for each tensor where the indices shared between terms are multiplied, and the indices that are not shared with the left-hand side are implicitly summed. Original einsum notation does not support all operations used in tensor algebra, so DSLs adopt an extended version. The grammar we support, G, is depicted in Figure 3, and our aim is to find a solution E that is in the language of G. Once we have a valid solution, it can then be exported to any DSL that supports einsum notation, e.g., PyTorch [29].

A. Language Model: Guess

KONRUL uses a trained encoder-decoder Transformer [64] to output a predicted einsum notation, $\hat{E}$, for a given tokenized input P as shown in the box labeled `Guess` in Figure 2. Given the low resource availability of einsum programs, we generate a synthetic training set to train the model.

1) Data generation. Transformers require a large data set that captures the domain of interest. However, training data for DSLs is scarce. As the number of existing C, einsum program pairs $\langle P, E \rangle$ is small, we automatically generate then based on a bottom-up enumeration of the grammar shown in Figure 3. The generated einsum expressions are then exported to an einsum compiler [36] which generates equivalent C code.

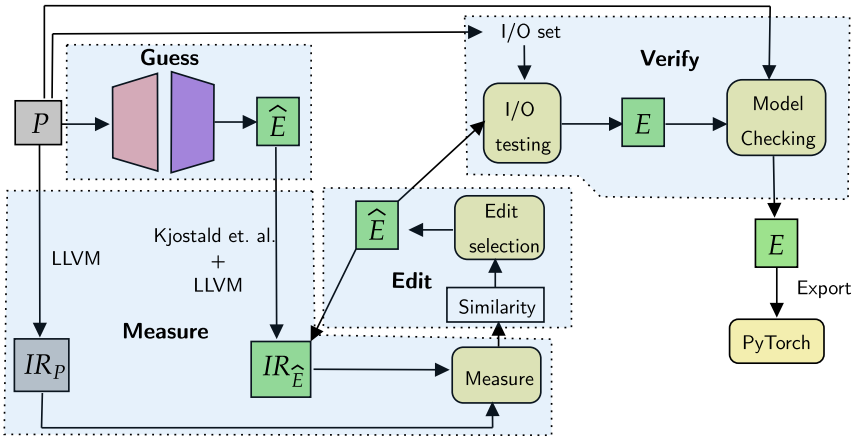


Fig. 2: Overall architecture of KONRUL. `Guess` is described in Section III-A, `Measure` is described in Section III-B, `Edit` is described in Section III-C and finally `Verify` is described in Section III-D. Kernel extraction and I/O generation are based on prior-work.

$$
\begin{aligned}
\textit{program} &::= \textit{tensor} = \textit{expr} \\
\textit{tensor} &::= \textit{id} : \textit{index-expr} \mid \textit{id} \\
\textit{index-expr} &::= \textit{index} \mid \textit{index-expr}, \textit{index} \\
\textit{index} &::= i \mid j \mid k \mid l \mid m \mid n \\
\textit{expr} &::= (\textit{expr}) \mid \textit{expr op expr} \mid -\textit{tensor} \mid \textit{tensor} \mid \\
&\quad \textit{const} \mid -\textit{const} \\
\textit{op} &::= + \mid - \mid * \mid / \\
\textit{id} &::= T_0 \mid T_1 \mid ... \\
\textit{const} &::= C_0 \mid C_1 \mid ...
\end{aligned}
$$

Fig. 3: The einsum grammar G, which KONRUL uses to synthesize programs. Programs expressed in this grammar can be exported to tools such as PyTorch's Einsum mode.

As the program space is unbounded, we limit both the number of tensors and their dimensionality to 5. This gives a dataset of approximately 800k $\langle P, E \rangle$ pairs.

The main problem is that the C code is generated by a compiler, not a human, and does not represent real-world hand-written legacy programs. We therefore pre-process each generated program before it is used in training, eliminating wrapper code and then normalizing identifiers in the program making them one-character long following lexicographical order. In addition, we replace constants and loop bounds with fixed symbols e.g. *CONS* and *DIM* to prevent many similarly generated programs differing just by a constant value polluting the data set. This produces code that is better for training as it is syntactically closer to the original code.

2) Training. We used the processed 800k $\langle P, E \rangle$ pairs for training, separating 5k for validation and 5k for testing. To ensure fairness, we ensured that none of the evaluated benchmark programs used in section IV were part of the training data. The programs were then tokenized using byte-pair-enconding (BPE) [58] before being used for training. We trained a 254M-parameter Transformer, with 6 encoder layers and 6 decoder layers, 16 attention heads and an embedding and context size of 1024. It shares embeddings across the encoder, decoder and output layer, and uses an Adam optimizer in place of gradient descent update [35].

3) Inference. At inference time, we apply the same pre-processing steps to the unseen source C program before tokenization. We use standard beam search decoding with a beam size of 5 and use the best prediction of einsum output as our initial guess. We replace any constant tokens (*CONS*) with a small random integer before passing on to the next stage.

B. Program Similarity: Lower and Measure

Given a kernel program P and an einsum guess $\hat{E}$, we wish to estimate how similar the einsum expression is to P. As shown in box `Measure` in Figure 2, to facilitate the measure task, we lower both programs to a common abstraction level. We lower the expression $\hat{E}$ using an einsum compiler [36] to generate a program in C, denoted $C_{\hat{E}}$. We then lower both P and $C_{\hat{E}}$ to LLVM -Oz IR, giving IR_P and $IR_{\hat{E}}$. The Oz version of LLVM IR was chosen as it is the most terse and has the strongest normalization effect.

We developed three similarity metrics to analyze program features relevant to tensor computation: variable similarity, indexing similarity and arithmetic operator similarity. They intuitively capture the number of variables, matrix indices, and operator applications, occurring in $IR_{\hat{E}}$ and IR_P, normalized by the length of IR_P.

1) Variable Similarity. The first similarity metric, Sim_{vars} compares the number of relevant variables in IR_P with

Algorithm 1: KONRUL search and edit algorithm. Procedure $Edit$ is described in Algorithm 2.

input : source computation kernel P, LM guess $\hat{E}$
output: lifted einsum program, or no solution
Algorithm *search(P, $\hat{E}$)*
  $candidates \leftarrow \emptyset$;
  $\phi_{IO} \leftarrow generateIO(P)$;
  $IR_P \leftarrow lower(P)$;
  while *not timeout* **do**
    $IR_{\hat{E}} \leftarrow lower(\hat{E})$;
    var-score $\leftarrow Sim_{vars}(IR_{\hat{E}}, IR_P)$;
    op-score $\leftarrow Sim_{ops}(IR_{\hat{E}}, IR_P)$;
    idx-score $\leftarrow Sim_{index}(IR_{\hat{E}}, IR_P)$;
    score $\leftarrow$ *var-score* $*$ *op-score* $*$ *idx-score*;
    if *score* $= 1$ **then**
      if *Check*(E, ϕ_{IO}, P) **then**
        return E
    else if *score* $> threshold$ **then**
      $candidates \leftarrow candidates \cup E$;
    $\hat{E} \leftarrow$ *Edit*$(\hat{E}, IR_{\hat{E}}, IR_P,$ *var-score*, *op-score*, *idx-score*$)$;
  for $c \in candidates$ **do**
    if *Check*$(c, \phi_{IO}, P) \wedge$ *Verify*(c, P) **then**
      return c
  return *no solution*

Algorithm 2: Procedure $Edit$ takes both the original program and a candidate and selects which edit to apply based on the similarity metrics.

Procedure $applyEditRule\,(\hat{E}, IR_{\hat{E}}, IR_P,$ *metric*$)$
  case *metric* $= Sim_{vars}$:
    if $|tensors(IR_{\hat{E}})| < |tensors(IR_P)| \Rightarrow addT(\hat{E})$
    elif $|tensors(IR_{\hat{E}})| > |tensors(IR_P)| \Rightarrow deleteT(\hat{E})$
    else $|tensors(IR_{\hat{E}})| = |tensors(IR_P)| \Rightarrow renameT(\hat{E})$
  case *metric* $= Sim_{index} :\Rightarrow indexing(\hat{E}, IR_P)$
  case *metric* $= Sim_{ops}$:
    if $|tensors(IR_{\hat{E}})| > 2 \Rightarrow op(\hat{E}, IR_P)$
    else $|tensors(IR_{\hat{E}})| \leq 2 \Rightarrow sign(\hat{E}, IR_P)$

Procedure $Edit\,(\hat{E}, IR_{\hat{E}}, IR_P,$ *var-score, op-score, idx-score*$)$
  case *var-score* $\neq 1 :\Rightarrow metric \leftarrow Sim_{vars}$
  case *idx-score* $\neq 1 :\Rightarrow metric \leftarrow Sim_{index}$
  case *op-score* $\neq 1 :\Rightarrow metric \leftarrow Sim_{ops}$
  $applyEditRule(\hat{E}, IR_{\hat{E}}, IR_P, metric)$
  return $\hat{E}$

the number of variables in $IR_{\hat{E}}$, denoted $Var(IR_P)$ and $Var(IR_{\hat{E}})$ respectively:

$$Sim_{vars} = max(1 - \frac{Var(IR_P) - Var(IR_{\hat{E}})}{Var(IR_P)}, 0).$$

A relevant variable is any variable in the program that is not an induction variable or loop bound. Intuitively, this metric returns 1 if the programs have the same number of variables, 0 if $IR_{\hat{E}}$ has too many variables and a number between 0 and 1 if $IR_{\hat{E}}$ has too few variables. Note that the number of variables may differ even in semantically equivalent programs because the lowered einsum program may contain auxiliary variables introduced by the compiler. We normalize variable names in both programs, i.e., we replace identifiers with $A, B, C, \ldots$ so any program with 3 variables will have the same 3 variable names.

2) Indexing Similarity. The second metric, Sim_{index}, considers the index expressions used in the program. We use Polly's [28] polyhedral analysis to obtain a list of index expressions. We augmented Polly's analysis to handle constants within C structures.

Given a program IR, we first extract the set of index variables used, $\{i_1, \ldots, i_k\}$. Let $M[e_1, \ldots, e_m]$ denote the matrix index expression where the variable M is indexed with expressions $e_1, \ldots, e_m$. For each matrix index expression, we generate a corresponding matrix index tuple $C = (c_1, \ldots, c_k)$, where c_1 is 1 if i_1 appears in e_1, and 2 if i_1 appears in e_2 and so on, and 0 if it does not appear in any index expressions. For a program IR that contains n matrix index expressions, we extract a list of matrix index tuples in the order that they occur: $Indices(IR) = \{C_1, \ldots, C_n\}$. Similar to the previous metric, Sim_{index} between two programs $IR_{\hat{E}}$ and IR_P is calculated as:

$$Sim_{index} = 1 - \frac{D(Indices(IR_{\hat{E}}), Indices(IR_P))}{|indices(IR_P)|},$$

where D is the Levenshtein Distance [43] distance between the two lists, i.e., the number of substitutions/insertions/deletions needed to change $indices(IR_{\hat{E}})$ into $indices(IR_P)$. This is computed using the Wagner-Fischer algorithm [46].

3) Arithmetic Operation Similarity. The final metric, Sim_{ops}, compares the mathematical operations occurring in each program. Let $App = (Op, x_1, \ldots x_m)$ be a tuple that denotes the operator Op is applied to the operands $x_1, \ldots x_m$. For a program IR that contains n operator applications, we extract a list of operator application tuples from the innermost loops in the lowered program, in the order that they occur: $Ops(IR) = \{App_1, \ldots, App_n\}$. We use $|Ops(IR)|$ to indicate the length of the list $Ops(IR)$, which considers the length of each operator application tuple added.

Sim_{ops} between two programs IR_P and $IR_{\hat{E}}$ is then defined as:

$$Sim_{ops} = 1 - \frac{D(Ops(IR_{\hat{E}}), Ops(IR_P))}{|Ops(IR_P)|},$$

where D is again the Levenshtein distance.

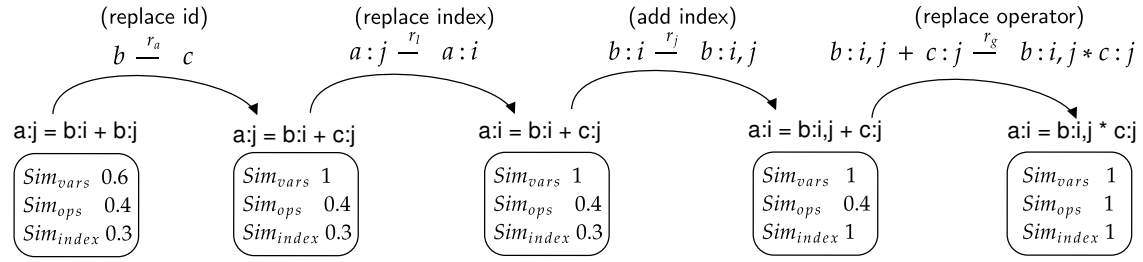


Fig. 4: Edit rule selection based on program similarities. KONRUL first selects rules based on the variable similarity, secondly it analyzes indexing similarity and finally apply rules to edit arithmetic operators.

$id \xrightarrow{r_a} id^*$	(replace id)
$(const\|tensor) \xrightarrow{r_b} (const\|tensor)^*$	(replace const/tensor)
$expr \xrightarrow{r_c} (expr\ op^*\ tensor^*)\|(tensor^*\ op^*\ expr)$	(add tensor)
$expr \xrightarrow{r_d} (expr\ op^*\ const^*)\|(const^*\ op^*\ expr)$	(add const)
$expr\ op\ tensor \xrightarrow{r_e} expr$	(remove tensor)
$expr\ op\ const \xrightarrow{r_f} expr$	(remove const)
$expr\ op\ expr \xrightarrow{r_g} expr\ op^*\ expr$	(replace operator)
$(tensor\|const) \xrightarrow{r_h} -(tensor\|const)$	(negate tensor/const)
$-(tensor\|const) \xrightarrow{r_i} (tensor\|const)$	(un-negate)
$id{:}index\text{-}expr \xrightarrow{r_j} id : index\text{-}expr, index^*$	(add index)
$id \xrightarrow{r_k} id : index^*$	(add index)
$id{:}index\text{-}expr,\ index \xrightarrow{r_l} id : index\text{-}expr$	(remove index)
$index \xrightarrow{r_m} index^*$	(replace index)

Fig. 5: The set of parameterized edit rules used by KONRUL. The symbols $expr$, $tensor$, op, *index-expr* and id correspond to the einsum expressions defined in the grammar in Figure 3. An asterisk (*) indicates that the expression has been introduced or changed by the edit rule.

If the candidate matches IR_P according to all three similarity metrics, that is, a candidate with an overall score equal to 1, the *edit* phase is skipped, and the candidate is passed forward to the *check* stage of the pipeline. Any candidates with a sufficiently high score are stored in a set of backup candidates to be checked if the search terminates without finding a candidate with a score of 1.

Example. The example depicted in Figure 4 shows the similarity scores for various candidate solutions when compared to a standard C implementation of a dense matrix-vector product.

C. Edit

Consider the box labeled `Edit` in Figure 2. If the similarity metrics determine that the lowered guess is far from the lowered original source, they are used as a guide to edit the einsum guess. An edit changes one or more elements of the einsum notation which is then lowered once again which is returned to the *measure* phase as the new candidate solution.

The set of parameterizable edit rules is shown in Figure 5. A key feature of these edit rules is that they are not semantics-preserving, unlike traditional rewrite rules used by compilers, which enables them to transform the potentially semantically incorrect initial guess into a semantically correct einsum expression. At each iteration, we select a similarity metric, and apply edit rules targeting that specific similarity metric, as shown in Algorithm 2.

KONRUL starts editing based on the variable similarity (with metric Sim_{vars} and rules r_a–r_f), because this gives us the correct size of the program before exploring further. In the algorithm, *addT* is a random choice between rules r_c and r_d, *deleteT* is a random choice between r_e and r_f, and $renameT$ is a random choice between r_a and r_b (noting that we choose rules that modify constants only if there are constants present in P).

Second, we repair the index operators, using metric Sim_{index}. In each iteration, we fix the first mismatched index returned by the polyhedral analysis, using an appropriate rule selected from r_j–r_m

Finally, we repair the operators used to act on the tensors, using metric Sim_{ops}. In the algorithm, op randomly chooses and replaces an operator using r_g, and $sign$ randomly chooses a tensor, and then applies either r_h or r_i. We edit the arithmetic operators last because that is the largest gap between einsum and IR representations, which makes the corresponding metric often imperfect.

Example. Figure 4 shows a sequence of edits selected by KONRUL to lift a matrix-vector product in einsum from an incorrect guess. At each step KONRUL analyzes each similarity score as described above, and selects a rule accordingly.

D. Check (Testing and Verification)

The final stage of the process is to check for correctness denoted by the `Verify` box in Figure 2. Once $IR_{\hat{E}}$ is determined to be sufficiently similar to IR_P, according to the similarity metrics, the *Check* call in Algorithm 1 tests that the result is correct. We do this using observational equivalence [25] — testing that the new einsum program produces the same results as the original C program. We implement automatic generation of input-output pairs from the original kernel implementation P, based on previous work [44]. This is a fast and scalable way to filter out incorrect candidates.

To provide stronger guarantees of correctness, we then verify the candidate using bounded model checking. We lower the original C code into MLIR [40], and we lower the einsum into MLIR using JAX [30]. We then generate a C file, which initializes two copies of a set of nondeterministically assigned

inputs, executes the original C code on one copy of the inputs, and the lowered einsum code on the other copy, and asserts that the outputs must be equal. We use CBMC [38], a bounded model checker for C programs, to verify that this assertion is never violated. Given the undecidability of the problem, we place two limits on this verification: first, we limit the size of the input matrices to a fixed bound; second, in (the small number of) cases where verification using IEEE floating-point semantics exceeds a time-out, we use verification with real numbers in place of the floating-point representation.

IV. Experimental Setup

A. Environment

Benchmarks. We gathered a suite consisting of 81 benchmarks used by different prior work on lifting tensor code [44], [51]. The benchmarks come from existing applications, benchmark suites and high-performance libraries that cover a variety of domains including image processing [6], digital signal processing [3], [68], [57], mathematical functions [2], array programming [60], [56], and deep learning [55], [1]. They include a wide ranging of programming styles and optimizations including unrolled loops and post-increment pointer addressing. We also include higher-dimensional tensor contraction programs described in [16].

Platform. Both original and lifted programs were executed on an 36-core Intel Xeon W-2285 mlti-core CPU at 3.00GHz with 125 GB of RAM (DDR4) at 2666 MT/s and an Nvidia RTX A6000 GPU using driver version 510.47.03 and CUDA runtime version 11.6. KONRUL's language model is implemented using Fairseq [48] version 0-12.2 with Google's SentencePiece [39] tokenizer. We use TACO version 0.1, LLVM version 14.0, PyTorch version 2.3.0, and gcc version 9.4.

B. Competitive techniques

We compare KONRUL to a set of prior published lifting techniques.

- **TF-Coder** [59]: neural-guided bottom-up synthesizer for TensorFlow programs.
- **C2TACO** [44]: bottom-up enumerative synthesizer for TACO [36] programs that uses static program analysis to drive search.
- **Tenspiler** [51]: verified-lifting-based approach that lifts tensor code written in low-level languages such as C++ and Python to different tensor API/DSLs
- **GPT-4** [4]: general-purpose large language model. We evaluate it on code generation experimenting with different prompts to maximize its accuracy.

We also implement variations of KONRUL.

- **Greedy**: greedily edits the einsum guess. At each iteration it randomly edits the current best candidate a number of times, lowers, and selects the nearest program according to the similarity metrics for the next iteration.
- **LM**: just uses the Transformer model to predict the einsum program. This baseline allows isolation of the impact of KONRUL's measure and edit phases.

C. Methodology

Each technique was given a time budget of 2 minutes to lift each program. We convert TF-Coder, C2TACO and Tenspiler output programs into einsum to allow direct and fair comparison. KONRUL was given the best einsum guess from the Transformer. We included its lifting and testing times in reported results. As it has to evaluate many candidates, we provide TF-Coder with a small I/O example, enabling it to significantly lift more candidates than reported in previous work [44]. Tenspiler was evaluated using the hand-written grammars provided in the reproducibility artifact [52]. GPT-4 was repeatedly tested with different prompts to find the best recall. The alternative baselines rely on a random component, so we selected the best performance over 10 runs as a competitive baseline. To ensure a fair evaluation, when evaluating the performance, we use gcc and PyTorch compilers (for each approach) and, unless otherwise stated, report the best performance achieved. We reported speedup as the ratio of the lifted einsum program running time over the original implementation compiled with gcc `-O3`. The speedup achieved by every lifting method is the geometric mean of the speedup of each benchmark that said method can lift. All programs lifted and unlifted are used to calculate the geomean speedup. If a program is unlifted, it has a speedup of 1 by definition.

V. Results

A. Coverage

Figure 6 shows the percentage of programs successfully lifted by each technique across the benchmark categories. KONRUL lifts 98% of the programs across the entire suite, lifting 100% of the benchmarks in 9 categories. It fails to lift 2 programs due to LLVM common sub-expression elimination during LLVM lowering, which optimizes away references, suggesting the need for a more sophisticated similarity metric.

C2TACO performs well in most categories and lifts 72% of the benchmark suite, though is outperformed by both Tenspiler and TF-Coder on `blend`, `DSPStone`, and `simpl_array`. Tenspiler is almost successful as C2TACO, lifting 70% of the benchmarks, being the most effective method on `blend`. Tenspiler's symbolic reasoning is effective on many benchmarks, but it struggles to tackle more complex programs involving large loop depths and tensors with high-dimensionality. TF-Coder achieves 62% of coverage, but its enumerative search does not scale when the search space becomes too large. 94% of TF-Coder failures are timeouts and the remaining are semantically wrong programs. GPT-4 has the lowest coverage value of 47%. Although it can always produce a solution in time due to its neural-based generation, it produces invalid answers in the majority of the benchmark suites. Semantically wrong programs represent 86% of its failures while 14% are syntactically invalid.

Correctness. Only KONRUL and Tenspiler are able to provide correctly lifted programs in all cases. Both the enumerative schemes C2TACO and TF-Coder occasionally fail due to their reliance on I/O examples as specifications

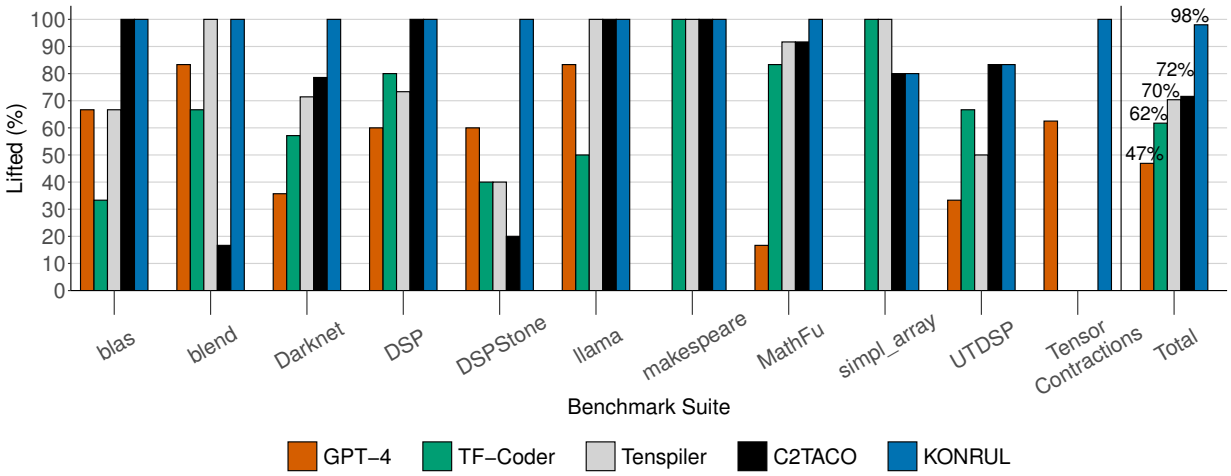


Fig. 6: Lifting coverage across different benchmark categories. Y-axis shows the percentage of programs lifted by each approach in each category listed on the X-axis. The TOTAL group shows the average across the whole benchmark suite.

of the target. C2TACO generates syntactically correct but semantically incorrect code in 2 cases due to insufficient coverage by its I/O examples. As an example, in one case it generates $C = AB$ rather than $C = AB+C$, assumming the C matrix is initialized to 0. TF-coder relies on a small number of examples to speedup its search, but in two separate cases these leads to incorrectly lifted code. GPT-4 unsurprisingly gives the largest number of semantically incorrect translations, 37, due to the well known problem of LLM hallucinations.

1) Coverage by Program Complexity.. Figure 7 shows lifting success rate as function of the highest dimensioned tensor in the lifted program. Apart from GPT-4, all methods have good coverage on the benchmarks with dimensionality 1. KONRUL lifts 98% of those benchmarks, while Tenspiler and TF-Coder lift 89.7% and C2TACO lifts 87.7%. On dimensionality 2, KONRUL is the only method that maintains high coverage, lifting 96%. Tenspiler and GPT-4 can both lift 60% while C2TACO lifts 52%. TF-Coder performance degrades and is only able to lift 24% of the benchmarks.

For the benchmarks performing operations on high-dimensionality tensors with 3 or 4 dimensions, only GPT-4 and KONRUL successfully lift any programs. This shows that enumerative techniques scale poorly since the program search space grows exponentially with dimensionality. GPT-4 achieves 80% and 50% of success rate respectively while KONRUL lifts all the programs in both categories. This result shows that language models are scalable and that using their output as an initial guess is useful to handle more complex programs.

B. Program Space Exploration

This section evaluates efficiency in terms of the number of candidates explored during search. GPT-4 is not considered as it does not explicitly search a candidate space. Tenspiler is also excluded as we cannot determine the number of candidates

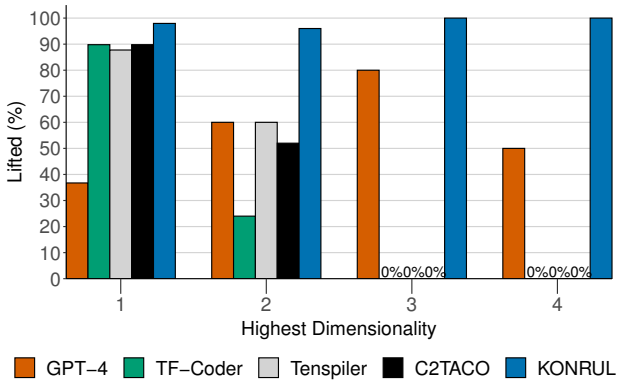


Fig. 7: Lifting coverage by program complexity. Y-axis shows the percentage of programs lifted given different values of highest-dimensionality listed on the X-axis.

the underlying SMT solver considers. Instead, we evaluate KONRUL against Tenspiler in terms of lifting time.

1) I/O-based Techniques. Figure 8 evaluates the efficiency of the various I/O-based techniques by plotting the number of programs lifted vs the number of candidate programs explored.

KONRUL lifts 79 benchmarks, 78 of them by exploring a maximum of 39 candidates. Of those 78, KONRUL lifts 50 by exploring fewer than 12 candidates and 30 with fewer than 6 candidates. KONRUL lift one further program by exploring 93 candidates, due its large number of bracketed sub-expressions. Out of the 56 benchmarks that C2TACO lifts, 81% are found visiting up to 10 candidates. However, when C2TACO explores more candidates it is only able to increase its coverage by 11 programs and is limited by an exponential search space. TF-Coder can lift 14 benchmarks when evaluating 1000 candidates, and 42% of the benchmarks when evaluating up to 10000 candidates. The alternative baseline approach Greedy,

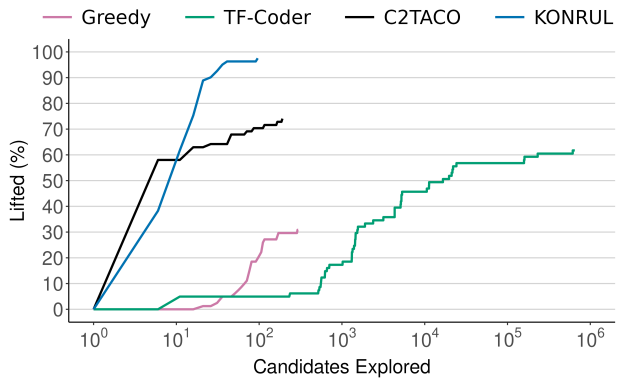


Fig. 8: Cumulative number of candidates explored during lifting by different I/O-based techniques. X-axis shows cumulative number of candidates explored and Y-axis shows the number of programs lifted. X-axis is on logarithmic scale.

performs poorly and although it explores fewer than 1000 candidates, it only lifts 25 programs. Using the metrics just to check if exploration is proceeding towards a solution is not enough to scale. Instead, KONRUL directly employs the values to decide what candidate to explore at every iteration.

2) The Impact of Similarity Metrics.. As explained in Section III, KONRUL does not execute any of the candidates explored during search. Instead, it only tests candidates that are near the original program according to our similarity metrics. Table I shows the average number of candidates explored and the number of candidates tested by KONRUL in different categories. Although the number of programs explored by KONRUL is on average 11.2, the number of candidates that are actually tested with I/O examples is on average 2.4. In fact, in 70% of the cases KONRUL tested only one candidate, which shows that the similarity metrics are a useful way to identify good candidates during search. In all cases, where the similarity metrics equal 1, the candidate is always verified as correct.

3) Evaluation Against Tenspiler. Tenspiler can lift more benchmarks than KONRUL given a very small time budget due to its externally provided target program sketches. Tenspiler lifts 13 benchmarks in under 1 second, while KONRUL's smallest lifting time is 1.5 seconds.

Nevertheless, KONRUL lifts more benchmarks in comparison to Tenspiler. Tenspiler reaches a maximum of 56 benchmarks lifted under 30 seconds. Although KONRUL lifts 55 under the same limit, if given a larger budget of 2 minutes, Tenspiler can only lift one additional benchmark while KONRUL can successfully lift 24 more. In total, KONRUL is lifts 79 programs against 57 by Tenspiler. Tenspiler's symbolic search struggles to reason about programs with large loop nesting levels (> 2), which is common in code with complex contractions that manipulates tensors with high-dimensionality.

TABLE I: Average number of candidates explored vs candidates tested with I/O across different benchmark categories.

Category	Explored	I/O Tested
`blas`	6	1
`blend`	35.6	8.3
`Darknet`	11	2.8
`DSP`	6.77	1.37
`DSPStone`	6.1	1.2
`llama`	16.8	1
`makespeare`	5.2	1
`MathFu`	7.4	1.9
`simpl_array`	4.9	1
`UTDSP`	7.5	1.8
`Tensor Contractions`	15.3	2.75
Mean	11.2	2.4

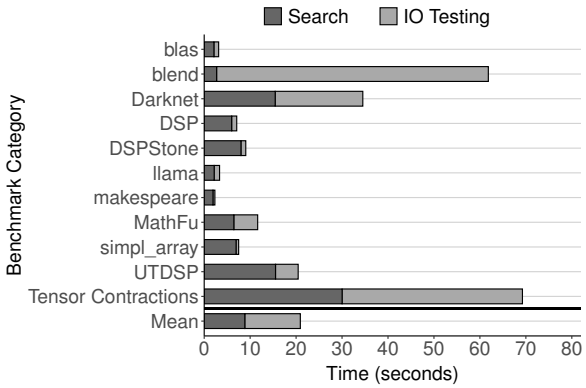


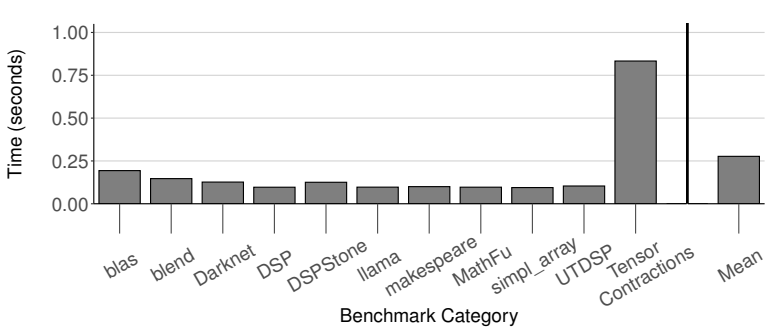
Fig. 9: Lifting time breakdown of KONRUL. Y-axis shows different benchmarks categories with the average value at the bottom. X-axis shows the time taken by KONRUL to search for candidates and test the ones near to the original C program.

4) Extended Timeout. As a form of limit study, we extended the timeout of 2 minutes to 24 hours to see how many additional programs Tenspiler and C2TACO were able to lift. In 24 hours, Tenspiler is not able to lift any additional programs while C2TACO is able to lift another 5. In both cases, none of the high dimensional tensor contractions were lifted. In fact, in some cases C2TACO was still constructing its search space and had not started checking candidates.

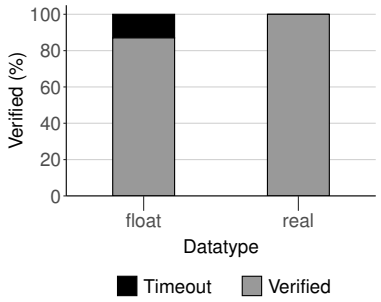
C. Detailed Evaluation of Lifting Time

Figure 9 shows the average time spent in KONRUL's search phase, that is, guessing, measuring and editing, and the time taken to test the final candidates, for different benchmark categories. KONRUL takes an average of 23 seconds to lift a program, of which 10 seconds correspond to search and 13 to testing.

Search time is usually longer than testing except for `blend`, `Darknet`, and `Tensor Contraction` programs. For `blend` and `Darknet` are the categories in which KONRUL I/O tests more than one candidate more often, 83% and 27% respectively. The `Tensor Contraction` benchmarks perform heavy computation on large tensors and the inputs are



(a) Average verification time across different benchmark categories.



(b) Verification coverage with floating points and real datatypes.

Fig. 10: CBMC verification results. 10a shows the average time taken to verify a program across different categories. 10b shows the verification success rate using two different datatypes on all the programs KONRUL lifts.

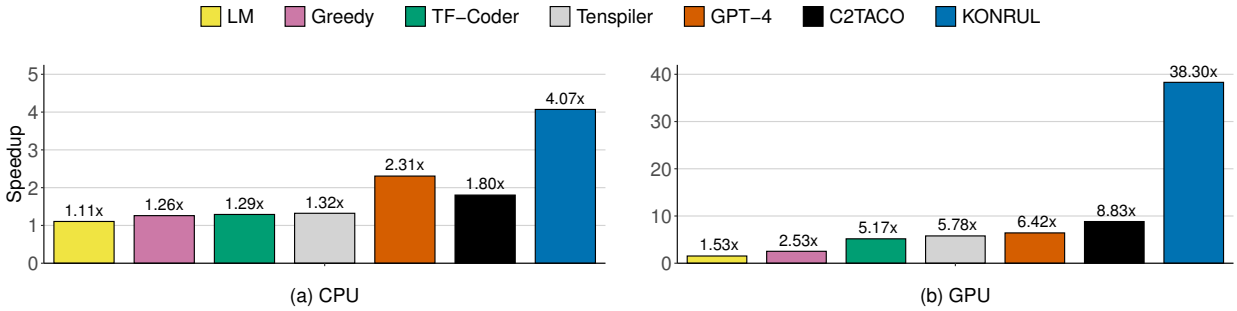


Fig. 11: Geomean speedup obtained by lifted programs on different platforms. We report speedup as the ratio between the running time of the einsum program over the original implementation. In each case the performance achieved is assuming using the best compiler gcc or PyTorch.

constant sizes. Testing time can be reduced by using smaller inputs.

D. Verification

We measure the time taken by CBMC to verify a lifted program. As shown in Figure 10a, the overall time is small with an average of 0.27 seconds. The longest time taken is 0.83 seconds on the `Tensor Contractions` programs. These programs contain higher-dimensional tensor operations in which bounded model checking can take up to 5.2 seconds. The fastest verification occurs `DSP`, `llama` and `simpl_array` categories with an average time of 0.09 seconds.

We verify lifted programs using two different datatypes: floating points and reals. However, verifying equivalence with floating point is challenging [10], and a small number of benchmarks exceed a timeout (this does not mean that the solutions are incorrect, simply that CBMC has not managed to prove equivalence). We thus extend CBMC to support arrays of real numbers, which we internally represent as rationals. Figure 10b shows the percentage of benchmarks successfully verified as correct. We prove floating point equivalence for solutions to 69/79 (87%), and the remaining 10 timeout. We are, however, able to prove that 100% of the translations are correct with real data types.

E. Speedup

Lifting code to Einsum notation enables us to leverage compilers that support said format to optimize tensor operations and generate fast code for different hardware platforms. We exported the lifted programs to PyTorch and executed them on a multi-core CPU and a GPU. Figure 11 depicts a summary of the geo-mean speedups achieved by each method on both platforms.

Performance gains are always higher when lifted code is executed on GPU. KONRUL significantly outperforms all other techniques as it is able to lift more of the benchmark suite. It achieves an overall speedup of $4.07\times$ on the CPU and $38.3\times$ on the GPU. This is primarily due to the large speedups available when executing higher-dimensionality optimized tensor code. TF-Coder performs relatively poorly on the CPU, achieving a speedup of $1.29\times$, but improves on the GPU providing a $5.17\times$ speedup. Tenspiler performs similarly, providing $1.32\times$ and $5.78\times$ speedups respectively. Finally, C2TACO achieves $1.80\times$ speedup on CPU and $8.83\times$ on GPU while GPT-4 reaches $2.31\times$ and $6.42\times$ respectively. Although C2TACO and Tenspiler are able to lift more programs than GPT-4, these do not result in meaningful performance on the CPU. However, on the GPU, this results in a significant im-

provement. The variation baselines approaches provide modest impact on both platforms as they achieve low coverage on the benchmark suite.

VI. Related Work

Neural Machine Translation. NMT has been applied to lifting-related translation tasks, in particular decompilation, lifting binary code into an IR or higher level language [33], [15], [9], [8]. Studies generally show that these models are prone to errors in their translation [49]. As a result, a number of error correction techniques have been developed [34], [26]. In [41], a language model (LM) is combined with symbolic solvers to automatically translate between ARMv8 and RISC-V assembly. It uses low confidence on translated tokens to initiate an SMT-based sketching process to search for a better solution. While the accuracy achieved is modest, it is an interesting orthogonal approach to improving translations. Another approach is to use NMT to learn repairs [18], where a sequence-to-sequence method is proposed to fix bugs in Java programs.

Synthesis for Lifting. Early work used synthesis to determine the behavior of an accelerator API [21], [19], [20] to enable later pattern matching and replacement of user code with library calls [27]. It was extended to mediate the mismatch between user code and accelerator hardware [65]. More generally, using program synthesis to generate one program that is equivalent to another has been explored in super-optimization [50], deobfuscation [31] and program lifting. It has been used for lifting in a number of domains from stencil codes [45], [32], to MapReduce [5], and digital signal processing [7].

General frameworks like MetaLift [11] enable the development of these lifters, and have been applied to matrix multiplication and convolution [47]. Metalift is used by Tenspiler [51] to lift sequential code to tensor processing languages. Its core is an IR that makes synthesis tractable by abstracting operations commonly used in tensor computations, as well as an underlying symbolic synthesizer that generates the target program and an invariant that proves the program is correct. They however require the user to define the syntax and semantics of their target language as well as a set of compilation rules. Recent work [12] uses few-shot learning with GPT-4 within a verified lifting framework, to suggest both candidate solutions and the invariants. Our work is complementary to this approach, as our novel synthesis method would allow guided repair of any incorrect guesses from the LLM.

TF-Coder [59] and C2TACO [44] both perform bottom-up enumerative synthesis to search the space of programs, using user-provided input-output examples as specifications. They use type information, compiler analysis and redundancy checking to narrow the search. mlirSynth [14] uses the same method as C2TACO in the MLIR [40] IR rather than DSL space. Like all enumerative approaches, these methods fail to scale to large tensor programs. A recent alternative approach [13] employs top-down sketching to overcome scalability at the cost of potentially finding sub-optimal solutions.

General Synthesis Algorithms. The majority of synthesis algorithms are based on Counter-Example Guided Inductive Synthesis (CEGIS) [61]. Here, the only feedback is a single input on which a proposed candidate fails. On receiving a counterexample, CEGIS discards the current candidate program without attempting any repair. In contrast, the *Guess, Measure & Edit* framework obtains more detailed feedback from the similarity metrics and attempts to repair the candidate, rather than discarding it, resulting in a more efficient search.

Some prior synthesis work uses metrics to guide the search. The following two works construct programs by enumeration. Euphony [42] learns a probabilistic higher-order grammar from a training data set, and then uses this to guide an A^* search in the space of the production rules of the grammar. Effectively, the cost metric used by the A^* search estimates the total cost of the production rules needed to turn a partial program into a complete one. SyMetric [25] enumerates the space of programs with bottom-up search and then clusters these programs into equivalence classes based on an expert-provided distance metric. The metric is focused on the *semantic* similarity of the input/output behavior of programs, rather than our metrics which focus on the *syntactic* similarity. Whilst this approach is effective for the domains on which it is evaluated, it is not a good fit for einsum programs: first, because very small syntactic differences in einsum programs can result in substantially different semantic behavior; second, due to the expressivity of our grammar, the space of programs that would be enumerated with bottom-up search before clustering could begin would be impractically large.

VII. Conclusion

This paper presented a new lifting methodology, *Guess, Measure & Edit*, that exploits the power of language model based translation and overcomes its shortcomings using compiler technology. Based on this methodolgy, we implemented KONRUL, a program lifter capable of taking legacy C code and delivering high performance on CPU and GPU platforms. We compared KONRUL against 4 state-of-the-art lifting approaches and show that is more accurate and scales better than both enumerative synthesis and language models. On a extended benchmark suite, KONRUL lifts 98% of the benchmarks exploring an average of 11.2 candidates during search and taking an average of 23 seconds. KONRUL's lifted programs can be easily ported to parallel architectures and achieve a speedup of $4.07\times$ on a multicore CPU and $38.30\times$ on a GPU platform.

Currently, our implementation has similarity metrics and edit space specialised to the tensor domain. Future work will investigate a more generic similarity and edit space, going beyond tensor DSLs.

Acknowledgements

José Wesley de Souza Magalhães and Alexander Brauckmann are partly sponsored by Huawei Research. We thank the reviewers for their insightful comments.

References

[1] llama2.cpp. https://github.com/leloykun/llama2.cpp/.

[2] Mathfu. https://github.com/google/mathfu.

[3] Texas instrument digital signal processing (dsp) library for msp430 microcontrollers. https://www.ti.com/tool/MSP-DSPLIB.

[4] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

[5] Maaz Bin Safeer Ahmad and Alvin Cheung. Automatically leveraging MapReduce frameworks for data-intensive applications. *SIGMOD*, 2018.

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

[7] Maaz Bin Safeer Ahmad, Alexander J Root, Andrew Adams, Shoaib Kamil, and Alvin Cheung. Vector instruction selection for digital signal processors using program synthesis. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1004–1016, 2022.

[8] Jordi Armengol-Estapé, Rodrigo CO Rocha, Jackson Woodruff, Pasquale Minervini, and Michael FP O'Boyle. Forklift: An extensible neural lifter. *Conference on Language Modeling*, 2024.

[9] Jordi Armengol-Estapé, Jackson Woodruff, Chris Cummins, and Michael F.P. O'Boyle. SLaDe: A portable small language model decompiler for optimized assembler. *CGO*, 2024.

[10] Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. cvc5: A versatile and industrial-strength SMT solver. In *TACAS (1)*, volume 13243 of *Lecture Notes in Computer Science*, pages 415–442. Springer, 2022.

[11] Sahil Bhatia, Sumer Kohli, Sanjit A Seshia, and Alvin Cheung. Building code transpilers for domain-specific languages using program synthesis. *ECOOP*, 2023.

[12] Sahil Bhatia, Jie Qiu, Sanjit A Seshia, and Alvin Cheung. Can llms perform verified lifting of code? *Technical Report*, 2024.

[13] Alexander Brauckmann, Luc Jaulmes, José W de Souza Magalhães, Elizabeth Polgreen, and Michael FP O'Boyle. Tensorize: Fast synthesis of tensor programs from legacy code using symbolic tracing, sketching and solving. In *ACM/IEEE CGO*, 2025.

[14] Alexander Brauckmann, Elizabeth Polgreen, Tobias Grosser, and Michael FP O'Boyle. mlirsynth: Automatic, retargetable program raising in multi-level ir using program synthesis. In *2023 32nd International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 39–50. IEEE, 2023.

[15] Ying Cao, Ruigang Liang, Kai Chen, and Peiwei Hu. Boosting neural networks to decompile optimized binaries. In *Proceedings of the 38th Annual Computer Security Applications Conference*, pages 508–518, 2022.

[16] Lorenzo Chelini, Andi Drebes, Oleksandr Zinenko, Albert Cohen, Nicolas Vasilache, Tobias Grosser, and Henk Corporaal. Progressive raising in multi-level ir. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 15–26. IEEE, 2021.

[17] Xinyun Chen, Chang Liu, and Dawn Song. Tree-to-tree neural networks for program translation. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, page 2552–2562, Red Hook, NY, USA, 2018. Curran Associates Inc.

[18] Zimin Chen, Steve Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. Sequencer: Sequence-to-sequence learning for end-to-end program repair. *IEEE Transactions on Software Engineering*, 47(9):1943–1959, 2019.

[19] Bruce Collie, Philip Ginsbach, Jackson Woodruff, Ajitha Rajan, and Michael FP O'Boyle. M3: Semantic api migrations. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pages 90–102, 2020.

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

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

[22] Adhitha Dias, Kirshanthan Sundararajah, Charitha Saumya, and Milind Kulkarni. SparseLNR: Accelerating sparse tensor computations using loop nest restructuring. *ICS*, 2022.

[23] Mehdi Drissi, Olivia Watkins, Aditya Khant, Vivaswat Ojha, Pedro Sandoval, Rakia Segev, Eric Weiner, and Robert Keller. Program language translation using a grammar-driven tree-to-tree model. *ICML*, 2018.

[24] Albert Einstein et al. The foundation of the general theory of relativity. *Annalen Phys*, 49(7):769–822, 1916.

[25] John K. Feser, Isil Dillig, and Armando Solar-Lezama. Inductive program synthesis guided by observational program similarity. *Proc. ACM Program. Lang.*, 7(OOPSLA2):912–940, 2023.

[26] Cheng Fu, Huili Chen, Haolan Liu, Xinyun Cheng, and Yuandong Tian. Coda: An end-to-end neural program decompiler. *NeurIPs*, 2019.

[27] Philip Ginsbach, Toomas Remmelg, Michel Steuwer, Bruno Bodin, Christophe Dubach, and Michael FP O'Boyle. 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*, pages 139–153, 2018.

[28] Tobias Grosser, Armin Groesslinger, and Christian Lengauer. Polly—performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters*, 22(04):1250010, 2012.

[29] Sagar Imambi, Kolla Bhanu Prakash, and GR Kanagachidambaresan. Pytorch. *Programming with TensorFlow: Solution for Edge Computing Applications*, pages 87–104, 2021.

[30] JAX. JAX: High performance array computing. Accessed 2024. Available at https://jax.readthedocs.io/en/latest/index.html.

[31] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. Oracle-guided component-based program synthesis. In *ICSE (1)*, pages 215–224. ACM, 2010.

[32] Shoaib Kamil, Alvin Cheung, Shachar Itzhaky, and Armando Solar-Lezama. Verified lifting of stencil computations. *PLDI*, 2016.

[33] D. S. Katz, J. Ruchti, and E. Schulte. Using recurrent neural networks for decompilation. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 346–356, 2018. https://doi.org/10.1109/SANER.2018.8330222 doi:10.1109/SANER.2018.8330222.

[34] Omer Katz, Yuval Olshaker, Yoav Goldberg, and Eran Yahav. Towards neural decompilation. *CoRR*, 2019.

[35] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017. https://arxiv.org/abs/1412.6980 arXiv:1412.6980.

[36] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. The tensor algebra compiler. *OOPSLA*, 2017.

[37] Julien Klaus, Mark Blacher, Joachim Giesen, Paul Gerhardt Rump, and Konstantin Wiedom. Compiling linear algebra expressions into efficient code. *ICCS*, 2022.

[38] Daniel Kroening and Michael Tautsching. CBMC — C bounded model checker. *TACAS 2014*, (8413):389–391, 2014.

[39] Taku Kudo and John Richardson. Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. *arXiv preprint arXiv:1808.06226*, 2018.

[40] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. Mlir: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 2–14. IEEE, 2021.

[41] Celine Lee, Abdulrahman Mahmoud, Michal Kurek, Simone Campanoni, David Brooks, Stephen Chong, Gu-Yeon Wei, and Alexander M Rush. Guess & sketch: Language model guided transpilation. *arXiv preprint arXiv:2309.14396*, 2023.

[42] Woosuk Lee, Kihong Heo, Rajeev Alur, and Mayur Naik. Accelerating search-based program synthesis using learned probabilistic models. In *PLDI*, pages 436–449. ACM, 2018.

[43] V. I. Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady*, 10:707, February 1966.

[44] José Wesley de Souza Magalhães, Jackson Woodruff, Elizabeth Polgreen, and Michael F.P. O'Boyle. C2TACO: Lifting tensor code to TACO. *GPCE*, 2023.

[45] Charith Mendis, Jeffrey Bosboom, Kevin Wu, Shoaib Kamil, Jonathan Ragan-Kelley, Sylvain Paris, Qin Zhao, and Saman Amarasinghe. Helium: lifting high-performance stencil kernels

from stripped x86 binaries to halide DSL code. ACM Press, 6 2015. URL: http://dx.doi.org/10.1145/2737924.2737974, https://doi.org/10.1145/2737924.2737974 doi:10.1145/2737924.2737974.

[46] Gonzalo Navarro. A guided tour to approximate string matching. *ACM Comput. Surv.*, 33(1):31–88, mar 2001. https://doi.org/10.1145/375360.375365 doi:10.1145/375360.375365.

[47] Yuto Nishida, Sahil Bhatia, Shahdaj Laddad, Hasan Genc, Yakun Sophia Shao, and Alvin Cheung. Code transpilation for hardware accelerators. *CoRR*, 2023. Available at https://arxiv.org/pdf/2308.06410.pdf.

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

[49] Rangeet Pan, Ali Reza Ibrahimzada, Rahul Krishna, Divya Sankar, Lambert Pougeum Wassi, Michele Merler, Boris Sobolev, Raju Pavuluri, Saurabh Sinha, and Reyhaneh Jabbarvand. Lost in translation: A study of bugs introduced by large language models while translating code. *ICSE*, 2024.

[50] Phitchaya Mangpo Phothilimthana, Aditya Thakur, Rastislav Bodik, and Dinakar Dhurjati. Scaling up superoptimization. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 297–310, 2016.

[51] Jie Qiu, Colin Cai, Sahil Bhatia, Niranjan Hasabnis, Sanjit A Seshia, and Alvin Cheung. Tenspiler: A verified lifting-based compiler for tensor operations. In *38th European Conference on Object-Oriented Programming (ECOOP 2024)*, 2024.

[52] Jie Qiu, Colin Cai, Sahil Bhatia, Niranjan Hasabnis, Sanjit A. Seshia, and Alvin Cheung. Tenspiler: A Verified-Lifting-Based Compiler for Tensor Operations (Artifact). *Dagstuhl Artifacts Series*, 10(2), 2024. URL: https://drops.dagstuhl.de/entities/document/10.4230/DARTS.10.2.17, https://doi.org/10.4230/DARTS.10.2.17 doi:10.4230/DARTS.10.2.17.

[53] Saurabh Raje, Yufan Xu, Atanas Rountev, Edward F Valeev, and Saday Sadayappan. CoNST: Code generator for sparse tensor networks. *CoRR*, 2024. Available at https://arxiv.org/html/2401.04836v1.

[54] Vikas Raunak, Arul Menezes, and Marcin Junczys-Dowmunt. The curious case of hallucinations in neural machine translation, 2021. https://arxiv.org/abs/2104.06683 arXiv:2104.06683.

[55] Joseph Redmon. Darknet: Open source neural networks in c. http://pjreddie.com/darknet/, 2013–2016.

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

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

[58] Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. *arXiv preprint arXiv:1508.07909*, 2015.

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

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

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

[62] Adilla Susungi, Norman A Rink, Albert Cohen, Jeronimo Castrillon, and Claude Tadonki. Meta-programming for cross-domain tensor optimizations. *GPCE*, 2018.

[63] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'14, page 3104–3112, Cambridge, MA, USA, 2014. MIT Press.

[64] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[65] Jackson Woodruff, Jordi Armengol-Estapé, Sam Ainsworth, and Michael FP O'Boyle. Bind the gap: Compiling real software to hardware fft accelerators. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 687–702, 2022.

[66] Rohan Yadav, Alex Aiken, and Fredrik Kjolstad. DISTAL: The distributed tensor algebra compiler. *PLDI*, 2022.

[67] Tian Zhao, Alexander Rucker, and Kunle Olukotun. Sigma: Compiling einstein summations to locatlity-aware dataflow. *ASPLOS*, 2023.

[68] Vojin Zivojnovic. Dspstone: A dsp-oriented benchmarking methodology. *Proc. Signal Processing Applications & Technology, Dallas, TX, 1994*, pages 715–720, 1994.