Accelerating Sparse Algebra with Program Synthesis

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

Shideh Hashemian
University of Edinburgh
Edinburgh, United Kingdom
shideh.hashemian@ed.ac.uk

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

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

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

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

Abstract

Linear algebra libraries and tensor domain-specific languages are able to deliver high performance for modern scientific and machine learning workloads. While there has been recent work in automatically translating legacy software to use these libraries/DSLs using pattern matching and program lifting, this has been largely limited to dense linear algebra.

This paper tackles the more challenging problem of porting legacy *sparse* linear algebra code to libraries and DSLs. It exploits the power of large language models to predict a sketch of the solution and then uses type-based program synthesis to search the space of possible code to target parameter bindings. We implement this in a tool named SLEB (Sparse LiftEr with Binding) and evaluate it on a large set of benchmarks and real-world datasets, comparing against two state-of-the-art compiler techniques, LiLAC and SpEQ; and GPT 4.o. Overall, we lift 94% of programs compared to 11%, 17%, and 48% for LiLAC, SpEQ, and GPT 4.o respectively. This delivers a geomean speedup of 2.6x and 7.2x on a CPU and GPU platform, respectively.

CCS Concepts: • **Software and its engineering → Retargetable compilers**.

Keywords: Sparse, Program Lifting, Synthesis, TACO, Tensor Algebra

ACM Reference Format:
José Wesley de Souza Magalhães, Shideh Hashemian, Alexander Brauckmann, Jackson Woodruff, Elizabeth Polgreen, and Michael F. P. O'Boyle. 2026. Accelerating Sparse Algebra with Program Synthesis. In *Proceedings of the 35th ACM SIGPLAN International Conference on Compiler Construction (CC '26), January 31 – February 1, 2026, Sydney, NSW, Australia.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3771775.3786281

CC '26, Sydney, NSW, Australia

ACM ISBN 979-8-4007-2274-5/2026/01
https://doi.org/10.1145/3771775.3786281

1 Introduction

Matrix-based linear algebra has been a key component of scientific applications for decades. The recent increase in machine learning workloads has seen significant interest in higher-dimensional linear algebra based on tensor contractions [21]. This, in turn, has led to the development of a number of accelerator libraries and high-level programming languages, which exploit domain knowledge in order to generate highly optimized code [18, 43, 46, 47].

Although libraries and DSLs can deliver potentially significant performance, existing programs, however, have to be manually rewritten and ported to them. This manual effort remains a major barrier to their wider adoption. This issue has led to significant interest in automatically porting legacy code via a variety of techniques, including matching for matrix libraries [17] and lifting to tensor DSLs [38, 39].

Matching and Lifting. Matching based schemes search the source program for code patterns that correspond to the meaning of a specific API or library. The pattern is frequently described as a code fragment at the compiler AST or IR level [17, 24], though other approaches, including program classification and input-output behavior, are also studied [45]. The meaning of the library is normally provided as an unoptimized code implementation.

In contrast, lifting approaches aim to express low-level source code in a higher-level DSL [37, 38]. They frequently use program synthesis to search a target space of programs and show equivalence between the source program and the candidate lifted [36]. Searching the exponential program space efficiently and proving equivalence are the key challenges of these schemes.

1.1 Sparse Tensor Algebra

While there has been much work in matching and lifting legacy dense linear algebra code, there has been, however, less progress in tackling sparse code. The reason is that analyzing legacy sparse computation is difficult [22]. Unlike dense operations. There are many different data storage formats employed [44] and idiosyncratic ways for programmers

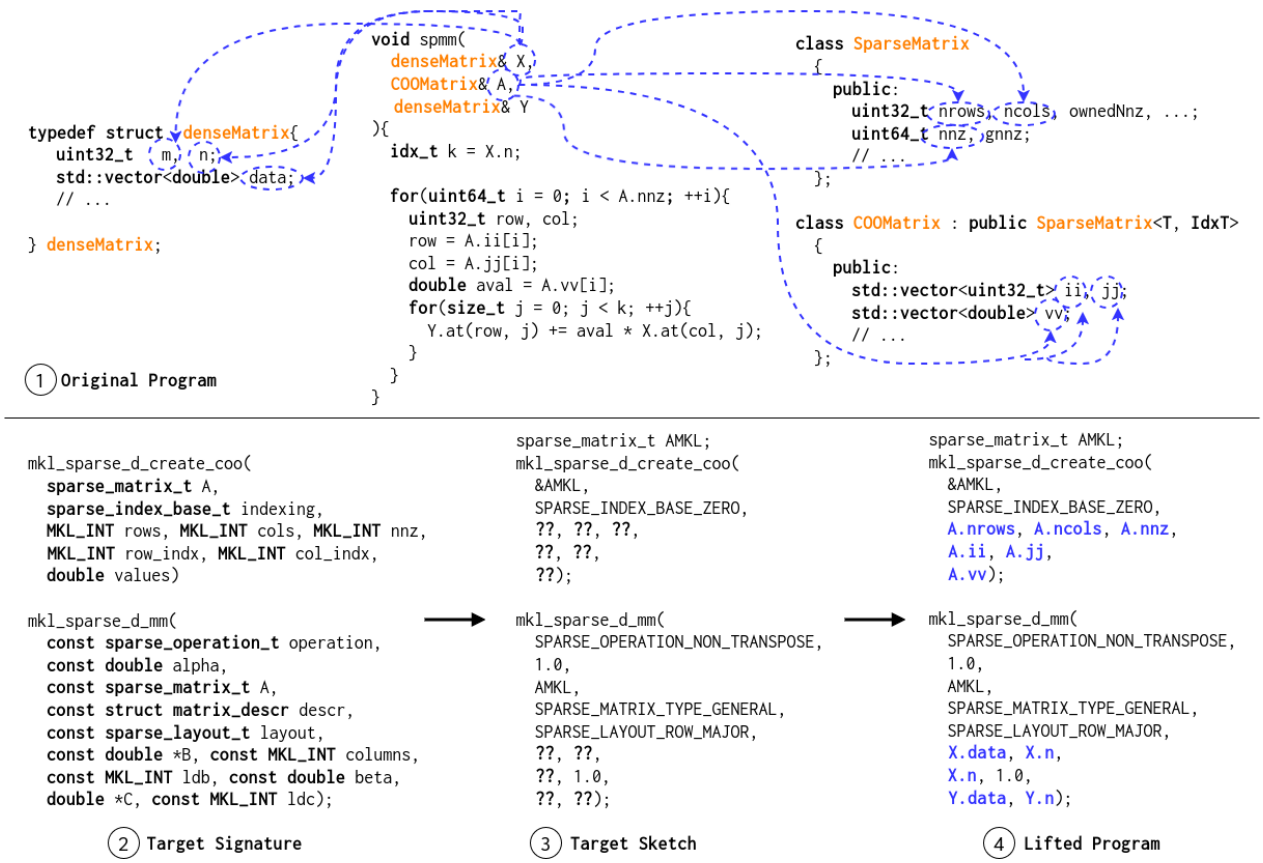

Figure 1. Example of sparse code lifting. The original program spmm ① is replaced by the lifted program ④ which consists of two calls to the MKL library with parameters originating as variables in the original program. The signature for these calls is shown in ② and partially completed with known values ③. SLEB determines that the unknown values or holes ?? correspond to the variables encircled in dotted blue lines from the original program. Orange text shows the type chain for relevant variables.

to code sparse linear algebra. Furthermore, while tensor DSLs are able to express and support a wide range of higher-order algebraic operations [7], they are still under development and in some cases may be outperformed by sparse matrix libraries. Thus, the best lifting target for an application is in flux. The complexity of source code and a moving target mean that the automatic porting of sparse code remains an open problem.

Prior Work. There has been some limited work in the area, focusing solely on matching sparse matrix-vector computation to library APIs. In [23], constraint-based LLVM IR pattern matching is used to detect code and replace it with optimized calls to an accelerator library. In [28], a data dependence graph is used as a pattern and stuttered bisimulation is employed to show equivalence. LiLAC assumes just one data format (CSR) and SpEQ supports only CSR and CSC, and neither is able to manage general sparse matrix operations or any higher-order tensor contractions. Each pattern to match has to be provided by the user plus a reference implementation. Furthermore, they are brittle in the presence of idiosyncratic code and only tackle fixed library APIs rather than open-ended DSLs. Ideally, we would like an approach that can handle general sparse computation with arbitrary data formats and target both libraries and DSLs, choosing which is ever the most appropriate.

1.2 Our Approach

This paper develops a new technique to lift sparse legacy code to both libraries and DSLs using a large language model (LLM) and program synthesis. It avoids the problem of hallucination by asking the LLM for a high-level classification of data format and tensor operation rather than using it for code generation. Given the classification, we generate an API or DSL program sketch with gaps or holes that source code variables must fill. Smart type-based binding is then used to match source code variables to library or DSL parameters, dramatically reducing the synthesis search space.

Automatic input-output testing is then used to validate the translation. We implement this methodology in a tool named SLEB, which is publicly available.[1]

We perform an extensive evaluation of our approach on 31 benchmarks and 14 real-world data sets selected from a variety of external sources. We consider 3 different targets: two APIs (Intel MKL [2] for CPU, cuSPARSE [1] for GPU); and a DSL (TACO [26]) for high-order tensor algebra. We compare our scheme against two different competitive compiler approaches, LiLAC [25] and SpEQ [28], and a well-known language model, GPT 4.o [6]. We evaluate each scheme in terms of the number of programs lifted, correctness, and the speedup obtained when executing the lifted programs on CPU and GPU. We lift 93% of the benchmarks, are always correct, and deliver a geomean speedup of 2.6x and 7.2x on CPU and GPU platforms, respectively.

2 Motivation/Background

Our objective is to automatically synthesize sparse algebra programs in a target high-level API or domain-specific language (DSL), given a source legacy implementation. A sparse implementation is characterized by two elements: (i) the operation performed, such as sparse matrix multiplication or sparse tensor addition, and (ii) the storage format of sparse data, such as compressed sparse row (CSR). SLEB determines both elements and generates correctly synthesized code.

2.1 Example

To illustrate our approach, Figure 1 presents a running example of SLEB. The original source code in ① is sparse matrix–matrix multiplication (SpMM) from the SpComm3D framework [5], implemented in C++. SLEB correctly classifies this code as performing SpMM with data stored in coordinate list (COO) format. Based on this prediction, SLEB identifies the computation as an operation supported by the Intel Math Kernel Library (MKL). Using the MKL library requires first declaring a data format and then calling the appropriate sparse matrix operation.

SLEB targets the MKL functions whose signatures are shown in ②. The first function `mkl_sparse_d_create_coo` creates a sparse matrix in COO format storing double-precision floating points. The second function `mkl_sparse_d_mm` performs sparse dense matrix-matrix multiplication with the same datatypes.

The COO format declaration is a function with 7 parameters, while the actual operation is performed by a function with 11 parameters. These function signatures are used to form a sketch where SLEB fills in known parameters such as `desc = SPARSE_MATRIX_TYPE_GENERAL` and leaves 11 unknown parameters as holes denoted by ?? in ③.

[1] https://github.com/JWesleySM/sleb

2.1.1 Binding.

Determining the correct parameters or bindings is non-trivial. Every variable in the source code is a potential candidate, which is made more complex in the presence of hierarchical type declarations. In ①, each of the parameters passed to the original spmm function refers to classes and structs declared elsewhere in the program, which in turn refer to standard std C++ library components.

Given that there are 29 variables in the source code and 11 parameters in the function calls, naive enumeration of all options would be combinatorially expensive. Instead, SLEB uses type analysis and smart synthesis to reduce the number of candidates from 1.2×10^{16} to 8 data sketch and 95 operation sketch candidates.

The relevant source code variables are those highlighted by blue dotted lines. We then wish to fill in the holes and generate the correct code shown in ④. SLEB is an automatic technique to replace ① with ④ which delivers a 5.2x speedup improvement on a CPU and 11.6x on a GPU over the original code.

.

3 Overview

Figure 2 provides a high level overview of SLEB. It combines an LLMs program analysis and smart synthesis to generate API or DSL code L from an input program P.

Classification. The first stage reads the source program in and asks an LLM using a prompt to classify the source code into a number of potential sparse operator classes; and a number of potential different data formats. For those programs involving tensors, we also prompt for the number of arguments and the dimensionality of the tensors.

From this information we create two types of sketch: data storage and sparse operation. Both APIs and DSL require data format declarations, so the data storage sketch is common to both. For APIs, the operation sketch is a function call with holes in as illustrated in Figure 1. For DSLs, the operation sketch is a smaller grammar expressing the space of possible matching programs.

Analysis. This second stage determines which source code variables are potential candidates for later binding. The type based analysis recursively examines all structured types until it reaches base types: ints, floats. Doubles and arrays of int, floats and doubles. These are all candidates for the next stage data binding.

Data Binding. In the third stage candidate variables are iteratively bound to the data sketch generated during classification and tested for validity. All successful candidates are passed on to both the operation binding used for APIs and to the IO testing stage.

Operator Binding. The fourth stage is either *operator binding* for API calls or *exploration* for DSLs. In operator

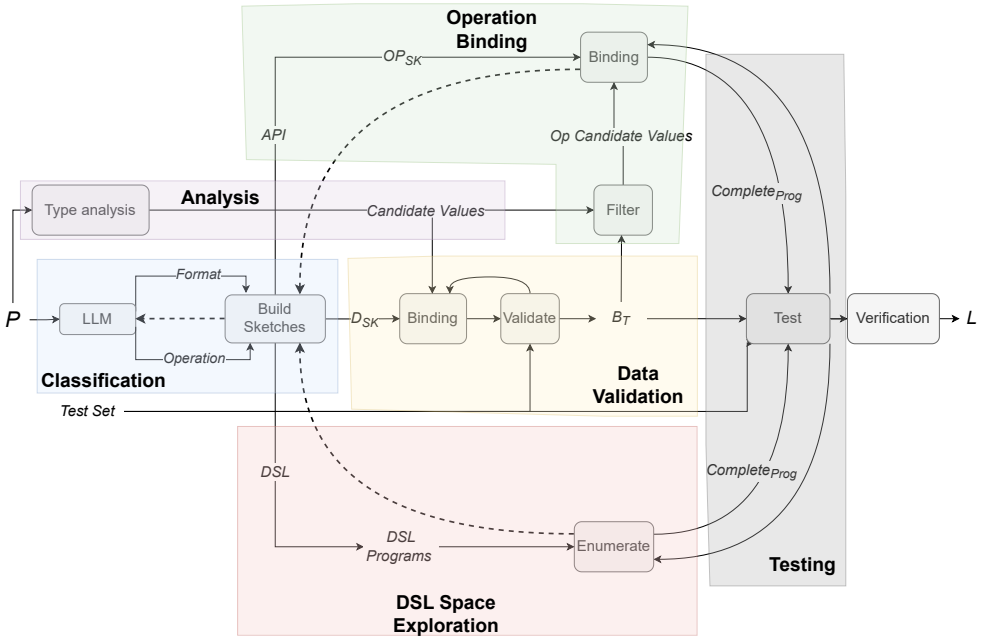

Figure 2. The overall pipeline of SLEB. A program P is classified by operator and data format from which a data and operator sketches are generated. Program analysis is performed on P to determine candidate variables t match the holes in the generated API or DSL sketch. Data binding finds the correct variables to complete the data sketch while operator binding completes the API operator call. When the program is lifted to a DSL rather than an API, as small grammar is explored to find the correct program. All of this is validated by IO testing before being available for subsequent verification

binding, all candidate variables that were bound during data binding are removed from consideration for operator binding. The intuition behind this is that during data binding, sparse variables are identified while operator binding is concerned with dense types. Operation binding searches for candidates to fit the operator sketch generated by classification. It checks the validity of the binding by executing the code on test examples in *testing*. If these all fail, the next highest probability operator class is chosen for exploration and the process is repeated.

DSL Exploration. For programs lifted to a DSL, the fourth stage consists of *exploration*. Here, the grammar describing the operator sketches supplied by the classification stage is enumerated using bottom-up enumerative synthesis, guided by predicted length and tensor dimension. These are then evaluated and tested as in operator binding.

Testing. Once the final stage has confirmed that the lifted program passes all IO tests it can be passed onto an external verification tool such as [20] completing the process.

4 Technique

Given a fragment of low-level tensor manipulating code P, which we assume to be manipulating sparse tensors, the aim is to automatically lift P to an equivalent expression in the form of either high-performance library calls or expressions in a domain-specific language for manipulating tensors, like TACO. We break this down into two separate subtasks: first, we identify the format of the sparse tensors used in the original code; second, we identify the semantics of the source code and choose a semantically equivalent high-performance library call or, if no such library call exists, the equivalent expressions in a high-performance DSL.

An overview of our approach is shown in Algorithm 1. Lines 2-3 correspond to the classification and program analysis stages, From the LLM response, we build the data sketch (line 4). The data binding phase, lines 5 to 7, validates that bindings generated by the `bind` are in accordance with section 4.3. The `bind` function is described in detail in Algorithm 2. In case the operation is supported by APIs, we proceed to operation binding (lines 9-11). The loop at line 12 corresponds to the final testing. In case we need to target a DSL, we proceed to the exploration phase followed by testing, as shown at lines 17-21.

4.1 Classification

The first step is to identify the storage format and the semantic behavior of the input source code. We frame this task as a classification problem and query an LLM to provide *labels*

Algorithm 1: SLEB lifting algorithm. The subprocedures *analyze* and *filter* are described in Sections 4.2 and 4.4 respectively.

```
input : source computation kernel P, set of tests φ
output: lifted MKL/TACO program, or no solution
Algorithm lift(P, φ)
    format, Op, N, orders ← queryLLM(P);
    candidates ← analyze(P);
    D_SK ← sketch(format);
    DB ← bind(D_SK, candidates);
    for b ∈ DB do
        if validate(b, P, φ) then
            if Op is MKL supported then
                Op_SK ← sketch(Op);
                candidates_Op ← filter(b, candidates);
                OPB ← bind(Op_SK, candidates_Op);
                for op_b ∈ OPB do
                    L ← compile(b, op_b);
                    if test(L, φ) then
                        return L
            else if Op is TACO supported then
                E ← genProgSpace(Op, N, orders);
                for e ∈ E do
                    L ← compile(b, e);
                    if test(L, φ) then
                        return L
    return no solution
```

Algorithm 2: Binding generation process for a given sketch SK.

```
Procedure genBindings(i, H, V, B, current, Seen)
    if i = |H| then
        b ← {h ↦ v | (v, h) ∈ current};
        if b ∉ Seen then
            B ← B ∪ {b};
            Seen ← Seen ∪ {b};
        return
    h ← H[i];
    for v ∈ V do
        if ⟨h, v⟩ type-checks and obeys all rules then
            current ← current ∪ ⟨h, v⟩;
            genBindings(i + 1, H, V, Bcurrent, Seen) ;
            current ← current \ ⟨h, v⟩;
Procedure bind(SK, 𝒱, Seen)
    ℋ ← holes(SK);
    B ← ∅;
    genBindings(0, ℋ, 𝒱, B, ∅, Seen);
    return B;
```

for the input program. We use GPT-4.o [6] as the LLM, giving it the prompt below, followed by the source code of P.

"The following code contains a sparse linear/tensor algebra operation. You have two tasks: first, tell me what is the sparse storage format being used, second, tell me what operation is being performed. For the format, choose among the following: [CSR, CSC, COO, JDS]. For the operator, be extremely brief. For example, if the code is computing a sparse matrix-vector product, say just SpMV, if it is tensor-times-vector, say SpTTV, and so on. If the operator takes as sparse input(s) a high-order tensor (>2), tell me also the number of tensors in the operation together with their respective orders."

From the LLM answer, we extract the *data label* and the *operator label*. In case it is a tensor operation, we also query for the number of tensors in P and their respective orders. We support the following data formats: compressed sparse column (CSC), compressed sparse row (CSR), coordinate list (COO), and jagged diagonal storage (JDS). For operators, we support the main sparse computations, from sparse-dense and sparse-sparse matrix products to tensor element-wise scaling and contraction operations.

Given a data label, we create a sketch to represent the API call that creates the guessed format. We term this D_{sk}, or the data sketch. We then analyze the operator label to detect whether the sparse operation is supported by our target APIs. If that is the case, we construct an operation sketch Op_{sk}, i.e., a high-level API call with holes for all the inputs; otherwise, we use the number of inputs and orders to build a set of short expressions in the TACO DSL and enumerate this set, testing each expression.

4.2 Program Analysis

We perform static analysis on the abstract syntax tree (AST) of the input program P to identify candidate values that may be bound to the holes in the sketches built in the classification stage. This analysis yields a set of source-level variables that can be used to instantiate the sketches.

Since both D_{sk} and Op_{sk} are statically typed, we can filter candidates for binding based on their type in the source code. We restrict ourselves to variables of integer types and double-precision floating points, as these commonly represent matrix or tensor dimensions, coordinates, and numerical values. We also consider pointers to these types as valid binding candidates.

In case a variable V has a type T which is composite (structs, classes, etc) or derived (references, type redefinitions, etc), we recursively traverse the type definition tree rooted at T, inspecting subnodes and collecting the values

that may type-check with sketch holes. For example, in Figure 1 ①, variable `X` cannot be directly bound to a sketch hole. We can, however, determine that `X.m`, `X.n`, and `X.data` type-check with sketch holes and keep those as candidates.

We perform the analysis in 3 stages. First, we scan the input arguments of P. If the lifting fails, the next iteration will also consider the variables defined in the body of P as candidates. Finally, we extend the analysis to consider the variables in the body of functions called in P. Because sparse kernels often iterate over entire data structures in loops, we also treat loop-bound variables as potential binding candidates.

4.3 Data Binding

A key step to making lifting scalable is the data binding process. During this phase, we complete D_{sk} by binding the set of candidate values produced by program analysis to the inputs of the API call in the data sketch.

A data sketch D_{sk} is defined as

$$D_{sk} = \langle n_{rows}, n_{cols}, NNZ, rows_{index}, cols_{index}, values \rangle$$

where:

- $\boldsymbol{n_{rows}}$ and $\boldsymbol{n_{cols}}$ are integers denoting the number of rows and columns of the sparse data structure. For higher-order tensors, D_{sk} has a hole $\boldsymbol{n}$ for each dimension.
- $\boldsymbol{NNZ}$ is an integer representing the number of non-zero entries.
- $\boldsymbol{rows_{index}}$ and $\boldsymbol{cols_{index}}$ are integer arrays storing the coordinates of non-zero entries. Again, there is one array per dimension for high-order tensors.
- $\boldsymbol{values}$ is a double-precision floating point array storing the non-zero entries of the sparse data structure.

We perform binding using type-constrained enumeration described in Algorithm 2.At each recursive step, it selects the next hole and attempts to bind it to each candidate value that satisfies the data validation constraints. When all the holes are bounded SLEB stores the binding, checking against a global set to prevent duplicates. This procedure returns all the bindings to be validated with format-specific constraints.

We enumerate all possible combinations of variable bindings and API call inputs, subject to the following constraints:

Type compatibility. Only combinations of variables whose types match the expected input types of the API call are considered.

Unique pointer binding. A pointer variable may be bound to at most one placeholder ("hole") in the sketch.

Naming constraint. If a hole in the sketch is bound to a field accessed through a pointer (e.g., `A->c`), all other values bound within the same sketch must either refer to the same object (`A`) or to standalone variables. References to fields of different objects (e.g., `B->d`) are not considered as candidates.

4.3.1 Data Validation.

Given the sparse format F and a binding $B\langle H, V\rangle$, we validate B by loading the corresponding values for V from the test set and verifying that they satisfy the storage-format constraints for F. These constraints represent the semantics of each format and are used to discard invalid bindings.

CSR (Compressed Sparse Row):

- NNZ is implicitly defined as the last entry of $rows_{index}$.
- $rows_{index}$ must have either length $n_{\text{rows}} + 1$ or there are two rows index holes of length n_{rows} storing the start and the end of each line.
- $cols_{index}$ and $values$ must have length equal to NNZ.
- $rows_{index}$ must be monotonically non-decreasing, with the first value being 0.

CSC (Compressed Sparse Column):

- NNZ is implicitly defined as the last entry of $cols_{index}$.
- $cols_{index}$ must have either length $n_{\text{cols}} + 1$ or there are two rows index holes of length n_{cols} storing the start and the end of each line.
- $rows_{index}$ and $values$ must have length equal to NNZ.
- $cols_{index}$ must be monotonically non-decreasing, with the first value being 0.

COO (Coordinate List):

- Arrays $rows_{index}$, $cols_{index}$, and $values$ must have length equals to NNZ.
- Indices in $rows_{index}$ and $cols_{index}$ must satisfy $0 \leq rows_{index}[i] < n_{rows}$ and $0 \leq cols_{index}[i] < n_{cols}$.

JDS (Jagged Diagonal Storage):

- The last value of the diagonal pointer array gives NNZ, which need not be explicitly stored elsewhere.

We iteratively validate each completed D_{sk} until a valid binding B_T is found. We then proceed to complete the operation sketch or enumerate TACO programs using $D_{sk}(B_T)$ as the data creation API call.

4.4 Operation Binding

We complete OP_{sk} again using type-constrained enumeration. We restrict the bindings so that variables may only be bound to the data *or* the operation sketch, not both. SLEB has a filtering component that takes as input the set of values produced by the analysis step and removes values $v \in B_T.V$. This filtering produces a much smaller set of candidates for filling holes in OP_{sk}, making the enumeration scalable.

Parameter Inversion. Operation sketches have parameters, i.e., holes that can only assume a value $v \in 0, 1$. For the MKL operation sketches, those parameters are `alpha` and `beta`, which can take values 0.0 or 1.0; and booleans indicating whether the sparse input is transposed and the storage layout of dense matrix inputs. Since there are only two possibilities for each parameter, we do not attempt to

bind variables for those holes, and instead simply enumerate all the options.

4.5 DSL Exploration

Operations involving high-order sparse tensors are not well-supported in sparse APIs. Therefore, we resort to domain-specific languages to lift those types of programs. Unlike an API call, DSLs have a less restricted nature. The same operation might have multiple different equivalent TACO expressions. For example, a tensor-times-vector operation can be expressed in many ways depending on which dimension is contracted. Therefore, SLEB does not create a single operation sketch and instead enumerates TACO programs.

We build a search space of TACO expressions parametrized by the number of tensors and their orders as predicted by the LLM. We then enumerate all the expressions in that space using the data created by D_{sk} and B_T. For each expression, we invoke the TACO compiler to generate C code and run it on the test set until we match the original program's output or exhaust the program space.

4.6 Testing

For each completed operation sketch, we test the combination of the completed D_{sk} and completed Op_{sk} against 10 test cases. In each test, the dense input values are randomly generated, while the sparse inputs (matrices or tensors) are drawn from real-world datasets. Each candidate instantiation is executed and compared against the output of the original program. If output matches the original, we return the completed sketches as a valid solution.

If all the possible completions of Op_{sk} failed for a given D_{sk}, we return to enumerating other possible completions for D_{sk}. When SLEB tries all possible completions of D_{sk} and none of them pass the tests, it randomly chooses a new data format and repeats the lifting process.

5 Experimental Setup

5.1 Benchmarks

We gathered a suite of 31 programs extracted from diverse benchmark suites, applications, and software libraries, none of which were developed by the authors. Our suite contains implementations of 15 different sparse algebra operations: 14 sparse matrix-vector product (**SpMV**) taken from CSparse [16], DOLFINx [10], GinkGO [9], Netlib [19], NAS Parallel Benchmarks [32], Parboil [42], QuantLib [3], SciMark [4], and TACO-generated from SpEQ artifact [27]; 3 sparse matrix-matrix multiplication (**SpMM**) from SuperLU [30], Sextans [41], and SpComm3D [5]; 2 sparse general matrix-matrix multiplication (**SpGEMM**) from CSeg [8] and also from GinkGO [9]; 1 sparse matrix addition (**SpMADD**) from CSparse [16]; 1 sampled-dense-dense matrix multiplication (**SDDMM**) from SpComm3D [5]; 4 tensor element-wise (**TEW**) (addition, subtraction, multiplication and division), 2 tensor-scalar addition (**TSA**) and multiplication (**TSM**), and tensor-times vector (**SpTTV**), and tensor-times matrix (**SpTTM**) and Matricized Tensor Times Khatri-Rao Product (**MTTKRP**) from the PASTA [29] benchmark suite; and another MTTKRP from Splatt [40]. These are implemented in 3 different languages (C, C++, and Fortran77) and use 4 different sparse storage formats (CSC, CSR, COO, JDS).

Table 1. Real-world matrices and higher-order tensors used for performance experiments

Dataset	Dimensions	#NNZ	Density
bcsstk17	$10.9K \times 10.9K$	428,650	4×10^{-3}
pdb1HYS	$36K \times 36K$	4,344,765	3×10^{-3}
rma10	$46K \times 46K$	2,329,092	1×10^{-3}
cant	$62K \times 62K$	4,007,383	1×10^{-3}
consph	$83K \times 83K$	6,010,480	9×10^{-4}
cop20k	$121K \times 121K$	2,624,331	2×10^{-4}
shipsec1	$140K \times 140K$	3,568,176	2×10^{-4}
scircuit	$171K \times 171K$	958,936	3×10^{-5}
mac-econ	$206K \times 206K$	1,273,389	9×10^{-5}
pwtk	$217.9K \times 217.9K$	11,524,432	2×10^{-4}
webbase-1M	$1M \times 1M$	3,105,536	3×10^{-6}
Facebook	$1591 \times 63K \times 63K$	737,934	1×10^{-7}
NELL-2	$12K \times 9K \times 28K$	76,879,419	2×10^{-5}
NELL-1	$2.9M \times 2.1M \times 25.5M$	143,599,552	9×10^{-13}

5.2 Alternative Approaches

We compared against three alternative approaches LiLAC [23]: a pattern matching approach that uses constraints over LLVM IR to detect SpMV and replace with calls to MKL and cuSPARSE; SpEQ [28]: this uses a data dependence graph and rewrite system to detect sparse storage format and computation, respectively replacing with calls to MKL or cuSparse; GPT4.o [6], a popular LLM, which is provided with the original program and prompted to provide equivalent library or TACO code.

5.3 Platform

SLEB is implemented in Python version 3.10 and clang/LLVM version 18.0. We target Intel MKL version 2025 1.16, cuSPARSE version 12, and TACO version 0.1. The original benchmarks are compiled with `gcc/g++/mpifort` version 11.4. The operating system is Ubuntu 22.04.5 LTS.

We evaluate on a 64-core AMD Ryzen Threadripper 7970X CPU with 125 GB of RAM (DDR5RAM). The programs lifted to GPU are executed on an NVIDIA GeForce GTX 1080 Ti using driver 550.163.01 and CUDA runtime version 12.4.

5.4 Methodology

We give a timeout of 10 minutes to each technique to lift a benchmark. We evaluate LiLAC and SpEQ with their respective artifacts [22] [27]. GPT.4o was repeatedly tested with different prompts to find the best recall. To ensure a fair evaluation, when evaluating the performance, we use

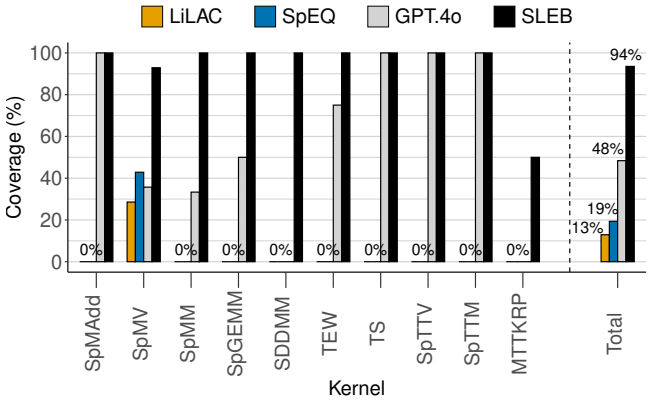

Figure 3. Coverage by different kernels.

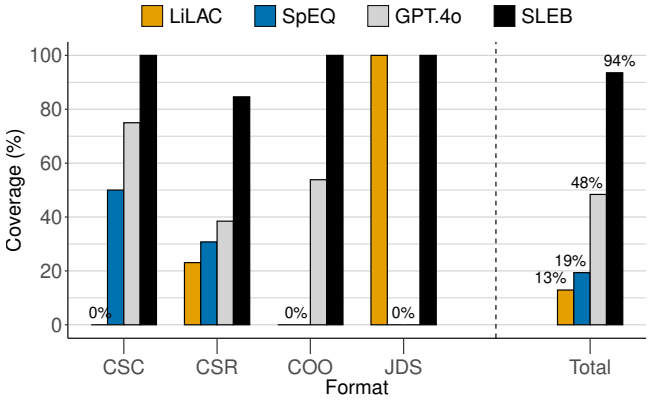

Figure 4. Coverage by different sparse storage formats.

gcc and the targeted backends, MKL, cuSPARSE, and TACO (for each approach), and unless otherwise stated, report the best performance achieved. We run each benchmark version (original and lifted) 10 times and report the average. We reported speedup as the ratio of the lifted 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, with the non-lifted programs assigned a speedup of 1 by definition. For the performance experiments, we use as inputs the same real-world sparse datasets used by the original TACO paper [26] in their evaluation. Said datasets are described in Table 1.

6 Results

6.1 Coverage

We evaluate the success rate of each technique in two dimensionalities: coverage by sparse operation and sparse storage format.

6.1.1 Coverage by Kernel.

Figure 3 shows the percentage of programs successfully lifted by each technique grouped by operation. LiLAC has the lowest coverage accross the benchmark suite, lifting only 13% of the benchmarks. LiLAC is only able to synthesize one type of operation, SpMVs, but even for that operation, it only lifts 28% of the benchmarks. SpEQ has a slight higher overall coverage lifting 19% of the benchmarks, but as well as LiLAC, it only lifts SpMVs. This result shows the brittleness of those techniques, which can only detect very specific patterns in source code and miss more complicated implementation styles and complex operations. Moreover, none of these two approaches can lift higher-dimensional tensor code.

GPT.4o has strong recognition ability, which enables it to lift benchmarks that implement distinct operations and achieve overall coverage of 48%. Nevertheless, it still struggles to synthesize the correct code for complex implementation styles. For the matrix operations, it lifts the SpMADD benchmark, 5 of the SpMV, and only 1 of SpMM and SpGEMM. It is unable to lift the SDDMM benchmark. We observe that GPT.4o is successful for the cases where the benchmark use standard algorithms and simple data types, e.g., when the sparse objects are represented with simple pointers to scalar types. However, GPT.4o fails to correctly lift benchmarks that contain optimizations and data structures which are more complex, e.g., user-defined structs/classes, specialized templates and containers from the standard library in C++. For the tensor benchmarks. GPT.4o reaches high coverage value for simpler operations such as tensor-scalar operations, but its coverage decreases for non-trivial programs such as MTTKRP.

SLEB is by far the technique with highest coverage, correctly lifting 94% of the benchmarks. Furthermore, it is the only approach able to lift all the different sparse operations, achieving 100% of coverage in all categories except for SpMV and MTTKRP. SLEB fails to lift the SpMV from GinkGO [9] because the benchmark assumes the input matrix has all the non-zero values as the same constant. For MTTKRP, SLEB cannot lift the kernel from Splatt [40]. This happens because while actual MTTKRP computation matricizes the input tensor, this benchmark uses an already matricized version of the actual tensor in the computation, which makes the equivalent TACO program much complex to synthesize.

6.1.2 Coverage by Format.

Figure 4 depicts the coverage of each technique in terms of sparse storage format. LiLAC can lift only 20% of CSR the JDS benchmark. SpEQ is able to lift only CSR and CSC benchmarks. GPT.4o can synthesizes benchmarks in CSC, CSR, and COO. However, its coverage is lower than SLEB in all cases. SLEB outperforms the alternative approaches and is the only method able to lift benchmarks that implement all different storage formats. The only occasion it fails to lift all the category is for the CSR based benchmark MTTKRP as described above.

Table 2. Lifting time results. The ᵀ symbol means the benchmark operates on the transpose of the sparse input.

Benchmark	Data	Data Discarded	Op	Time(s)	Benchmark	Data	Data Discarded	Op	Time(s)
CSparse SpMAdd	40	79	1036	34.4	SuperLU SpMM	1	838	1	329.14
CSparse SpMV CSC	1	3	1	1.24	Sextans SpMM	1	25	1	9.72
CSparse SpMV CSR	1	1	3	0.45	CSeg SpGEMM	2	3	307	2.7
CSparse SpMV CSRᵀ	1	2	0	0.82	SpComm3D SpMM	8	1055	95	479.06
CSparse SpMV COO	6	16	13	6.09	SpComm3D SDDMM	210	419	1	99.62
CSparse SpMV COOᵀ	1	1	5	0.4	PASTA TEW Add	13	73	87877	64.63
Dolfinx SpMV	1	3	98	1.64	PASTA TEW Div	1	1	45099	20.96
Netlib SpMV	1	1	2	0.42	PASTA TEW Mul	6	31	2009	8.18
Netlib SpMVᵀ	1	2	1	0.78	PASTA TEW Sub	39	229	73607	109.17
NPB CG SpMV	1	1	17	0.41	PASTA TSA	4	4	29	1.32
Parboil SpMV	1	1	31	0.44	PASTA TSM	1	1	121	0.27
Quantlib SpMV	1	1	25	2.3	PASTA SpTTV	7	21	350	4.47
SciMark SpMV	1	3	1	1.14	PASTA SpTTM	10	30	462	7.7
TACO SpMV	1	3	2	1.18	PASTA MTTKRP	1	3	9680	11.83
GinkGO SpGEMM	71	141	14322	60.95					

6.2 Lifting Time

Table 2 summarizes the number of data and operation bindings that SLEB considers per benchmark and the time in seconds it takes to correctly determine a correct binding. The number of data bindings considered ranges from 1 to 210 with up to 1055 discarded due to static format incompatibility. Data binding, however, does not require any program execution and thus lifting time is relatively invariant of it as can be seen in Figure 5. Operation binding does require program execution and is closely correlated to lifting time as can also be seen in the figure. For simple benchmarks such as CSparse SpMV CSC, Netlib SpMVᵀ and SciMark SpMV, only one execution is needed, however for more complex tensor benchmarks such as PASTA TEW Add, more than 87k are required. Table 2 also reports the number of bindings discarded with data validation. Due to the format constraints, SLEB is able to always discard more data bindings than what it needs to consider during the operation phase. This result how data validation is efficient to reduce the search space during operation phase and make lifting scalable.

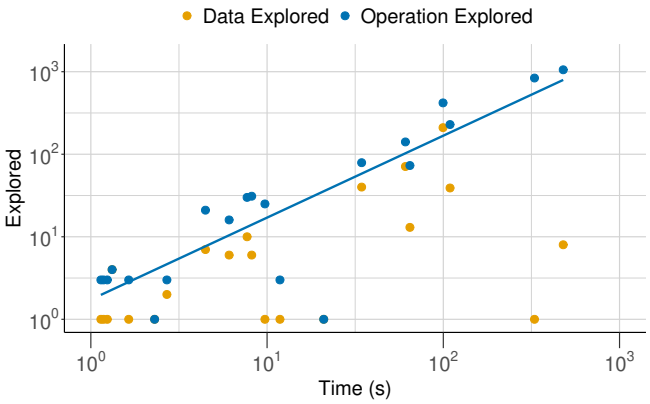

Figure 5. Relation between candidates explored and lifting time.

6.3 Speedup

Speedup on CPU. Figure 6 shows the speedup achieved by each method on the CPU for each benchmark together with a geometric mean. We lift the SpMV, SpMM, and SpGEMM benchmarks to MKL, totaling 18. The remaining 11 benchmarks are lifted to TACO, which includes sparse matrix addition, SDDMM and the high-order tensor benchmarks.

The highest speedup values come from PASTA SpTTV and GinkGO SPGEMM with speedup of 12x and 17x respectively. Programs lifted to MKL achieve good speedup for CSR and CSC, but not COO.

For tensor programs, TACO overall generates efficient code for 75% of TEW benchmarks and for kernels where the output is dense or semi-sparse. We also observe good performance improvement on SpTTV benchmarks, where TACO schedules optimized the computation when the outptut is stored in CSR. We do not achieve speedup for tensor-scalar as the original TSA and TSM benchmarks from PASTA [29] only scale the non-zero values from the tensor and assume a particular tensor structure, whereas TACO-generated code is general and also unable to parallelize those two benchmarks. The structure assumption is also the reason why TEW division cannot be accelerated.

SLEB provides the highest speedup of 2.6x against 1.7x by GPT.4o and 1.2x by LiLAC and SpEQ. This is due to the strong lifting capabilities of SLEB, which is able to achieve great coverage and synthesize benchmarks in which speedup

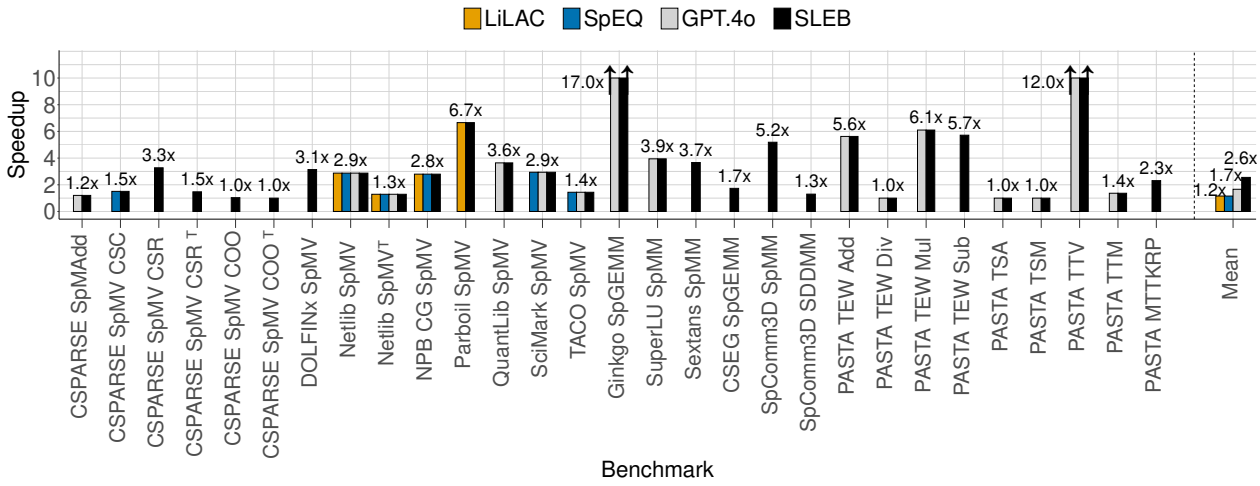

Figure 6. Overall speedup of benchmarks lifted to CPU. X-axis lists the benchmarks. Y-axis shows the average speedup over the baseline.

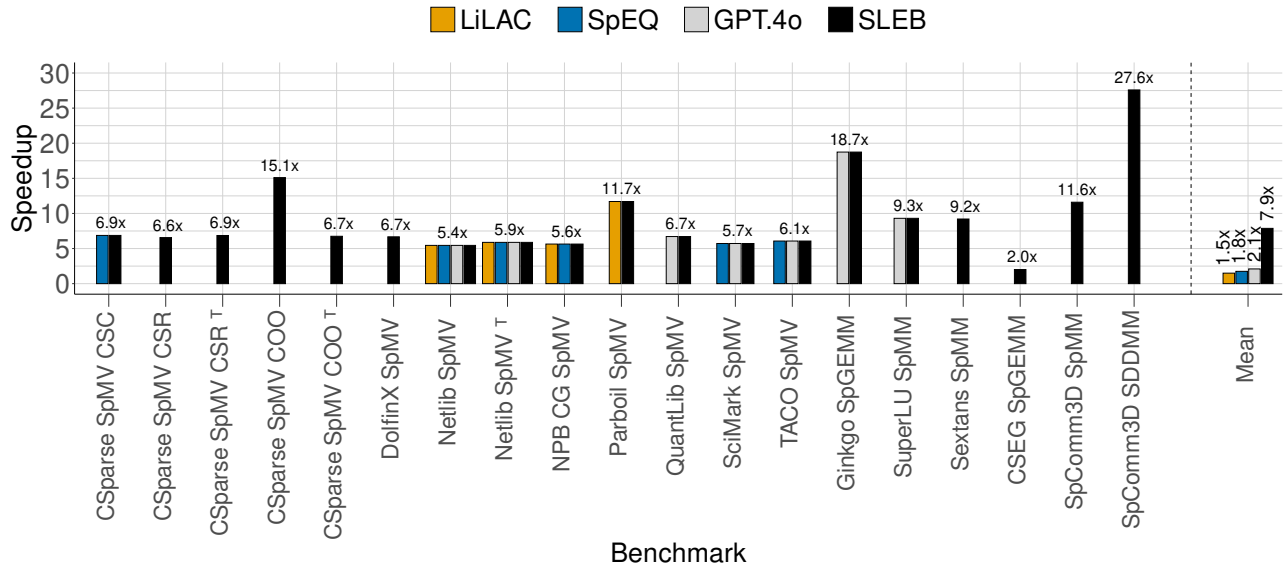

Figure 7. Overall speedup of benchmarks lifted to GPU. X-axis lists the benchmarks. Y-axis shows the average speedup over the baseline.

improvements are higher, such as tensor contraction ones and matrix-matrix kernels.

Speedup on GPU. We are not aware of a target that supports high-order sparse tensor algebra for GPUs. TACO, for instance, is unstable to generate CUDA code when the output is also sparse. We therefore restrict GPU evaluation for the benchmarks that take sparse matrices as inputs and evaluate the speedup improvements on a GPU platform lifting the programs to cuSPARSE. Figure 7 depicts the results. Speedup gains are always higher in the GPU than the CPU. The lowest improvement happens in the SpGEMM from CSeg [8], which is twice faster than the original implementation. Other than that, the lifted cuSPARSE programs are at least 5x faster. The highest speedup is 27.6x on the SDDMM benchmark. As well as for CPU, SLEB is the approach that delivers the highest geomean speedup of 7.8x against 2x, 1.7x and 1.5x from GPT.4o, SpEQ, and LiLAC respectively.

Speedup by Input. To evaluate how the performance improvements vary with the input, we measure the average achieved by the benchmarks on each input. Figure 8 illustrate those results. For the CPU matrix benchmarks, the speedup ranges from 2x on `bcsstk17` and `webbase-1M` to

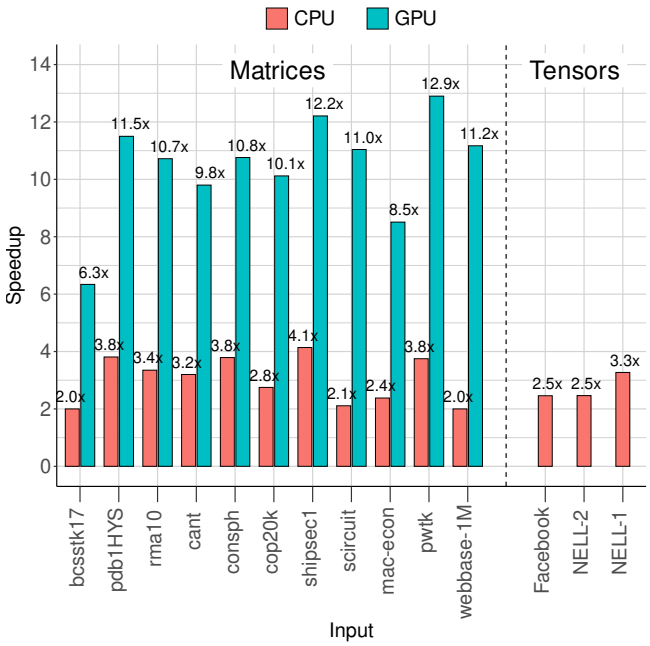

Figure 8. Overall speedup of benchmarks lifted to CPU and GPU. X-axis show different inputs sorted by size. Y-axis shows the average speedup over the baseline.

4.1x on `shipsec1`. For the GPU, the speedup varies from 6.3x `bcsstk17` to 12.9x on `pwtk`, which is the matrices with the largest number of non-zero entries. In the case of the tensor benchmarks, there is a clear trend showing that the speedup achieved by lifting tends to grow with the sparsity. Lifted programs run 2.5x faster on `facebook` and `NELL-2` and 3.3x faster on `NELL-1`.

7 Related Work

API Matching. There has been prior work in trying to detect program structures for a variety of tasks, including accelerating linear algebra kernels [24]. Here, a constraint language is used to search over a normalized IR to detect acceleratable regions that match an API. Automatically, determining the constraints that describes an API is explored in [13–15]. However, such an approach is often sensitive to code structure [24]. KernelFaRer [17] is a more robust approach, but only focuses on one pattern, namely dense matrix multiplication, and is also susceptible to code structure. A different approach is used in FACC [45]. This uses input-output equivalence of code sections and is considerably more robust than pattern-based techniques. However, it only tackles fast Fourier transformation acceleration with narrow API function signatures. ATC [35] is a similar approach, this time targeting dense GEMMs like KernelFaRer. It is extremely robust, yet again limited to just one operator.

Sparse API Matching. There is considerably less research in matching sparse code to accelerator libraries due to the complexity of the task. LiLAC [22] is a development of [24] tackling sparse vector multiplication (SpMv). It uses a vanilla implementation of the SpMv to generate constraints that are used to search the legacy code. More recently, SpEQ [28] uses a dependency graph of an SpMv implementation to identify the sparse data format used and egraph-based normalisation to recognize sections that can be replaced with API to a sparse matrix vector libraries. Both of these techniques are only evaluated on SpMv and cannot tackle tensors or lifting to DSLs.

Tensor Lifters. There has been much recent work in lifting dense legacy code to high-level tensor DSLs. A popular approach is to use bottom-up enumerative synthesis to generate potential candidates, which are checked using user IO testing. Examples include TF-coder [39] C2TACO [34] and mlirSynth [12], which use type information, compiler analysis and redundancy checking to narrow the search. Bottom-up enumerative search has difficulty scaling to large program size and recently [11] has been developed, which employs a top-down sketching scheme.

An alternative, Tenspiler, is based on detecting program invariants [38]. It uses a symbolic synthesizer that generates the target program and an invariant that proves the program is correct. Although powerful, it requires external definition of the syntax and semantics of their target language in their internal intermediate language. More recent approaches uses a language model to guess a (set) of candidate(s) and search for the correct solution [31, 33]. All of these schemes are limited by the fact that they can only work on dense linear algebra and are unable to tackle the more challenging task of sparse lifting to a sparse tensor DSL.

8 Conclusion

This paper presents SLEB which tackles the challenging problem of porting legacy sparse linear algebra code. SLEB use an LLM to predict a sketch of the solution and then uses program analysis and type based synthesis to dramatically reduce the search space of possible code to target parameter bindings. When evaluated on a large set of benchmarks and real world data sets it outperforms two state-of-the-art compiler schemes and LLM.

Future work will explore tackling different sparse DSLs. While LLMs are powerful, they are expensive and future work will look at replacing it with a more task-specific trained transformer model.

Acknowledgments

José Wesley de Souza Magalhães is partly sponsored by Huawei Research. We thank the reviewers for their insightful comments.

References

[1] [n. d.]. cuSPARSE [n. d.]. Basic Linear Algebra for Sparse Matrices on NVIDIA GPUs. https://developer.nvidia.com/cusparse.

[2] [n. d.]. Intel® oneAPI Math Kernel Librar. https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl.html.

[3] [n. d.]. QuantLib: a free/open-source library for quantitative finance. https://www.quantlib.org/.

[4] [n. d.]. Scimark 2.0. https://math.nist.gov/scimark2/.

[5] Nabil Abubaker and Torsten Hoefler. 2024. SpComm3D: A Framework for Enabling Sparse Communication in 3D Sparse Kernels. arXiv:2404.19638 [cs.DC]

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

[7] Willow Ahrens, Teodoro Fields Collin, Radha Patel, Kyle Deeds, Changwan Hong, and Saman Amarasinghe. 2025. Finch: Sparse and Structured Tensor Programming with Control Flow. *Proc. ACM Program. Lang.* 9, OOPSLA1, Article 117 (April 2025), 31 pages. doi:10.1145/3720473

[8] Xiaojing An and Ümit V. Çatalyürek. 2021. Column-Segmented Sparse Matrix-Matrix Multiplication. In *HiPC21: 28th IEEE International Conference on High Performance Computing, Data, & Analytics*.

[9] Hartwig Anzt, Terry Cojean, Goran Flegar, Fritz Göbel, Thomas Grützmacher, Pratik Nayak, Tobias Ribizel, Yuhsiang Mike Tsai, and Enrique S. Quintana-Ortí. 2022. Ginkgo: A Modern Linear Operator Algebra Framework for High Performance Computing. *ACM Trans. Math. Software* 48, 1 (Feb. 2022), 2:1–2:33. doi:10.1145/3480935

[10] Igor A Baratta, Joseph P Dean, Jørgen S Dokken, Michal Habera, Jack HALE, Chris N Richardson, Marie E Rognes, Matthew W Scroggs, Nathan Sime, and Garth N Wells. 2023. DOLFINx: the next generation FEniCS problem solving environment. (2023).

[11] Alexander Brauckmann, Luc Jaulmes, José W de Souza Magalhães, Elizabeth Polgreen, and Michael FP O'Boyle. 2025. Tensorize: Fast Synthesis of Tensor Programs from Legacy Code using Symbolic Tracing, Sketching and Solving. In *ACM/IEEE CGO*.

[12] Alexander Brauckmann, Elizabeth Polgreen, Tobias Grosser, and Michael FP O'Boyle. 2023. mlirSynth: Automatic, Retargetable Program Raising in Multi-Level IR using Program Synthesis. In *2023 32nd International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 39–50.

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

[14] 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.

[15] 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.

[16] Timothy A Davis. 2006. *Direct methods for sparse linear systems*. SIAM.

[17] 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.

[18] Adhitha Dias, Kirshanthan Sundararajah, Charitha Saumya, and Milind Kulkarni. 2022. SparseLNR: Accelerating Sparse Tensor Computations Using Loop Nest Restructuring. *ICS* (2022).

[19] Jack Dongarra, Victor Eijkhout, and Henk van der Vorst. 2001. An iterative solver benchmark. *Scientific Programming* 9, 4 (2001), 223–231.

[20] Tristan Dyer, Alper Altuntas, and John Baugh. 2019. Bounded Verification of Sparse Matrix Computations. In *2019 IEEE/ACM 3rd International Workshop on Software Correctness for HPC Applications (Correctness)*. 36–43. doi:10.1109/Correctness49594.2019.00010

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

[22] Philip Ginsbach. 2020. https://github.com/ginsbach/llvm/tree/linearalgebra, https://github.com/ginsbach/clang/tree/research.

[23] Philip Ginsbach, Bruce Collie, and Michael FP O'Boyle. 2020. Automatically harnessing sparse acceleration. In *Proceedings of the 29th International Conference on Compiler Construction*. 179–190.

[24] 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.

[25] 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.

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

[27] Avery Laird. 2024. SpEQ: Translation of Sparse Codes using Equivalences. https://zenodo.org/records/10906216.

[28] Avery Laird, Bangtian Liu, Nikolaj Bjørner, and Maryam Mehri Dehnavi. 2024. SpEQ: Translation of Sparse Codes using Equivalences. *Proc. ACM Program. Lang.* 8, PLDI, Article 215 (June 2024), 24 pages. doi:10.1145/3656445

[29] Jiajia Li, Yuchen Ma, Xiaolong Wu, Ang Li, and Kevin Barker. 2019. PASTA: a parallel sparse tensor algorithm benchmark suite. *CCF Transactions on High Performance Computing* 1, 2 (2019), 111–130.

[30] Xiaoye Sherry Li, James Demmel, John Gilbert, Laura Grigori, and Meiyue Shao. 2011. *SuperLU*. Springer US, Boston, MA, 1955–1962. doi:10.1007/978-0-387-09766-4_95

[31] Yixuan Li, José Wesley de Souza Magalhães, Alexander Brauckmann, Michael F. P. O'Boyle, and Elizabeth Polgreen. 2025. Guided Tensor Lifting. *Proc. ACM Program. Lang.* 9, PLDI, Article 227 (June 2025), 23 pages. doi:10.1145/3729330

[32] Júnior Löff, Dalvan Griebler, Gabriele Mencagli, Gabriell Araujo, Massimo Torquati, Marco Danelutto, and Luiz Gustavo Fernandes. 2021. The NAS parallel benchmarks for evaluating C++ parallel programming frameworks on shared-memory architectures. *Future Generation Computer Systems* 125 (2021), 743–757.

[33] José Wesley de Souza Magalhães, Jackson Woodruff, Jordi Armengol-Estapé, Alexander Brauckmann, Luc Jaulmes, Elizabeth Polgreen, and Michael FP O'Boyle. 2025. Guess, Measure & Edit: Using Lowering to Lift Tensor Code. In *34th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE.

[34] José Wesley de Souza Magalhães, Jackson Woodruff, Elizabeth Polgreen, and Michael F.P. O'Boyle. 2023. C2TACO: Lifting Tensor Code to TACO. *GPCE* (2023).

[35] 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.

[36] Charith Mendis, Jeffrey Bosboom, Kevin Wu, Shoaib Kamil, Jonathan Ragan-Kelley, Sylvain Paris, Qin Zhao, and Saman Amarasinghe. 2015. Helium: lifting high-performance stencil kernels from stripped x86 binaries to halide DSL code. *PLDI*. doi:10.1145/2737924.2737974

[37] Yuto Nishida, Sahil Bhatia, Shahdaj Laddad, Hasan Genc, Yakun Sophia Shao, and Alvin Cheung. 2023. Code Transpilation for Hardware Accelerators. *CoRR* (2023). Available at https://arxiv.org/pdf/2308.06410.pdf.

[38] Jie Qiu, Colin Cai, Sahil Bhatia, Niranjan Hasabnis, Sanjit A Seshia, and Alvin Cheung. 2024. Tenspiler: A Verified Lifting-Based Compiler for Tensor Operations. In *38th European Conference on Object-Oriented Programming (ECOOP 2024)*.

[39] 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.

[40] Shaden Smith and George Karypis. 2016. SPLATT: The Surprisingly ParalleL spArse Tensor Toolkit. http://cs.umn.edu/~splatt/.

[41] Linghao Song, Yuze Chi, Atefeh Sohrabizadeh, Young kyu Choi, Jason Lau, and Jason Cong. 2022. Sextans: A Streaming Accelerator for General-Purpose Sparse-Matrix Dense-Matrix Multiplication. In *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 65–77.

[42] John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen-mei W Hwu. 2012. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing* 127, 7.2 (2012).

[43] Adilla Susungi, Norman A Rink, Albert Cohen, Jeronimo Castrillon, and Claude Tadonki. 2018. Meta-programming for Cross-Domain Tensor Optimizations. *GPCE* (2018).

[44] Jaeyeon Won, Charith Mendis, Joel S Emer, and Saman Amarasinghe. 2023. WACO: learning workload-aware co-optimization of the format and schedule of a sparse tensor program. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 920–934.

[45] Jackson Woodruff, Jordi Armengol-Estapé, Sam Ainsworth, and Michael FP O'Boyle. 2022. 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*. 687–702.

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

[47] Tian Zhao, Alexander Rucker, and Kunle Olukotun. 2023. Sigma: Compiling Einstein Summations to Locatlity-Aware Dataflow. *ASPLOS* (2023).

Received 2025-11-12; accepted 2025-12-10