THE UNIVERSITY *of* EDINBURGH

Program Lifting for Acceleration

José Wesley De Souza Magalhães

Doctor of Philosophy

Institute of Computing Systems Architecture
School of Informatics
University of Edinburgh

2026

Abstract

The fast evolution of computer architectures brings the promise of increased performance. To leverage these advances and achieve high-performance, legacy code must be ported to new hardware. However, emerging hardware is increasingly complex and specialized, making the task of porting code highly challenging. Compilers must have a solid knowledge of the target architecture to generate efficient code, and developing such compilers for each new device is prohibitively costly in a scenario of rapid and constant change.

New hardware is often programmed by specific interfaces or higher-level domain-specific languages (DSLs). DSLs embed knowledge about the application's domain which is crucial for optimization. Hence, automatic translation or lifting of existing programs to hardware-oriented languages can bridge the gap between legacy implementations and unseen architectures and improve code portability.

This thesis presents new solutions to automatically porting existing code to new architectures with Program Lifting: the translation of general-purpose code to higher-level application programming interfaces (APIs) or DSLs. It presents lifting techniques in the context of dense and sparse linear/tensor algebra, the fundamental blocks of many modern workloads, such as data science and machine learning. This thesis shows that program lifting facilitates portability of code, enabling enormous performance gains on specialized hardware.

Lay Summary

Current computers increasingly rely on specialized hardware to deliver high performance. However, this performance comes at the cost of complexity. Specialized architectures are typically difficult to program, and most existing software cannot easily leverage these new devices. This is largely because such software was implemented for older systems, and must be rewritten to run on newer hardware, a process that is time-consuming, expensive, and error-prone. This is particularity complex in a scenario where computer architecture evolves quickly.

Modern hardware is usually accessed through special programming interfaces or domain-specific languages (DSLs), which are designed to efficiently express certain types of computations. Therefore, translating existing code to those interfaces and DSLs can improve automatic software migration.

This thesis proposes novel techniques to bridge the gap between existing code and emerging hardware using a technique called *Program Lifting*. This technique automatically translates existing programs into a equivalent and more efficient version in a higher-level programming interfaces or specialized languages. This thesis focus on programs that perform dense and sparse linear/tensor algebra, which form the backbone of several modern applications, such as machine learning. The results show that program lifting can translate code without requiring extensive manual rewriting, which allows existing software to be ported to modern specialized hardware and achieve significant performance improvements.

Acknowledgements

It was fun, it was intense, it was crazy. A whirlwind of emotions. The Ph.D. was undoubtedly one of the most remarkable experiences of my life. I was not alone in it, so I would like to express all my gratitude here. First of all, I thank God for always being with me, blessing me and giving me strength in all the difficult moments. It is always comforting to know that You are with me.

I would like to thank my supervisor, Michael O'Boyle, for welcoming me as one of his students and for being the best Ph.D. supervisor one could ever have. Thank you very much for all the advice, guidance, and support. Your expertise ensured that my research never went off track, but above all, your human side proved to be admirable and essential to me. It is an honor to have been a student of such a legendary figure.

I would also like to thank the other professors who provided valuable insights for my research. Thank you, Elizabeth Polgreen, for being an excellent co-supervisor and for teaching me everything and much more about program synthesis. Thank you, Jackson Woodruff, for helping me from the very beginning, when we were still Ph.D. colleagues. Thank you, Björn Franke, for taking part in my annual reviews and always providing valuable feedback. Finally, thank you, Amir Shaikhha, for the great ideas shared during our conversations. Also, many thanks to the members of IGS who provided support and assistance throughout the entire period.

I would like to give a special thank you to all the co-authors and friends I made in Edinburgh. Thank you, Alex, Amir, Celeste, Esra, Jordi, Mahesh, Matt, Rodrigo, Shideh, Tianyi, Yixuan, and everyone else. Arriving in a new country is never easy, and you made my time in Edinburgh really special and enjoyable. It would never have been the same without you. A big, heartful thank you to all of you.

A special thank you to my family. My mother Josefa, my brothers Thales and João Paulo, and my love Juliana. Being away from you for so long was the greatest challenge of my life, but I knew and I could always count on you even from afar. Thank you a million times. I love you all!

Finally, to all those who helped me in some way and I did not name here, I feel grateful for your cooperation.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified. Some of the material used in this thesis has been published in the following papers:

1. **José Wesley de Souza Magalhães**, Jackson Woodruff, Elizabeth Polgreen, Michael F.P. O'Boyle, *C2TACO: Lifting Tensor Code to TACO*. 22nd ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE), 2023. **(Best Paper Award)**

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

3. **José Wesley de Souza Magalhães**, Shideh Hashemian, Alexander Brauckmann, Jackson Woodruff, Elizabeth Polgreen, Michael F.P. O'Boyle, *Accelerating Sparse Algebra with Program Synthesis*. 35th International Conference on Compiler Construction (CC), 2026.

During my Ph.D., I have also co-authored the following papers, which are not part of this thesis:

1. Jordi Armengol-Estapé, Jackson Woodruff, Alexander Brauckmann, **José Wesley de Souza Magalhães**, Michael F.P. O'Boyle, *ExeBench: an ML-scale dataset of executable C functions*. 6th ACM SIGPLAN International Symposium on Machine Programming (MAPS), 2022

2. Alexander Brauckmann, Luc Jaulmes, **José Wesley de Souza Magalhães**, Elizabeth Polgreen, Michael F. P. O'Boyle. *Tensorize: Fast Synthesis of Tensor Programs from Legacy Code using Symbolic Tracing, Sketching and Solving*. 23rd ACM/IEEE International Symposium on Code Generation and Optimization (CGO), 2025.

3. Yixuan Li, **José Wesley de Souza Magalhães**, Alexander Brauckmann, Michael F. P. O'Boyle, Elizabeth Polgreen. *Guided Tensor Lifting*. 46th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 2025.

4. Alexander Brauckmann, Aarsh Chaube, **José Wesley de Souza Magalhães**, Elizabeth Polgreen, Michael F. P. O'Boyle. *Tensor Program Superoptimization through Cost-Guided Symbolic Program Synthesis*. 24th ACM/IEEE International Symposium on Code Generation and Optimization (CGO), 2026.

(*José Wesley De Souza Magalhães*)

Contents

1 Introduction **1**

1.1 The Specialized Hardware Scenario 2
1.2 The Code Migration Challenge 3
1.2.1 Existing Approaches 3
1.3 Thesis 4
1.4 Contributions 5
1.5 Outline 6
1.6 Summary 7

2 Technical Background **9**

2.1 Program Synthesis 9
2.1.1 Synthesis Specifications 10
2.1.2 Synthesis Algorithms 17
2.2 Neural Machine Translation 25
2.2.1 Sequence-to-Sequence Modeling 26
2.2.2 Tokenization 28
2.2.3 Language Models 29
2.2.4 Classification 30
2.3 Verification for Lifting 32
2.3.1 Bounded Model Checking 32
2.3.2 Post-conditions and Loop Invariants 33
2.3.3 Stuttering Simulation 34
2.4 Linear and Tensor Algebra 35
2.4.1 Dense Algebra 35
2.4.2 Sparse Algebra 35
2.4.3 Einsum Notation 37
2.4.4 Applications in Computer Science 37

2.5 Summary 38

3 Related Work **39**

3.1 Rule-based Translation 40
3.1.1 Transpilation 40
3.1.2 Lifting 43
3.1.3 API Replacement 45
3.2 Program Synthesis 47
3.2.1 IO-based Synthesis 47
3.2.2 Transpilation 51
3.2.3 Lifting 52
3.2.4 API Replacement 56
3.3 Neural Machine Translation 58
3.3.1 Transpilation 58
3.3.2 Lifting 60
3.3.3 API Replacement 62
3.4 Tensor Algebra Compilers 63
3.5 Summary 65

4 Lifting Tensor Code with Guided Enumerative Synthesis **67**

4.1 Overview 68
4.1.1 TACO 68
4.1.2 Example 69
4.2 C2TACO Architecture 70
4.2.1 Classification 71
4.2.2 IO Generation 71
4.2.3 Lifting via Synthesis 71
4.2.4 Validity 72
4.2.5 Exporting 72
4.3 Enumerative Template Synthesis 72
4.3.1 The Grammar 73
4.3.2 Specification 74
4.3.3 Template Enumeration 74
4.3.4 Instantiating Templates 75
4.4 Synthesis Guided by Static Code Analysis 75
4.4.1 TACO Program Length 77

4.4.2 Tensor Dimensions . . . 78
4.4.3 Operator Analysis . . . 79
4.5 Experimental Setup . . . 79
4.5.1 Environment . . . 79
4.5.2 Competitive Approaches . . . 80
4.5.3 Methodology . . . 82
4.6 Evaluation . . . 82
4.6.1 Coverage . . . 83
4.6.2 Error Analysis . . . 85
4.6.3 Generation Time . . . 87
4.6.4 Performance of Lifted Code . . . 89
4.6.5 Summary . . . 91
4.7 Conclusions . . . 92

5 Guess, Measure & Edit: Using Lowering to Lift 93
5.1 Guess, Measure & Edit . . . 94
5.2 Applying Guess, Measure & Edit to Tensor Code . . . 97
5.2.1 Language Model: Guess . . . 99
5.2.2 Program Similarity: Lower and Measure . . . 102
5.2.3 Edit . . . 104
5.2.4 Check (Testing and Verification) . . . 108
5.3 Experimental Setup . . . 109
5.3.1 Environment . . . 109
5.3.2 Competitive Approaches . . . 110
5.3.3 Methodology . . . 111
5.4 Evaluation . . . 112
5.4.1 Coverage . . . 112
5.4.2 Program Space Exploration . . . 114
5.4.3 Detailed Evaluation of Lifting Time . . . 119
5.4.4 Verification . . . 119
5.4.5 Speedup . . . 121
5.5 Conclusions . . . 122

6 Sparse Lifting with Neuro-Guided Sketch Synthesis 123
6.1 Sparse Lifting . . . 123
6.1.1 Example . . . 124

6.2 SLEB Overview . . . 127
6.2.1 Classification . . . 127
6.2.2 Analysis . . . 127
6.2.3 Data Binding . . . 127
6.2.4 Operator Binding . . . 128
6.2.5 DSL Exploration . . . 128
6.2.6 Testing . . . 128
6.3 Implementation . . . 129
6.3.1 Classification . . . 129
6.3.2 Program Analysis . . . 131
6.3.3 Data Binding . . . 132
6.3.4 Operation Binding . . . 135
6.3.5 DSL Exploration . . . 136
6.3.6 Testing . . . 137
6.4 Experimental Setup . . . 137
6.4.1 Environment . . . 137
6.4.2 Competitive Approaches . . . 141
6.4.3 Methodology . . . 141
6.5 Evaluation . . . 142
6.5.1 Coverage . . . 142
6.5.2 Lifting Time . . . 145
6.5.3 Speedup . . . 147
6.6 Conclusions . . . 150

7 Conclusion **151**
7.1 Contributions Summary . . . 151
7.1.1 C2TACO: Lifting with Guided Enumerative Synthesis . . . 151
7.1.2 Guess, Measure & Edit: a Methodology to Lift code Using Compiler Lowering . . . 152
7.1.3 SLEB: Lifting Sparse Code with Neuro-Guided Sketch Synthesis . . . 152
7.2 Critical Evaluation . . . 153
7.2.1 Hard-Wired Heuristics . . . 153
7.2.2 Correctness . . . 153
7.2.3 Benchmarks . . . 154

7.3 Future Work 154
7.3.1 Expansion to Other Domains 155
7.3.2 Generalization 155
7.3.3 Targeted Architectures 155
7.3.4 Usage of Language Models 155
7.4 Summary 156

Bibliography **157**

List of Figures

2.1 Example of grammar for a simple arithmetic language. 11
2.2 Parsing tree for the program `if x + y > 10 then x else y` for the grammar depicted in Figure 2.1. 12
2.3 Equivalent implementations of a data-processing task in Python and SQL 16
2.4 Bottom-Up exploration example. 19
2.5 Top-Down exploration example. 21
2.6 CEGIS Loop. The synthesizer produces candidates that are checked by a verifier. If a candidate matches the specification, it is returned as the solution. Otherwise the verifier returns a counterexample which the next candidate must satisfy. 22
2.7 CEGIS synthesis of a program that computes the maximum function over two integer values. 23
2.8 Sketch-based synthesis for a program that return the maximum of two integer values. 24
2.9 Sequence-to-sequence architecture with one encoder and one decoder layers. 27
2.10 Different classification types applied to source code. 31
2.11 Example sparse matrix and its representation using dense and sparse storage formats. 36

4.1 C implementation of matrix vector product and summation. . . . 68
4.2 Lifting C code to TACO using C2TACO. Given a program implemented in C, C2TACO generates a equivalent program written in TACO tensor index notation which the TACO compiler can use to produce high-performance code targeting a variety of hardware platforms. 69

4.3 Architecture of C2TACO. 70
4.4 TACO grammar. 73
4.5 Overall lifting coverage across benchmarks suites. 84
4.6 Distribution of failure causes for the different approaches evaluated. 85
4.7 Lifting time on real-world benchmarks. Y-axis is on logarithmic scale. 88
4.8 Speedup obtained by the synthesized TACO programs on different hardware platforms. The baseline is the average running time of the original implementations when compiled with `gcc -O3`. 90

5.1 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. 95
5.2 Overall architecture of KONRUL. Kernel extraction and IO generation are based on prior-work. 96
5.3 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. 97
5.4 Overview of the post-processing pipeline. (a) Original C code. (b) Compiler-generated tensor code. (c) Post-processed tensor code used to train the Transformer. 100
5.5 The set R 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 5.3. An asterisk (*) indicates that the expression has been introduced or changed by the edit rule. 105
5.6 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. 106

5.7 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. 112

5.8 Lifting coverage by program complexity. Y-axis shows the percentage of programs lifted given different values of highest-dimensionality listed on the X-axis. 114

5.9 Cumulative number of candidates explored during lifting by different IO-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. 115

5.10 Search space exploration by program complexity. Y-axis shows the average number of candidates visited during search given different values of highest-dimensionality listed on the X-axis. An asterisk (*) means that the method could not lift any program in that complexity class. 117

5.11 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. 119

5.12 CBMC verification results. 5.12a reports average verification time across benchmark categories. 5.12b reports verification coverage using floating-point and real datatypes for all programs lifted by KONRUL. 120

5.13 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. 122

6.1 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. 125
6.2 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 126
6.3 Example of program analysis on a type tree. Dashed arrow represents inheritance relation. 132
6.4 Example of TACO program space generation for a MTTKRP operation on a 4D sparse tensor. 136
6.5 Coverage by different kernels. 143
6.6 Coverage by different sparse storage formats. 144
6.7 Relation between candidates explored and lifting time. 145
6.8 Overall speedup of benchmarks lifted to CPU. X-axis lists the benchmarks. Y-axis shows the average speedup over the baseline. 147
6.9 Overall speedup of benchmarks lifted to GPU. X-axis lists the benchmarks. Y-axis shows the average speedup over the baseline. 148
6.10 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. 149

List of Tables

3.1 Comparison of the techniques discussed in this literature review with in terms of extensibility, scalability, and correctness. The ✗ denotes the major weakness of each technique group. 65

4.1 Synthesis coverage of different approaches on the artificial dataset. 83
4.2 Speedup obtained given different tensor dominant orders. We consider the highest order among the tensors in a program as dominant. 91

5.1 Tensor contraction benchmarks taken from [Chelini et al., 2021] and the correspondent einsum counterparts. 110
5.2 Average number of candidates explored vs candidates tested with IO across different benchmark categories. 116
5.3 Average number of candidates explored by KONRUL vs candidates tested with IO by highest dimensionality. 118

6.1 Benchmark suite description. The ᵀ symbol means the benchmark operates on the transpose of the sparse input. 139
6.2 Real-world matrices and higher-order tensors used for performance experiments 142
6.3 Lifting time results. The ᵀ symbol means the benchmark operates on the transpose of the sparse input. 146

Chapter 1

Introduction

For years, improvements in computer performance closely followed Dennard scaling. Dennard scaling [Dennard et al., 1974] is the principle that states that as transistor dimensions shrink, the power density remains approximately constant, meaning that hardware could increase performance by just increasing transistors density without increasing power consumption. In practice, that means processors achieve higher clock frequency and performance improvement without significant architectural changes. Hence, existing programs could simply be recompiled to newer processors to run more efficiently.

However, Dennard scaling slowed down around the mid-2000s and effectuality ended in the early 2010s [Hennessy and Patterson, 2019]. The reason was that as transistors continued to shrink, leakage currents increased significantly and became the main source of power consumption. This has led to a rapid increase in heat generation and power density, a phenomenon known as *the power wall* [Esmaeilzadeh et al., 2011]. As a consequence, processors frequencies plateaued, and automatic performance improvement can no longer be achieved by frequency scaling [Hennessy and Patterson, 2019].

As general processors scaling slowed, hardware has become increasingly complex, parallel, and specialized to deliver application performance. Different architectures such as multi-core processors and heterogeneous systems that combine general-purpose processors with accelerators have been developed to exploit parallelism in more effective ways. Moreover, several domain-specific architectures have emerged, providing tailored programming models that deliver high-performance for particular workloads.

1.1 The Specialized Hardware Scenario

Recent years have seen a significant increase in heterogeneous specialized architectures. Architectures such as Graphic Processing Units (GPU), Neural Processing Units (NPU), Digital Signal Processors (DSP), and other domain-specific accelerators are now widely used for compute-intensive tasks. Unlike general-purpose processors, these architectures trade generality for efficiency by optimizing for particular computation patterns.

However, extracting efficient performance from emerging hardware platforms is difficult with current general-purpose languages and compiler technologies. On modern architectures, performance improvements increasingly rely on exploiting specialization rather than relying solely on compiler optimizations and processor design. Modern accelerators expose complex memory hierarchies, parallel execution models, and hardware-specific constraints that are difficult for traditional compilers to reason about automatically. As a consequence, performance-critical optimizations often require detailed domain knowledge and hardware awareness that are beyond the capabilities of current general-purpose compilation pipelines.

This language/compiler failure has led to the growth of domain-specific languages (DSLs). Frequently, new architectures are accessed using specialized platform-specific DSLs, which deliver excellent cross-platform DSLs provide high-level abstractions tailored to particular application domains, enabling programmers to express intent while allowing compilers to apply aggressive, domain-aware optimizations. By embedding domain knowledge directly into the language and compiler, DSLs are able to deliver high performance across diverse hardware platforms, often outperforming general-purpose approaches. Nevertheless, DSLs are not only interfaces to specialized hardware, but also performance-enhancing tools for a broad range of applications. In fact, they have outperformed general-purpose ones in a variety of domains; and languages such as Halide [Ragan-Kelley et al., 2013] used for image processing or PyTorch [Imambi et al., 2021] used for machine learning are widely adopted in both industry and in academic research.

1.2 The Code Migration Challenge

Accessing the performance available in specialized hardware is relatively straightforward for new applications which can directly programmed in the appropriate language. However, for legacy programs, this is more problematic as existing code bases must be rewritten and ported to target new platforms. Code migration is a highly complex task as the new programs must be rewritten and the final generated code match new execution and memory models, and expose hardware-specific parallelism and data locality to obtain performance.

Rewriting code is a significant issue in both manual and automated cases. For manual migration, the programmer is responsible for both rewriting sections in the new DSL and reintegrating it with the existing application. This is a costly, time consuming and error-prone task, often requiring deep knowledge of both the source and target programming models. Even when performed as a one-off effort, the task complexity deters code migration and consequently the adoption of new hardware technologies.

Code migration can also be approached in a more automated manner through compilers. However, building compilers or compiler extensions requires significant expertise and engineering effort to design program analyses, transformations, and code generation pipelines tailored to each architecture, which is not feasible as hardware platforms and DSLs continue to evolve rapidly.

Although developers rely on vendor-supported tools to generate efficient code for each new platform, these tools expect programs to be written in specific programming models or domain-specific languages. Nevertheless, as code translation into these required formats remains a difficult task, presenting a significant barrier for legacy software to exploit modern hardware.

1.2.1 Existing Approaches

The code migration issue has led to significant interest in exploring automated techniques to rewrite legacy code to access hardware performance without programmer effort. Nowadays, these approaches can be classified into three main groups: rule-based, synthesis and neural machine translation methods.

Rule-based, or pattern matching schemes search the source program for code patterns that correspond to the meaning of a specific DSL or library.

Once a pattern is found, the these methods use a series of predefined translation rules to produce code for the target. However, rule-based translation is often brittle and cannot be extended. Small changes in original implementation can break the pattern recognition and complicated patterns make this non-scalable. Furthermore, rule-based approaches require retooling whenever the target changes, which makes such approaches non-portable.

Another approach is using program synthesis, that is, algorithms for generating programs from specifications. Such techniques have the benefit of being portable and able to generate 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, and these techniques may require an excessively large number of candidates to find the solution, making them impractical for realistic programs.

Finally, neural machine translation and language models have been increasingly used for program translation tasks. 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, which makes them unreliable on their own.

These limitations highlight a fundamental gap in existing code translation techniques. Rule-based approaches lack robustness and adaptability, pure synthesis-based methods suffer from severe scalability issues and neural approaches remain unreliable and data-hungry. As a result, no current approach provides a practical solution able to deliver scalable, portable, automated, and semantically correct translation.

1.3 Thesis

Since emerging hardware is often programmed by DSLs, translating general-purpose code and domain-specific languages can enhance code portability and bridge the gap between legacy implementations and new architectures. The limitation of current code translation techniques highlight the need of new methods to automatically translate code in a fast, scalable, and correct way with little knowledge of the target.

This thesis investigates new methods for specialized hardware adoption using *Program Lifting*: the translation of general-purpose code to higher-level operators or DLSs. As opposed to traditional code translation, lifting converts a program from a lower to a higher level representation while accounting for abstraction differences. This thesis addresses the following research question:

> *Can program lifting efficiently translate legacy code into equivalent domain-specific language implementations to enable software portability to specialized hardware architectures and accelerate programs?*

To answer this question, this thesis introduces a series of lifting techniques that target different DSLs and operators and port code from general-purpose processors to domain-specific accelerators.

1.4 Contributions

This thesis introduces new techniques for program lifting leveraging the strengths of both neural machine translation and program synthesis and combining compiler technology to overcome their limitations. The case study of this thesis is tensor code. In the last decade, we have witnessed a dramatic increase in machine learning use in many diverse applications. Machine learning workloads are dominated by tensor computation [Sidiropoulos et al., 2017], making performance improvement of such code a key concern.

The first contribution of this thesis is C2TACO a lifting tool for lifting dense tensor code written in C to TACO [Kjolstad et al., 2017], a tensor DSL. C2TACO implements compiler static analyses to extract features from the original C code and uses these features to restrict the search space and guide a enumerative synthesizer towards a solution. Using automatically generated input-output examples, C2TACO tests every candidate explored until it find a TACO program that is behavioral equivalent to the original C code.

Although C2TACO effectively lifts complex tensor programs, its enumerative nature has scalability limitations. The second contribution improves the lifting approach with a new lifting methodology named Guess, Measure & Edit. This methodology 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. Based on this methodology, we developed KONRUL a code lifter that targets legacy tensor algebra C programs and lifts them to einsum notation, the basis of tensor DSLs. KONRUL then exports the einsum program to PyTorch [Imambi et al., 2021], which supports this format.

Both C2TACO and KONRUL are possible solutions to the problem of lifting *dense* tensor programs. The third contribution goes beyond and tackles a significantly harder challenge: lifting *sparse* legacy code. SLEB (Sparse LiftEr with Binding) is a new lifting approach to port sparse legacy code to both libraries and DSLs using a large language model (LLM) and program synthesis. This approach queries an LLM for a high-level classification of data format and tensor operation rather than using it for code generation. Given the classification, it generates an API or DSL program sketch with gaps or holes that source code variables must fill. SLEB then implements type-based binding 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. This technique is able to lift legacy sparse code to highly optimized backends, including the Intel Math Kernel Library (MKL) [Intel, 2025] and NVIDIA cuSPARSE [NVIDIA, 2025b] libraries and the TACO language.

These three new techniques contribute towards advancing the sate-of-the-art program lifting techniques. We tackle the high-relevant domain of tensor code and show that lifting aids migration of legacy code to modern hardware accelerators, resulting in significant performance improvements.

1.5 Outline

The remaining of this thesis is structured as follows:

- **Chapter 2: Technical Background** explains the fundamental techniques on which the contributions of this thesis are built, including program synthesis, neural machine translation, verification methods for program lifting. The chapter closes by explaining the concepts of linear and tensor algebra, the case study of the thesis.

- **Chapter 3: Related Work** presents an extensive literature review of the research related to this thesis. The chapter covers automatic code translation methods based on pattern matching, synthesis and neural machine translation, and details modern backends for tensor algebra programming, including DSLs and compiler frameworks.
- **Chapter 4: Lifting Tensor Code with Guided Enumerative Synthesis** presents a guided bottom-up enumerative program synthesis technique that discovers and lifts legacy C code to TACO, a domain-specific tensor language based on behavioral equivalence and on the original program structure. This lifting enables porting such legacy code to multi-core CPU and GPU, resulting in significant speedups.
- **Chapter 5: Guess, Measure & Edit: Using Lowering to Lift** introduces the first verified lifting framework to exploit lowering and source measurement as a means of to edit language model hallucinations. It uses a language model to generate a translation guess and iteratively lowers it, measure it against the input code and based on the measurements applies a high-level edit on the guess to repair it. This is iterated until it finds a equivalent program that is formally verified as equivalent.
- **Chapter 6: Sparse Lifting with Neuro-Guided Sketch Synthesis** describes a sparse code lifter that uses neural guidance from a large language model (LLM) to prune the search space of a sketch synthesis algorithm. Given a input legacy program, this method leverages program classification with an LLM to build sketches in the target language and develops a synthesis technique that binds source code variables to the sketch holes.
- **Chapter 7: Conclusion** concludes the work on this thesis, summarizing the main contributions; reflecting on the approaches used and outlining future work.

1.6 Summary

The end of Dennard scaling has led to the development of new hardware architectures to maintain performance growth. Yet, legacy code needs to be por-

ted to these new architectures to leverage available performance, which is a complex task as emerging hardware is increasingly specialized. This thesis investigates new solutions to automatically port code to specilaized hardware platforms using program lifting, the translation of programs implemented in general-purpose languages to higher-level DSLs. This chapter introduced three contributions of this thesis: C2TACO, Guess, Measure & Edit, and SLEB. Each technique uses a novel approach to lift tensor programs, starting from dense code and proceeding to lift legacy sparse implementations.

Chapter 2

Technical Background

This chapter explains fundamental concepts and techniques to understand the contributions of this thesis. We cover topics that cover automatically code generation. We first explain the techniques of program synthesis, followed by a description of neural machine translation. We then detail some of the mechanisms to formally verify program lifting. We close this chapter by defining the case study of this thesis: the concepts of linear and tensor algebra and how it is used in modern applications.

2.1 Program Synthesis

Program synthesis is the set of techniques that automatically generate a program based on some external specification. Unlike traditional compilation, which produces programs based on syntax-driven translation, synthesis techniques usually perform a search over a program space to find a program that meets the specification. Using program synthesis to generate programs from a specification is a long-studied area [Fedyukovich et al., 2017; Singh et al., 2014].

Synthesis approaches require the definition of both the *syntax* and the *semantics* target program. Typically, the syntax is specified by a context-free grammar (CFG) [Hopcroft and Ullman, 1979]. This grammar specifies the space of all valid programs that the synthesizer will explore. The semantics are defined by the specification, which determine how the target program should behave. Given a grammar $\mathcal{G}$, a language $\mathcal{L}(\mathcal{G})$ defined by $\mathcal{G}$, and specification ϕ, the synthesis task can be defined as finding a program $\mathcal{P} \in \mathcal{L}(\mathcal{G}) \mid \mathcal{P} \models \phi$, that is, a program implemented in the target language that satisfies the specification.

Different synthesis approaches are characterized by the form of the specification and the methodology used to explore the program space. We start defining the main specification formats followed by an explanation of essential synthesis algorithms and how the search for programs.

All the contributions from the thesis use program synthesis. The technique presented in Chapter 4 is based on pure program synthesis, while the techniques presented in Chapters 5 and 6 introduce novel ways of combining synthesis with neural machine translation.

2.1.1 Synthesis Specifications

There are several ways of specifying the intended behavior of the target program for a synthesis task. We will cover the main types typically found in literature: grammars as syntactic specification; first-order logic formulas, input-output examples, and code at different abstraction levels as semantic specifications.

2.1.1.1 Grammar

Program synthesizers typically use grammars to specify the space of programs where the search takes place. A grammar is a set of rules that formally specifies the structure of a valid program given a language. According to Chomsky hierarchy [Chomsky, 1956], programming languages belong to the class of context-free languages , therefore can be generated via a context-free grammars (CFG). The Backus-Naur Form (BNF) [Backus, 1959] is a grammar notation commonly used to specify syntax for programming languages. Formally, a BNF grammar $\mathcal{G}$ is a tuple:

$$\mathcal{G} = \langle \mathcal{T}, \mathcal{N}, \mathcal{S}, \mathcal{R} \rangle$$

where $\mathcal{T}$ is the set of *terminals*, or tokens of the language, $\mathcal{N}$ is the set of *non-terminals*, $\mathcal{S}$ is the *start symbol* and $\mathcal{R}$ is the set of derivation rules, or *productions* of the grammar.

The terminals are atomic elements of the grammar, which means they do not expand to other elements. Identifiers, constants and keywords are examples of terminals of a programming language. The non-terminals are symbols that do not appear in the program final text, they expand into strings of terminals or

program ::= *arith-expr* | *cond-expr*
arith-expr ::= *arith-expr* **+** *arith-expr* |
arith-expr **-** *arith-expr* |
x | **y** | *const*
cond-expr ::= **if** *cond* **then** *arith-expr* **else** *arith-expr*
cond ::= *arith-expr comp-op arith-expr* |
not *cond* | **TRUE** | **FALSE**
comp-op ::= **<** | **>** | **==** | **!=** | **<=** | **>=**
const ::= C_0 | C_1 | C_2 | ...

Figure 2.1: Example of grammar for a simple arithmetic language.

into other non-terminals. The start symbol is a special symbol from which every program is constructed and that expands to the other grammar symbols.

The set of productions consists of the rules to build programs in the grammars. Each production has a left-hand and right-hand sides that are connected via a separator symbol. In a production rule $\mathcal{P} = lhs ::= rhs$, the left-hand side lhs is a single non-terminal, while the right-hand side rhs is a sequence of language symbols that can be either terminals or non-terminals. A production defines a single possible expansion. For example, $\mathcal{P}$ that, during the construction of a program, lhs can be replaced, or *expand* to rhs in order. There can be multiple productions sharing the same left-hand side. In that case, the BNF format abbreviates all the possible right-hand sides in a list separated by the special symbol $|$.

Figure 2.1 shows an example of a BNF grammar for a simple programming language that supports a simple control-flow structure via if-then-else conditional statements and both arithmetic addition and subtraction expressions. This grammar has *program* as the start symbol. The set of terminals $\mathcal{T}$ contains the arithmetic and logical operators $(+, -, <, >, ==, !=, <=, >=, not)$, the arithmetic and boolean constants $(TRUE, FALSE, C_0, C_1)$, the keywords $(if, then, else)$ and the only two variable identifier allowed (x, y). The set of non-terminals contains all the symbols on the left-hand side of production rules:

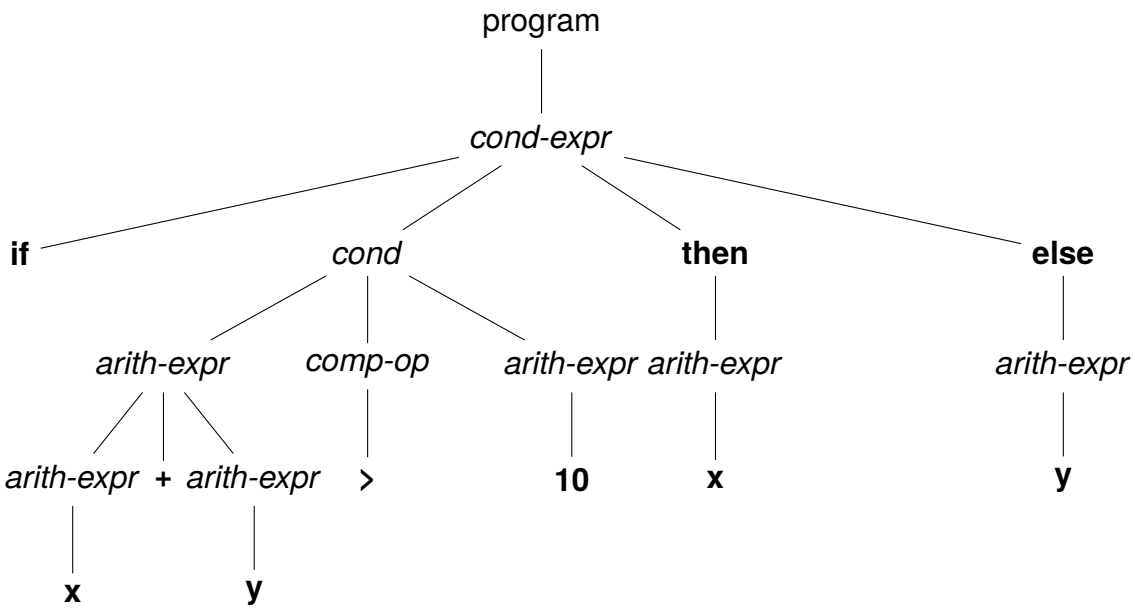


Figure 2.2: Parsing tree for the program `if x + y > 10 then x else y` for the grammar depicted in Figure 2.1.

(*program, arith-expr, cond-expr, cond, comp-op, const*). The non-terminals expand according to the productions shown in the figure. For example, *arith-expr* can expand to:

1. addition expression (*arith-expr ::= arith-expr + arith-expr*)
2. subtraction expression (*arith-expr ::= arith-expr - arith-expr*)
3. terminal identifier (*arith-expr ::= x*)
4. terminal identifier (*arith-expr ::= y*)
5. a constant (*arith-expr ::= const*)

All the productions above are abbreviated with | in figure 2.1. Notice that productions 1, 2, and 5 have on the right-hand side non-terminals which are further expanded in the derivation process.

Parsing. Parsing is the process of converting a program in string format to a *parse tree*. The way parse tres are build is specified by grammars. The start symbol $\mathcal{S}$ becomes the root of the tree. and. Following the productions, the

left-hand side symbols are added as children of the non-terminal symbol on the left-hand side. The parsing process ends when all the leaves in the tree are terminals. In other words, the set of all strings that can be transformed into a parse tree for a grammar $\mathcal{G}$ defines the language specified by $\mathcal{G}$. Although grammars are finite, this set is usually infinite [Webber, 2010].

Figure 2.2 depicts the parsing tree for the program `x + y > 10 then x else y` according to the grammar in Figure 2.1. The original string program can be seen by reading the leaves from left to right. All the contributions from this thesis define grammars as syntactic specification of the lifting targets.

2.1.1.2 Logic

In some synthesis approaches, like Syntax-Guided Synthesis(SyGuS) [Alur et al., 2013], the program specification is provided in the form of first-order logic. First-order logic is a formal logical system to reason about properties of objects and systems. In this type of logic, formulas are built with predicates to describe properties and quantifiers that allow reasoning over multiple variables and functions.

Given a universally quantified first-order logic, the goal is to synthesize a program or function that satisfies said formula for all possible inputs. This type of specification allows SMT solvers such as Z3 [de Moura and Bjørner, 2008] to be used in a synthesis loop to rapidly synthesize candidate programs.

Example. Suppose we want to synthesize a program f to compute the sum of the first x natural numbers starting from y. This program should take two inputs x and y and compute:

$$f(x, y) = \sum_{i=0}^{x-1} (y + i)$$

In first-order logic, we can define the domain of f as $f : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$, meaning that f that takes two natural numbers as input and produces natural number as output. The synthesis task is to find an implementation of f such that the following first-order specification holds:

$$f(0, y) = 0 \ \wedge \ \forall x > 0, \ f(x, y) = f(x - 1, y) + (y + x - 1).$$

Given this specification the synthesizer can generate the following function:

$$f(x, y) = x \cdot y + \frac{x(x-1)}{2}.$$

The main advantages of logic specifications is that it allows the integration of automated verification tools, such as SMT solvers, in the synthesis algorithm to produce programs that are formally guaranteed to satisfy the specification. On the other hand, logical specifications can be difficult for the user to provide, and it may lack the expressivity to capture synthesis tasks in more complex domains. That is the case for linear and tensor algebra, the case study of this thesis. Providing logical specifications that capture the semantics of tensor programs is challenging; therefore, we rely on alternative forms of semantic specification.

2.1.1.3 Input-Output Examples

Input-output-based synthesis is part of the Programming by Example (PBE) style of synthesis, in which input-output examples are used as the specification. In PBE, the user provides a set of IO pairs $\phi = \{(I_1, O_1), \ldots, (I_n, O_n)\}$ and the synthesizer must find a program $\mathcal{P}$ such that $\mathcal{P}(I) = O \quad \forall (I, O) \in \phi$. In other words, for each input in the specification set, $\mathcal{P}$ must produce same output as the set determines. In more complex cases, the user can provide some constraints over the outputs instead of their precise values to help pruning search.

Example. Suppose we want to synthesize a function $f(n)$ that computes the factorial of a number n. The factorial of n, denoted as $n!$, is defined as:

$$f(n) = n \times (n-1) \times (n-2) \times \cdots \times 1$$

with the base case:

$$f(0) = 1$$

We can provide the following input-output set examples for the IO-based synthesizer:

1. **Input:** $n = 0$
   Output: $f(0) = 1$

2. **Input:** $n = 1$
   Output: $f(1) = 1$

3. **Input:** $n = 2$
   Output: $f(2) = 2$

4. **Input:** $n = 3$
   Output: $f(3) = 6$

5. **Input:** $n = 4$
   Output: $f(4) = 24$

Given these examples, the system can synthesize the recursive function:

$$f(n) = n \times f(n-1), \quad \text{for} \quad n > 0$$
$$f(0) = 1$$

There also exists the Programming by Demonstration (PBD) style of synthesis. Unlike PBE, PBD specifications are more detailed, consisting of traces that depicts not just what the output should be, but how it should be computer. For the example above, one example of trace could be $f(4) = 4 * (3 * (2 * 1))$. However this style of specification is less commonly used since it is harder to specify than IO pairs, specially for non-recursive programs.

PBE synthesis is widely used since input-output pairs are relatively easier for users to generate in relation to other types of specification. Moreover, input–output examples can be used to synthesize functionally equivalent versions of components whose source code is unavailable, given that such examples reproduce the behavior of components. Yet, this type of specification is inherently limited: multiple programs may satisfy the same set of input–output examples, and the synthesizer may therefore produce a program that does not reflect the user's intended behavior. In addition, programs synthesized using PBE cannot be formally proven correct and are only guaranteed to satisfy the specification on the subset of inputs represented by the provided examples.

All the contributions from this thesis use IO pairs to test whether the lifted programs are equivalent to the original implementation. Candidates that fail

```python
def process_users(users):
  adults = [user for user in users if user['age'] >= 18]
  updated = []
  for user in adults:
    updated.append({'name': user['name'], 'age': user['age'] + 1})
  return sum(user['age'] for user in updated)
```

(a) Python reference implementation.

```sql
SELECT SUM(age + 1) AS total_age
FROM users
WHERE age >= 18;
```

(b) Equivalent SQL query.

Figure 2.3: Equivalent implementations of a data-processing task in Python and SQL

the IO tests are discarded, while the first successful candidate is returned as the solution. The technique presented in Chapter 5 introduces an additional verification step after the IO testing phase to formally check equivalence.

2.1.1.4 Code

Using a program as the specification is a synthesis method where the goal is to find an implementation that is semantically equivalent to a reference program, usually on a different programming language or paradigm. This type of specification is applied in different domains, such as super-optimization [Phothilimthana et al., 2016] and lifting, where the goal is to find a equivalent program that is more computationally efficient; and in program repair [Le et al., 2016], where a incorrect program can be used to synthesize a fixed version, usually augmented by some other type of correctness specification.

Example. Suppose we are given the Python function shown in Figure 2.3a. That function processes a list of users, records the ones with age greater than or equal to 18, increments their age by one, and finally returns the sum of all

ages. From this function, the goal is to synthesize an equivalent query in SQL that performs the same operations on a relational database. The equivalent SQL query is shown in Figure 2.3b. For this data-processing operation, we can see that the synthesized SQL code is cleaner, shorter, and more intuitive. It employs filtering, transformation, and aggregation into a single declarative expression, eliminating intermediate data structures as we see in the original code.

Using code as specification brings benefits to synthesis efficiency since it provides many features that can be extracted and used to guide the search and it can express more complex semantics than other specification methods. The weakness of such approach is that, except in the program repair setting, a buggy program will inevitably lead to a wrong synthesis result, as the synthesizers typically cannot distinguish intended behavior from errors. Usually, program lifting methods use the original implementation as part of the specification. That is the case for all the techniques presented in this thesis.

2.1.2 Synthesis Algorithms

The algorithms form the backbone of synthesis techniques. They define the strategies employed by synthesizers to explore the program space searching for a solution. According to Gulwani et al. [Gulwani et al., 2017], the synthesis algorithms can be classified into two major groups: *inductive* and *deductive*. Inductive synthesis are approaches generate programs by generalizing from a inductive specification, such as IO-examples. On the other hand, deductive synthesis uses theorem provers to construct a proof of the user-provided specification using logical reasoning and transform the proof into a program. This approach synthesizes programs that are correct by design, because each construction step is justified by a proof obligation.

In this section, we will focus on inductive algorithms, as they are most closely related to the techniques developed in this work. Gulwani et al. [Gulwani et al., 2017] classifies inductive search methods into enumerative, constraint solving and statistical search. The techniques developed in this thesis are based on enumerative search, so we describe those methods in more detail. We begin by describing fundamental enumerative methods, including bottom-up, top-down, counterexample-guided inductive synthesis (CEGIS), and sketch-based

Algorithm 1: Bottom-up enumerative synthesis

input : grammar, specification
output : synthesized program

```
1  Algorithm BOTTOM-UP(grammar, specification)
2      c_list ← TERMINALS(grammar);
3      while true do
4          for r ∈ PRODUCTIONS(grammar) do
5              c_list ← EXPAND(c_list, r)
6          end
7          c_list ← ELIM_EQUIVALENT(c_list);
8          for c ∈ c_list do
9              if IS_CORRECT(c, specification) then
10                 return c
11         end
12     end
```

synthesis. We conclude this section with a complementary discussion on deductive synthesis and other search strategies.

2.1.2.1 Bottom-Up Exploration

Bottom-up synthesis algorithms starts enumeration from the terminals in the target language. It then expands the set of visited programs using the non-terminals in the grammar to construct new terms following the production rules. It keeps combining increasingly larger fragments until a complete program is constructed. Bottom-up explicitly builds all possible programs specified by a grammar, which does not scale as the size of expressions grow. Therefore, this algorithm has a step to eliminate candidates based on equivalence: if two or more candidate programs are equivalent according to the specification, only one is considered, which reduces the number of candidates to check. This selection criteria is usually adapted from the domain of the synthesis work. Finally, all the valid candidates are checked against the specification to assert correctness. This process is illustrated in algorithm 1.

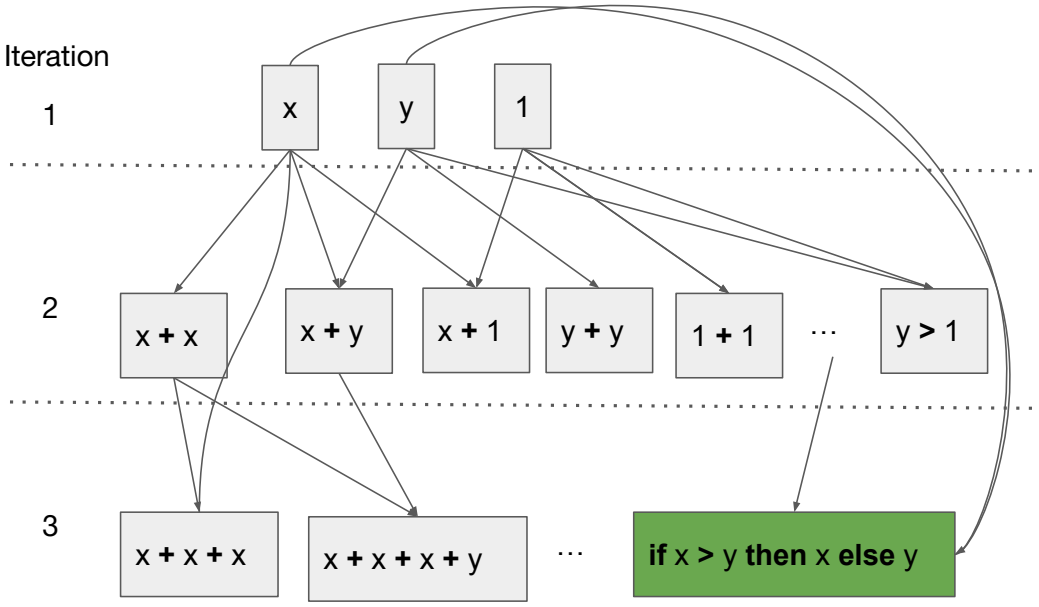


Figure 2.4: Bottom-Up exploration example.

Example. Figure 2.4 shows an example of an execution of a bottom-up algorithm searching for the program that satisfies:

$$f(x,y) = \begin{cases} x & \text{if } x \geq y, \\ y & \text{otherwise.} \end{cases}$$

The figure shows the set of programs constructed during search at three different stages. At each iteration, it uses the grammar productions to expand the set of candidates. In iteration 1, the algorithm builds the initial set with the terminals *x*, *y*, and the constant token *1*. In the second iteration,it builds programs of size 2 using binary operators. Finally, in the third iteration the synthesis,m, finds a program `if x > y then x else y`, which satisfies the specification.

The main advantages of bottom-up search is that it is simple to implement and naturally explores smaller programs first, which is desirable in many scenarios. Additionally, is allows for easy introduction of heuristics to prune search in the expansion and the rejection of equivalent candidates. However, bottom-up enumeration perform poorly when synthesizing constants, due to the large number of possible values that must be considered. Also, the number of candidate programs grows exponentially with program size, which can quickly make the search process intractable and limit scalability.

Algorithm 2: Top-down enumerative synthesis

```
   input  : grammar, specification
   output : synthesized program
 1 Algorithm TOP-DOWN(grammar, specification)
 2     c_list ← ∅;
 3     r_list ← PRODUCTIONS(grammar);
 4     while true do
 5         for r ∈ r_list do
 6             for t ∈ TERMINALS(grammar) do
 7                 if r can expand to t then
 8                     c_list ← EXPAND(c_list, t);
 9                 end
10             end
11             for r′ ∈ PRODUCTIONS(grammar) do
12                 if r can expand to r′ then
13                     r_list ← EXPAND(r_list, r′);
14                 end
15             end
16         end
17         c_list ← ELIM_EQUIVALENTS(c_list);
18         for c ∈ c_list do
19             if IS_CORRECT(c, specification) then
20                 return c;
21             end
22         end
23     end
```

The technique introduced in Chapter 4 uses a bottom-up enumerative algorithm to lift programs with static code analysis aid to overcome scalability.

2.1.2.2 Top-Down Exploration

In top-down synthesis, the search starts from the productions in the grammar to build a set of partial programs. The synthesis progresses iteratively expanding

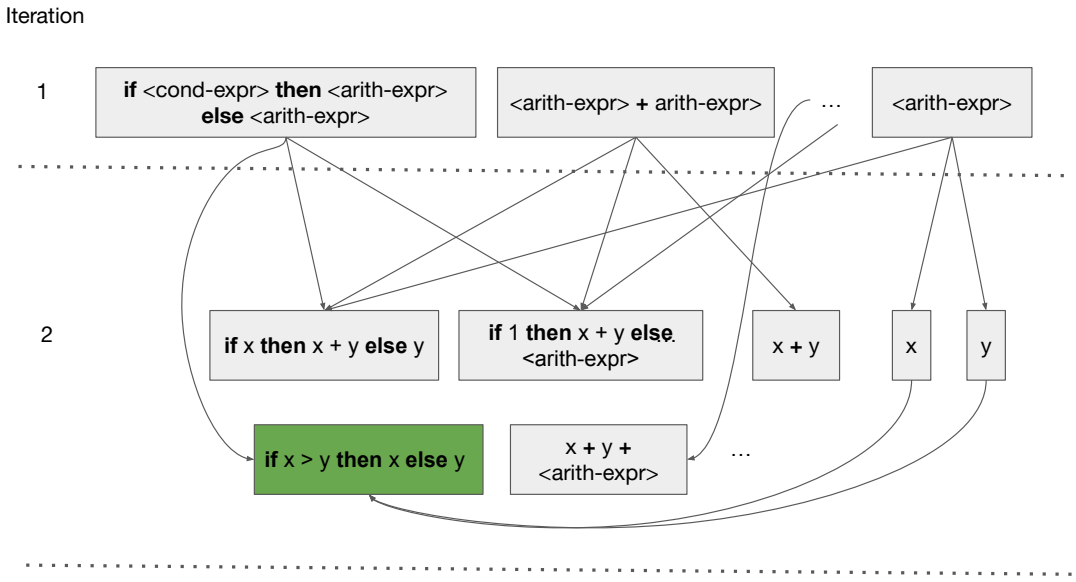


Figure 2.5: Top-Down exploration example.

this set until a set of complete programs is constructed. Top-down search attempts to expand analyzing both the terminals and productions from the grammar to generate complete programs that are then checked against the specification for correctness. Top-down-based algorithms prune the search space based on equivalence, ruling out redundant candidates before checking. Another termination criteria is an optional bound parameter can be passed to limit the number of expansion. Algorithm 2 illustrates the overview of this approach.

Example. Figure 2.5 shows an example of a top-down enumeration that search for a program to satisfy the same specification from section 2.1.2.1. In the first iteration, only incomplete programs with non-terminals are explored. In this case, a equivalent program is found on the second iteration, where the first complete programs are discovered.

Top-down search has proven efficient in synthesize functional programs. It can optimize exploration building a smaller number of candidates given a good initial guess. Furthermore, top-down is better than bottom-up at synthesizing constant values, as the context built by partial programs limits the number of possible values to synthesize. Still, top-down also has scalability problems and it may struggle with recursion if temporary programs need to be executed. None of the contributions of this thesis use top-down synthesis directly. Nevertheless,

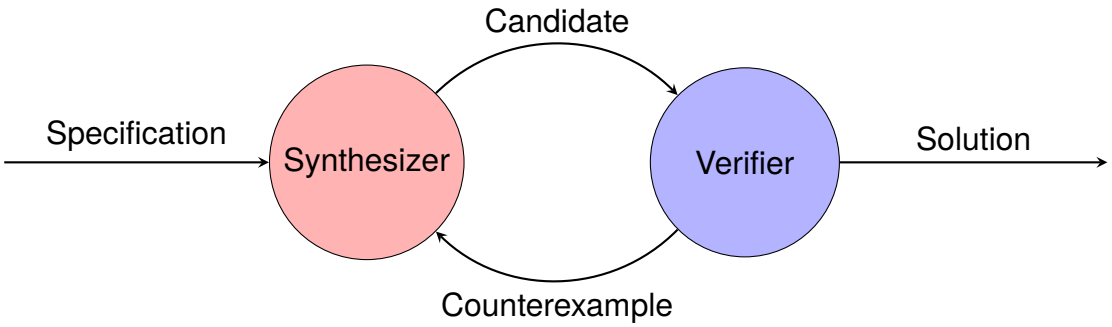


Figure 2.6: CEGIS Loop. The synthesizer produces candidates that are checked by a verifier. If a candidate matches the specification, it is returned as the solution. Otherwise the verifier returns a counterexample which the next candidate must satisfy.

the algorithm presented in Chapter 6 incorporates some elements of top-down enumeration, such as using the exploration context to expand programs during search.

2.1.2.3 Counterexample-Guided Inductive Synthesis

The majority of synthesis algorithms are based on Counter-Example Guided Inductive Synthesis (CEGIS) [Solar-Lezama et al., 2006]. CEGIS is synthesis technique that implements a feedback loop between a synthesizer and a verifier. The synthesizer generates candidate programs that are checked by the verifier component. The key idea is to use a verifier capable of producing counterexample inputs, instead of simply discarding a wrong candidate. This feedback strengths the task specification and allows the use of an inductive algorithm to is require to produce new candidates that are correct for all the counterexamples discovered at that point. This restriction not just discards the current invalid program, but it also rules out every possible candidate that would have also failed on that same specification. Figure 2.6 illustrates this approach.

Example. Figure 2.7 shows the execution of a CEGIS algorithm synthesizing a program that returns the maximum of two integers. At each iteration, the synthesizer proposes a candidate consistent with the current examples seen so far. The verifier either returns a counterexample that is added to the inputs or confirms correctness.

Iteration	Candidate Program	Verifier Output
0	$f(x, y) = 0$	Counterexample: $(x = 1, y = 2)$
1	$f(x, y) = x$	Counterexample: $(x = 1, y = 2)$
2	$f(x, y) = y$	Counterexample: $(x = 2, y = 1)$
3	$f(x, y) = \texttt{if } x > 0 \texttt{ then } x \texttt{ else } y$	Counterexample: $(x = -1, y = 2)$
4	$f(x, y) = \texttt{if } y > 0 \texttt{ then } y \texttt{ else } x$	Counterexample: $(x = 2, y = -3)$
5	$f(x, y) = \texttt{if } x > y \texttt{ then } x \texttt{ else } y$	Verified

Figure 2.7: CEGIS synthesis of a program that computes the maximum function over two integer values.

CEGIS allows for different synthesis and verification mechanisms to be used together, which has made it a widely adopted framework. CEGIS is particularly effective when the incorrect candidates fail on a large subset of inputs, resulting on a larger number of similar candidates being discarded at once. Conversely, its performance degrades significantly when candidate programs satisfy the specification on most inputs but fail only on rare or corner-case instances, since such counterexamples provide limited pruning of the search space.

The approach introduced in Chapter 5 is inspired by CEGIS. However, CEGIS simply discards the current candidate program upon receiving a counterexample, without any attempt to repair it, whereas the approach from Chapter 5 obtains more detailed feedback from the similarity metrics and attempts to repair the candidate, rather than discarding it.

2.1.2.4 Sketch-based Synthesis

Sketching [Lezama, 2008] is a synthesis technique that constraints the syntactic form of the desired solution via *sketches*. A sketch is a partial implementation of the desired solution that contains concrete high-level constructs, such as control-flow structure, but that omits low-level details. The missing pieces are represented by *holes* in the sketch, which act as placeholders for unknown expressions, constants, or subprograms. Users can provide a sketch and the synthesizer searches for assignments to the holes that produce a complete program that can be checked against the specification.

```
1 int f(int x, int y) {
2     if ( ??cond(x,y) )
3         return ??e1(x,y);
4     else
5         return ??e2(x,y);
6 }
```

(a) Sketch with holes.

x	y	$f(x,y)$
1	2	2
3	1	3
-1	4	4
5	5	5

(b) IO specification.

```
1 int f(int x, int y) {
2     if (x >= y)
3         return x;
4     else
5         return y;
6 }
```

(c) Synthesized complete program.

Figure 2.8: Sketch-based synthesis for a program that return the maximum of two integer values.

Example. Figure 2.8 shows an example of synthesis-based synthesis to synthesize a function that returns the maximum of two integer inputs. The user provides a sketch with holes 2.8a and a specification in the form of input–output examples 2.8b. The holes are marked as ?? and highlighted on the figure. The hole at line 2 is a placeholder for a condition expression and the holes at lines 3 and 5 are placeholders for integer expressions. The synthesizer instantiates the holes to expressions $x >= y$, x, and y, respectively to produce a complete program satisfying the specification as shown in 2.8c.

Sketching facilitates the usage of synthesizers as incomplete programs typically are easier for user to provide than other formalisms and it increases user control as it can indicate how they want the generated code to be. Yet, providing a good initial sketch is challenging, especially in complex domains where the desired program is syntactically complex. The algorithm presented in Chapter 6 employs sketch synthesis to lift programs using a language model to overcome the challenge of providing the initial sketch.

2.1.2.5 Deductive Synthesis

As explained in section 2.1.2, deductive synthesis approaches first construct a proof of the specification and then convert that proof into a program using logical reasoning. Unlike inductive approaches, which employ a explicit search, deductive synthesis build programs by applying pre-defined transformation rules in the specification. Although this process may resemble traditional compilation, a key distinction is that compilers apply transformations in a pre-defined order, while deductive techniques uses sophisticated search, often with some user guidance, to determine the order in which rules are applied.

Programs generated by deductive methods are correct by construction, given that the transformation are semantic-preserving, which eliminates the need for validating every candidate. However, this approach is difficult to adopt in practice because it is highly domain-specific due to the reliance on predefined rewrite rules and it assumes that the specification is complete and formal, which in many cases can be as difficult as the synthesis task itself.

Deductive synthesis has proven very effective in domains such as conditional linear integer arithmetic (CLIA) [Reynolds et al., 2015], but it is not well suitable to the tensor programs targeted in this thesis as their implementation is highly complex and constructing a complete formal specification of tensor algebra operations is extremely challenging.

2.2 Neural Machine Translation

Neural machine translation (NMT) is a technique to perform automatically translation using neural networks to translate sentences between two different languages [Stahlberg, 2020]. NMT networks are trained to predict entire sequences of words, assigning a probability for each translation and producing the one with highest probability. Although its rooted in natural language translation, NMT has been recently widely used to translate among programming languages [Roziere et al., 2020; Xinyun et al., 2018; Kim and Kim, 2019].

This section covers the NMT-concepts related to the techniques presented in this thesis. We start describing sequence-to-sequence, the state-of-the-art neural architecture for NMT. We then cover the tokenization process to model source code into embeddings understandable by neural networks. We explain

what language models are and close with program classification, a technique used to identify code regions suitable for lifting.

2.2.1 Sequence-to-Sequence Modeling

NMT techniques apply different solutions to model the probability of a target sentence given a source sentence. Over the year, many different architectures have been used for this task, including networks like feedforward (FFN), recurrent (RNN), and convolutional (CNN). Among these, sequence-to-sequence models have proven particularly successful and have become the main architecture used.

Sequence-to-sequence [Sutskever et al., 2014a; Cho et al., 2014] breaks the translation task into two stages. First, the model processes the tokenized input string and extracts contextual information, and *encodes* it into a numerical representation named embeddings. Second, it *decodes* the numerical representation to generate the output string in the target language.

The encoder is the component responsible for converting the input into a internal numeric representation. It processes the input sequence $i = (i_1, i_2, \ldots, i_n)$ and computes a sequence of hidden states $h = (h_1, h_2, \ldots, h_n)$. Early sequence-to-sequence only pass a single fixed-length embedding vector to the decoding state, typically the final hidden state h_n. This design struggles to handle long sequences that do no fit into the vector. Newer networks use the attention mechanism [Bahdanau et al., 2014], allows the encoder to expose a sequence of hidden states $(h_1, h_2, \ldots, h_n)$ rather than a single vector to the decoder. These hidden states collectively encode the input sequence and are all available to the decoder.

The decoder takes the encoded information as input and generates the output. Usually, a it generates the output on token at a time. Each token is predicted based on a context formed by the internal representation and the previously generated token. In attention-based models, this context is computed dynamically as weighted sum of the hidden states:

$$c_t = \sum_{i=1}^{n} \alpha_{t,i} h_i$$

where the weights $w_{t,i}$ reflect the relevance of each state h_i to the current decoding step t. This allows the model to focus on different parts of the input

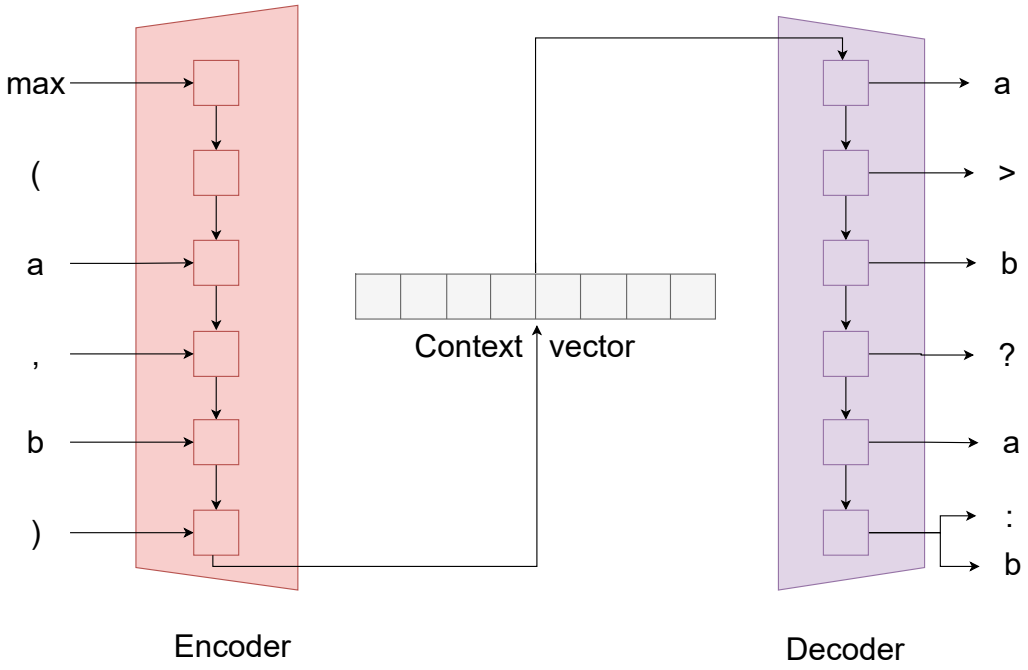


Figure 2.9: Sequence-to-sequence architecture with one encoder and one decoder layers.

significantly improves performance and scalability.

Figure 2.9 illustrates the sequence-to-sequence approach with encoder and decoder to translate the Python function call `max(a,b)` to the semantically equivalent C++ expression `a > b ? a : b`. The example architecture has one encoder and one decoder layers. Separating the translation task into encoding and decoding phases helps the model to handle cases where the input and output have very different lengths. Additionally, different architectures may have a multiple encoders and decoder layers.

Transformer. The Transformer [Vaswani et al., 2017] is the current state-of-the-art architecture for sequence-to-sequence modeling. Unlike other architectures based on recurrence or convolution, Transformers rely entirely on attention mechanisms to model dependencies within sequences. In particular, they employ self-attention within the encoder and the decoder, while the decoder additionally incorporates attention to condition output generation on the input sequence. These attention operations are performed in parallel using multi-head attention, enabling the model to retain from different representation subspaces simultaneously. Furthermore, Transformers incorporate positional

encoding into token embeddings to provide explicit information about token order, which helps modeling the structure of the processed stream. The NMT model described in Chapter 5 uses a Transformer architecture.

2.2.2 Tokenization

Tokenization is the process of converting a raw text stream into smaller discrete units called *tokens*. This process takes place before the encoding phase, as the tokens are then converted into the numerical embeddings for machine learning processing. Tokenization improves models abilities to understand languages as it divides longer inputs into manageable chunks.

There exists different types of tokenization that break down thhe input stream with different granularities. The most common method is word tokenization, where the input string is divided into individual words. Analogously, character tokenization divides the input into individual characters. subword tokenization sets place in between these two, breaking down the input into units that are larger than a single character but smaller than a words. There also exists sentence tokenization, where each token represents a full sentence, and n-gram tokenization, which fixes the size of the tokens. For tasks involving programming languages as source/target, tokenization resembles compiler's lexical analysis, and tokens typically include keywords, identifiers, operators and literal constants. Unlike natural language, programming-language tokenization must preserve syntactic correctness, making it more structured and deterministic.

Byte-Pair Encoding. Byte-Pair Encoding (BPE) [Sennrich et al., 2015] is a subword tokenization algorithms that combines the most-common sequences of characters into a new subword. Unlike the original algorithm proposed by Gage [Gage, 1994], which operates directly on bytes, modern BPE is applied to text in NMT tasks. In BPE, the vocabulary is set of all subwords contained in the model training dataset. BPE iteratively merges frequent pairs of characters until it reaches a desired vocabulary size.

Consider a small corpus consisting of the statements `int total_sum = total + value;` and `int sum_total = sum + value;`. BPE starts representing each identifier as a sequence of elementary symbols, typically characters, and ap-

pending an end-of-token marker. The algorithm then iteratively counts the frequency of adjacent symbol pairs across the dataset and merges the most frequent ones. In this example, character sequences such as ⟨s, u⟩, or ⟨t, o, t, a, l⟩ occur often, hence they are merged to form subwords added to the vocabulary ⟨sum, total . This process produces a vocabulary ⟨int, total, sum, value, _, =, +, ;⟩ that models meaningful code units . At inference time, previously unseen identifiers such as total_value can be decomposed into known subwords, for example ⟨total, _, value⟩, ensuring the model understand out-of-vocabulary (OOV) tokens from the input.

Byte Pair Encoding (BPE) helps handle previously unseen words by decomposing them into smaller subword units that the model has already learned during training. By reducing the effective vocabulary size, BPE makes it easier to scale to large text corpora, and this subword-based representation mitigates the OOV problem and enables more robust modeling of diverse and morphologically rich vocabularies. Both the NMT baseline in Chapter 4 and the guesser component from the technique presented in Chapter 5 use sub-word tokenization with BPE.

2.2.3 Language Models

Language models (LM) are probabilistic models that assigns a probability to a sequence of tokens by modeling the likelihood of each token given its preceding context [Li, 2022]. These models have been implemented using statistical techniques such as n-grams. N-grams language models approximates the probability of a next token in a sequence based on the the previously $n-1$ tokens. Formally, the n-gram probability of a sequence x is given by the equation

$$P(x) \approx \prod_{i=1}^{T} P(x_i \mid x_{i-n+1}, \ldots, x_{i-1}).$$

More recently, language models have been implemented using neural networks, significantly improving performance due to their ability to learn distributed representations and capture complex dependencies. Unlike n-gram models, which rely on a fixed context window of size $n-1$, neural language models use hidden states or attention mechanisms to retain information across long-range dependencies. Furthermore, neural networks share parameters across

all contexts, whereas n-gram models estimate probabilities independently for each context. The probability of a sequence x in a neural language model is defined as

$$P(x) = \prod_{i=1}^{T} P_\theta(x_i \mid x_1, \ldots, x_{i-1}).$$

where P_θ is a conditional probability distribution parameterized by neural network weights θ.

Large Language Models. Recent years have witnessed the rise of Large Language Models (LLM). LLMs are neural language models that contain a large number, often billions, of trainable parameters, and are typically trained on massive corpora using self-supervised learning objectives such as next-token prediction. Most contemporary LLMs are based on the Transformer architecture and leverage attention mechanisms to model long-range dependencies efficiently. LLMs such as ChatGPT [OpenAI, 2022], Gemini [Google, 2023a] and LLaMa [Touvron et al., 2023a] have been shown effective for language generation, reasoning, classification and other tasks.

Although originally designed for natural language processing, language models have been widely used for code translation tasks. The technique presented in Chapter 5 uses a self-trained Transformer-based language mode to guess a solution, and the technique from Chapter 6 leverages a large language model to perform program classification.

2.2.4 Classification

Classification is a supervised machine learning task where the goal is to assign a *class*, or a *label* to the model's input. Machine learning classifiers process the training data to extract features that distinguish between classes and form patterns whose the model can learn. Several algorithms have been developed for this task, for example Support Vector Machines (SVM) [Vapnik, 1997], K-Nearest Neighbours (KNN) [Cover and Hart, 1967; Fix, 1985] and Random Decision Forests [Ho, 1995]. Classification tasks differ in the number of classes to be predicted. They can be *binary*, *multi-class*, or *multi-label*.

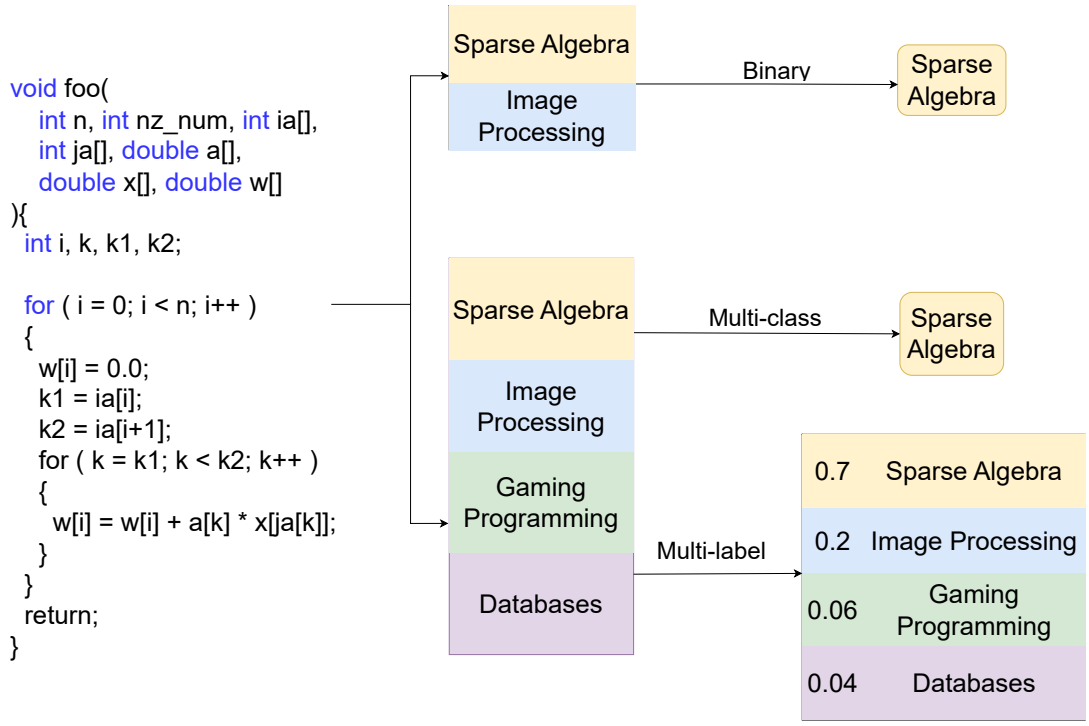


Figure 2.10: Different classification types applied to source code.

In binary classification the goal is to assign a class to the input choosing from just two options. Multi-class classification extends this task and predicts a class from a set of multiple possible choices. Finally, in multi-label classification a data point can belong to multiple classes simultaneously. Typically, the output of binary and multi-class algorithms is a single label that has the highest probability among the options. In multi-label classification, the output is a numeric vector that stores the probability of the input belonging to each possible class.

Figure 2.10 shows an example of input code and classes representing computational domains. The code on the left is a C implementation of sparse matrix-vector produce (SpMV) originally from the CSPARSE [Davis, 2006] benchmark suite. Binary classification predicts the domain of this computation choosing from only two options (sparse algebra and image processing). Multi-class classification selects exactly one class among several candidates (sparse algebra, image processing, gaming programming and databases), and multi-label classification assigns multiple classes with associated confidence scores.

Not all parts of legacy code can be ported to DSLs or APIs, therefore lifting techniques usually use classification to detect the pieces of code that are suitable for translation. The lifting approaches introduced in Chapters 4 and 5 use a program classifier for code clone detection modified to detect liftable regions. The structure of these classifiers is a modest neural network, as training data is relatively low for DSLs, that uses pretrained program embedding. The approach described in Chapter 6 uses a LLM to classify sparse code regions, predicting both the sparse operation and data storage format.

2.3 Verification for Lifting

There exists a huge variety of methods in the literature to verify correctness and equivalence of programs. In this section, we will cover techniques commonly used in program lifting to assert that the lifted program is semantically equivalent to the original implementation.

2.3.1 Bounded Model Checking

Model checking is a verification technique that checks whether a *model* of a system satisfies given properties. This model is constructed as a logical formula constructed encoding the program's execution semantics. The formula is then given to an SAT or an SMT solver which determines whether the desired properties hold for that model.

Model checking tools work by unrolling loops and recursion in programs. When this unrolling is restricted to a certain depth, this is known as *bounded* model checking. The bound limits the number of loop iterations and recursion depth considered during verification; in practice, it may also implicitly restrict the size of inputs through bounded data structures. Bounded tools are often more scalable than unbounded model checking, which can struggle with state-space explosion on large programs. While the use of bounds limits the scope of verification, bounded model checking can provide strong correctness guarantees within the specified bounds.

In bounded model checking, we encode the negation of the property to be verified. Hence, if the generated formula is satisfiable, the solver returns a model that is decoded into a counterexample, representing a concrete execu-

tion trace that violates the specification. If the formula is unsatisfiable, the property is proven correct up to the given bound. Said bound may then be increased to further extend the verification scope.

Bounded model checking tools are very precise and able to handle fine-grain details such as bit-level values. However, they are limited to a certain input size, and can still struggle with exponential growth of the model size. Several successful bounded model checking exist, including CBMC [Kroening and Tautschnig, 2014] for C programs, and SeaBMC [Priya et al., 2022] for programs in LLVM intermediate representation. The lifting technique introduced in Chapter 5 uses CBMC to prove that all the lifted programs are equivalent to the original counterparts.

2.3.2 Post-conditions and Loop Invariants

In program synthesis and verification, verification conditions are formal logical formulas derived from a program and its specifications. Examples of specifications include pre/post-conditions and assertions, which prove the program's correctness when they are valid. For a loop-free program, standard techniques like weakest precondition automatically generate a formula that can be checked in a solver. However, real-world programs often contain loops, which requires the verification conditions to have a loop invariant. A loop invariant is an inductive hypothesis that summarizes the program state at each loop iteration. Although the presence of loops makes verification in general undecidable, loop invariants allow the problem to be reduced to a finite set of conditions that can be checked.

Example. For a program `while B do S` with precondition P and postcondition Q, a loop invariant I must satisfy three conditions:

1. **Initialization**: The precondition P must imply that the invariant must hold before the the first iteration: $P \implies I$.

2. **Preservation**: If the invariant I holds before an iteration and the loop condition B is true, then I must hold after the loop body S completes: $\{I \wedge B\}S\{I\}$.

3. **Termination**: When the loop terminates, meaning B is false, and the invariant I holds, then the postcondition Q must be satisfied: $I \wedge \neg B \implies Q$.

If all three conditions are proven true, the loop is considered partially correct with respect to the given specification. Here, partially means that the prove does not consider termination.

The verified lifting framework [Bhatia et al., 2023] summarizes input programs using verification conditions to correctly lift programs. This approach computes, pre-conditions and automatically synthesizes the loop invariants and post-conditions. Once the correct invariants and post-conditions are synthesized, the summary can be translated to the target language using syntax-driven rules. However, this framework is not well-suited to complex tensor programs, as its SMT-driven symbolic search struggles to scale to deeply nested loops structures typical of tensor contraction kernels and computation manipulating high-dimensionality tensors.

2.3.3 Stuttering Simulation

Stuttering simulation is a proof technique used to verify that two transition systems are equivalent. In program verification the system is composed by states of the program, and transitions represent execution steps. States may be synchronized, meaning that there is a direct correspondence between them in both programs, or may be unsynchronized, means that there is no corresponding state among the two systems. In this case, one system may take several internal steps while the other remains in an equivalent abstract state. Unlike strict simulation, stuttering defines a relaxed notion of equivalence in which sequences of transitions in one system to correspond to a single transition in the other that, enabling verification of systems with no strict equivalence states, as long as the observational state is equivalent.

The sparse lifter SpEQ [Laird et al., 2024] uses stuttering simulation to prove equivalence between a sparse and a dense versions of the same program. The dense version has unsynchronized states corresponding to computation involving zero elements, which are skipped in the sparse implementation. Once equivalence is checked, the dense program can be safely used in a equality saturation loop to find a lifted program equivalent to the sparse input.

2.4 Linear and Tensor Algebra

Linear algebra is the branch of mathematics that operates with vector spaces and linear transformations between them. Linear algebra operates on scalar, vectors and matrices, and it contemplates operations such as vector addition, inner products, scalar and matrix multiplication. A vector is an ordered collection of numerical values, typically representing a point or direction in space, while a matrix is a two-dimensional array of values that represents a linear transformation between vector spaces

Tensor algebra is a generalization of linear algebra to higher-dimensional arrays called tensors. Linear algebra which extend matrices to an arbitrary number of dimensions. Scalars are 0-order tensors, vectors are first-order tensors and matrices are second-order tensors. Tensor algebra typically operates on tensors of order three or higher. Core operations include tensor contraction, outer products, reshaping, and generalized Einstein summation notation.

2.4.1 Dense Algebra

In dense algebra, all the elements in vectors, matrices, or tensors are explicitly stored, including zeros. Dense representations are well suited for data where most entries are non-zero and regular memory access patterns are desirable, which accelerates data throughput. Dense algebra is also highly parallelizable, and number of numerical libraries leverage that representation, suck as BLAS [Blackford et al., 2002], LAPACK [Netlib, 2025b] and cuTENSOR [NVIDIA, 2025c] to optimize this computations on hardware accelerators. The techniques presented in Chapter 4 and Chapter 5 target programs that operate on dense tensors.

2.4.2 Sparse Algebra

Sparse algebra prioritizes scalability and efficiency for large, structured datasets in which the majority of elements are zero. Sparse representations store only the non-zero entries and their indices, significantly reducing memory usage and computational cost for large, structured datasets. APIs such as MKL [Intel, 2025] and cuSPARSE [NVIDIA, 2025b] provide extremely efficient implementations for sparse kernels.

$$A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 3 & 0 & 4 \end{bmatrix}$$

(a) Sparse matrix

```
[1, 0, 0, 0, 2, 0, 3, 0, 4]
```

(b) Dense storage

$\texttt{rows\_index} = [0,\ 1,\ 2,\ 4]$
$\texttt{cols\_index} = [0,\ 1,\ 0,\ 2]$
$\texttt{values} = [1,\ 2,\ 3,\ 4]$

(c) CSR format

$\texttt{rows\_index} = [0,\ 2,\ 1,\ 2]$
$\texttt{cols\_index} = [0,\ 2,\ 3,\ 4]$
$\texttt{values} = [1,\ 3,\ 2,\ 4]$

(d) CSC format

$\texttt{rows} = [0,\ 1,\ 2,\ 2]$
$\texttt{cols} = [0,\ 1,\ 0,\ 2]$
$\texttt{values} = [1,\ 2,\ 3,\ 4]$

(e) COO format

Figure 2.11: Example sparse matrix and its representation using dense and sparse storage formats.

To efficiently handle irregular memory access and dynamic sparsity patterns, sparse implementations leverage different sparse storage formats which are effective depending on the operation and structure of the input. A widely used format is Compressed Sparse Row (CSR), which stores non-zero values row by row along with column indices and row pointers. CSR is particularly efficient for row-wise operations such as sparse matrix–vector multiplication. Its column-oriented counterpart, Compressed Sparse Column (CSC), stores data column by column and is well suited for column-based operations and certain factorization algorithms. The Coordinate (COO) format represents a sparse structure as a list of $(row, column, value)$ tuples, making it simple to construct and manipulate, although less efficient for arithmetic operations. Other formats, such as Diagonal (DIA) and Jagged Diagonal Storage (JDS), exploit specific sparsity patterns to improve performance on structured matrices. Figure 2.11 depicts a sparse matrix and its representation in dense format and in CSR, CSC, and COO sparse formats. The technique introduced in Chapter 6 performs lifting of sparse algebra programs, including operations in all the storage formats described here.

2.4.3 Einsum Notation

Einstein summation (einsum) notation [Einstein et al., 1916] 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 extended version supports other arithmetic operator including subtraction and division; element-wise computation; and constants.

Example. A general matrix-matrix multiplication is expressed in einsum notation as:

$$C_{i,j} = A_{i,k} \times B_{k,j}$$

Notice that the multiplication occurs on the contracted index k and i and j are summed over. A tensor element-wise subtraction order-3 followed by constant scaling can be written in extended einsum notation as:

$$C_{i,j,k} = (A_{i,j,k} - B_{i,j,k}) \times N$$

Einsum notation express tensor computations regardless of the data storage type, that is, dense or sparse. All the contributions of this thesis target some einsum-based language, which is able to express the main linear and tensor algebra implementations.

2.4.4 Applications in Computer Science

Linear algebra plays a central role in scientific computing, optimization, computer graphics, signal processing, and machine learning. For example, many algorithms in numerical analysis reduce to sequences of matrix–vector or matrix–matrix operations, and many iterative solvers have matrix-based sparse operations at their core. High-order tensor algebra has become essential in modern computation due to the recent increase of machine/deep learning, which use multi-dimensional tensors to represent data.

In domains such as image and signal processing and numerical simulations, data is usually stored as dense, since data structures are typically compact

and operations can be efficiently vectorized. On the other hand, graph analytics, recommendation systems, and natural language processing data often exhibits higher levels of sparsity, which make them suitable to sparse storage formats. Applications often combine these paradigms. For example, neural networks use dense tensor algebra for core computations while incorporating sparse representations for embeddings and attention patterns.

2.5 Summary

This chapter described the technical background required to understand the contributions of this thesis. First, the chapter introduced Program Synthesis (section 2.1), followed by Neural Machine Translation (section 2.2). Synthesis is the code of the approach presented in Chapter 4, while the approaches presented in Chapters 5 and 6 introduce novel ways of combining both program synthesis with neural machine translation techniques.

We followed by explaining some verification mechanisms for program lifting (section 2.3). Chapter 5 uses bounded model checking (section 2.3.1) to prove the lifted programs are correct. The verified baselines evaluated in Chapters 5 and 6, that is, baselines that prove equivalence between the original and lifted programs, use the methods described on sections 2.3.2 and 2.3.3 respectively. Finally, the chapter detailed linear and tensor algebra, which are the case study of this thesis. We described its fundamental concepts and discussed how it is widely used in computer science.

Chapter 3

Related Work

In this chapter we discuss how the work presented in this thesis relates to the different areas and techniques to automatically construct code. We will explore how those areas are used in the literature in the following code translation tasks:

- **Transpilation:** Source-to-source compilation, where both input and output are in the same abstraction level. Usually, transpilation occurs among general-purpose languages.
- **Lifting:** Translating general-purpose code to higher level domain-specific languages (DSLs).
- **API Replacement:** Replacing matched code to a fixed Application Programming Interface API call. This is intrinsically a limited form of lifting as such approaches focus on fixed APIs rather than the open-ended nature of DSLs.

Challenges in Automatic Code Translation. Throughout this chapter, we highlight the limitations of existing code translation approaches along three different dimensions. *Extensibility* refers to the ability of an approach to be adapted and applied to domains beyond those for which it was originally designed, including yet-to-be-developed targets. *Scalability* concerns how well an a technique performs as the complexity of the target program increases; in particular, for search-based approaches, it captures the ability to effectively explore large search spaces and synthesize programs of substantial size. *Correctness* denotes whether an approach provides guarantees that the generated program is semantically equivalent to the original implementation. We discuss how existing

techniques have limitations along these dimensions and how the contributions of this thesis overcome those challenges.

We will open this chapter discussing the usage of rule-based, or pattern-matching, techniques (section 3.1). Next, we detail how program synthesis is used to generate code (section 3.2). Finally, we discuss neural machine translation (NMT) based works (section 3.3). We close the chapter presenting some related work in tensor algebra compilation, the domain targeted in this thesis (section 3.4).

3.1 Rule-based Translation

Rule-based code translation follows the traditional compilation paradigm, where a set of predefined rules deterministically determine how a code section or object should be translated to another representation.

3.1.1 Transpilation

Code translation at source level has been long-standing goal. Albrecht et al. [Albrecht et al., 1980] proposed a translator between Pascal and ADA. This paper leverages a common tree representation of subsets of both languages and describes a series of program transformations to perform translation. ADA is also the target of the works by Slape and Wallis [Slape and Wallis, 1983] and Huijsman et al. [Huijsman et al., 1987]. The former developed a technique to modernize Fortran legacy code by also using a language-neutral tree-based intermediate representation that captures the most common Fortran control-flow structure. The latter introduced a mechanical translator that applies standard compilation processes for example parsing, semantic analyses to annotate a intermediate tree and generate ADA from Algol 60 programs.

Baillie [Baillie, 1986] developed a translator from Fortran to C. Ít focused on a subset of Fortran 77 along with matrix and vector extensions from ICL Distributed Array Processors (DAP) Fortran. It was primarily used to convert numerical simulation kernels in Fortran to C code compatible with the GEC Retangular Image and Data GRID processors. A more robust and successful work is f2c [Feldman, 1990], which is able to convert large programs and libraries. f2c has been used to translate software such as the PORT3 [Fox

et al., 1978] mathematical library and Schryer's floating point unit test [Schryer, 1981] framework. It was also used to create CLAPACK [Netlib, 2025a], the C version of the original Fortran linear algebra library LAPACK [Netlib, 2025b]. The original C++ compiler, cfront is also a transpilation tool in the sense that it generated C code as output [Stroustrup, 1996].

Modern languages have also become case studies for source-level translation. S2S [Schiavio et al., 2022] is a framework that uses a set of manually crafted rewrite rules to translate data-processing SQL code to JAVA programs leveraging the Stream API [Oracle, 2025]. S2S takes as input a SQL query and a database definition. During translation, it queries the database to retrieve table and scheme name that will become classes name in the output program. Each translation is automatically checked using a unit test also generated by S2S itself. This technique was used to generate a large-scale JAVA Stream benchmark suite.

DBCert [Benzaken et al., 2022] is a compiler that translates SQL to Nested Relational Algebra (NRA). Such translation is expressed in DBCert's own imperative intermediate representation `Imp`, which express SQL semantics and can be translated to imperative languages. As a case study, the paper converts SQL queries to JavaScript programs that access databases storing JSON objects. The entire compilation process is formally verified using Coq [Huet et al., 1997].

A semi-automatic approach named 3C (Checked C Converter) is proposed in [Machiry et al., 2022], which translates legacy C code to Checked C [Elliott et al., 2018]. Checked C is language extension that augments C with inbounds pointer types to spatially safe uses. 3C uses two novel static analysis. The first one is a constraint-based analysis that detects C pointers that are suitable for translation and then annotates the pointers with Checked C types, either indexed or incremented pointers. The second analysis is used to infer array bounds by correlating the pointers with potential bounds starting from allocations sites. 3C employs heuristics to determine initial bounds in case that information is not available. The bound information is forwardly propagated to the pointer's usage sites. 3C only automates the detection and the annotation tasks and still requires human interaction to refactor the generated code.

More recently, there has been increasing interest in modernizing large C codebases into Rust. The c2rust compiler [Inc and Immunant, 2020] is a major

industrial project that converts C-99 compliant code into Rust. The translation takes place mostly at AST level. c2rust leverages clang to parse and type-check C code, producing a C AST which is converted into a Rust AST using pattern-matching rules. During translation, c2rust employs control-flow analyses to handle C constructs which are not explicitly supported in Rust, like `goto` statements. The resulting Rust AST is then raised back to Rust source-code. The c2rust compiler focuses on preserve the functionality of the original code, not in producing fully safe outputs or leveraging Rust's language features. This is the subject of ongoing research.

Emre et al. [Emre et al., 2021] improve c2rust by implementing secure translation of pointer usage in C to Rust code that uses references with safety annotations. Rust uses an ownership-based model that allows compiler to safely deallocate annotated memory when it is no longer needed. This technique takes c2rust output and uses Rust's borrow checker in order to extract the lifetime, sharing, and mutability information. With that information, raw pointers are turned into annotated references. Another work [Hong and Ryu, 2024], proposes a postprocessing step to use Rust algebraic datatypes. This paper develops an abstract interpretation framework to identify *output parameters* in c2rust generated code. Output parameters are pointers that are only used to store return values, not as inputs to functions. Once these parameters are identified, they are translated to Rust algebraic datatypes, more specifically `Option` and `Result`.

Other tools focus on transpilation of C programs for specific tasks. CRustS[2] [Ling et al., 2022] uses TXL [Cordy, 2006] source-to-source transformation rules and uses the idea of relaxing semantics constraints of program transformations and approximate the behavior of the original program improving API-safeness in C $\rightarrow$ Rust translation. GenC2Rust [Wu and Demsky, 2025] implements static analyses to compute typing constraints on polymorphic void pointers in the C input. Using said constraints, it converts the void pointers into Rust's generic pointers.

Rule-based translation also appears in transpilation of scripting languages. DuoGlot [Wang et al., 2023] is a system that transpiles Python scripts to JavaScript. However, unlike traditional techniques in which the set of rewrite rules is predefined, DuoGlot starts with a basic set that match general features of programming languages, (such as identifiers, conditional statements, etc). In

case the input program cannot be translated using only the rules in the initial set, DuoGlot prompts the user for additional ⟨ source, target ⟩ examples. DuoGlot parses the user examples and derives new transformation rules to grow its rules set. SKEL [Wang et al., 2025a] also translates from Python to JavaScript using the idea of program skeletons to decompose the translation task into fragments and make it scalable to large real-world programs. A program skeleton abstracts lower-level code regions that can be easily translated to the target language while also allows for different ways of completing the output program, which can work with different synthesizers.

Transpilation is widely adopted in contexts where toolchains are mature and the semantics of both source and target language are closely aligned, such as transpilation between C and C++, both of which have low abstraction overhead. It is also used in scenarios with a single runtime environment, such as web development; and in scenarios of very controlled domain, including query and shader languages. Yet, transpilation is less used when there is a significant mismatch in semantic aspects, like memory and concurrency models, or when performance guarantees are hard to provide. Transpilation is also not broadly used in verified systems, where it can increase the verification complexity. Additionally, organizational factors such as composability challenges, fragmented tooling, and resistance to changes in development ecosystems also can further limit transpilation adoption.

3.1.2 Lifting

Early work in lifting [Waters, 1988] proposed a translation paradigm to translate programs from Cobol to a data processing DSL called Hibol. The key idea is to obtain program-independent abstractions that describe the computation performed in the input program with a set of constraints and then re-implement these abstractions in the target language using rule-based code generation.

MOLD [Radoi et al., 2014] is a syntax-driven compiler to convert sequential JAVA into MapReduce-style Scala code. It converts the input code into array Single Static Assignment (SSA) form and uses `fold` operators to keep the JAVA loop structures. Using the array SSA program plus the fold operations, MOLD applies a set of rewriting rules to generate a space of MapReduce programs into an IR which is explored via heuristics with a customizable objective function.

The work by Yang et al. [Yang et al., 2016] introduces a compiling framework that applies domain-specific program analysis and transformations to translate image-processing programs in Python to Cython [Behnel et al., 2010], a language that permits C-like parallel annotations to be added to Python. A key difference of this technique is that the transformations are not applied based only on the pattern-matching, they are selected using an auto-tuner component. This approach is similar to the one presented by Recoules et al.[Recoules et al., 2019], where LLVM IR [Lattner and Adve, 2004] programs are lifted back to C removing inline assembly. The IR module is generated preserving debug information that is used to transform the IR a assembly-free version which is lifted back to C. This IR transformation process is verified using a translation validation component. The resulting C code can then be analyzed by formal verification tools.

In a different domain, QRANE [Gerard et al., 2022] is a tool that lifts programs written in OpenQASM [Cross et al., 2017], a quantum assembly language to the intermediate representation used in the polyhedral frameworks. It takes as input OpenQASM programs that exhibit static contol-flow and performs delinearization to generates polyheadral abstractions and memory access relations from linear relationships among qubit indices. QRANE employs different policies to select delinearization strategies that are shown to affect the quality of the lifted program.

Multi Level Tactics Multi Level Tactics (MLT) [Chelini et al., 2021] performs lifting of LLVM IR programs to high-level MLIR [Lattner et al., 2021] dialects using domain-specific rules to match and translate code patterns. MLT proposes a declarative Tactics Description Language (TDL) that define the lifting transformations independent of program structures such as loops and iterators. MLT also uses TDL to define optimizations for matrix-chain multiplications after the raising is performed.

Automatic program parallelization can also be understood as lifting, where instead of producing code directly, legacy implementations are lifted to higher-level abstract representations general implementations are lifted to higher-level abstract representations, such as polyhedral model [Benabderrahmane et al., 2010], which enable the application of analyses and transformation to optimize the computation. The lifted format can be parallelized and lowered back to a executable format. As an example, Calotoiu et al. [Calotoiu et al., 2022]

use lifting as a method to automatic parallelize C code. This work develops a series of translation passes to lift C code to a high-level representation where the dataflow of the original program is analyzed, optimized and lowered back to a parallelized C version. Using symbolic analysis, the authors keep track of memory accesses via patterns. This enable automatic exposition of data parallelism by identifying and optimizing updates to shared memory locations.

Lasagne [Rocha et al., 2022] is a hand-written binary translator that lifts assembly to LLVM IR to enable translation between x86 and ARM assembly. The authors proposed a novel memory model LLVMIR Concurrency Memory Model (LIMM) to mitigate the differences in memory ordering and rules between x86 and Arm. This concurrency model enables Lasagne to design mapping between the two memory models and prove the translations correct.

Rule-based approaches often contain rules that are tailored to the lifting target and cannot be extended to other domains. Additionally, defining lifting rules requires manual effort and domain expertise, which can be costly to maintain as the target evolves.

3.1.3 API Replacement

Several pattern-matching based approaches have been developed to adapt legacy code to use high-performance APIs and other software libraries. Although many such techniques are domain-specific, general methods also exist. For example, Patternikka [Blech et al., 2021] is a general tool that uses code gathered from open-source repositories to create some common code patterns and based on those generate a set of AST-based library migration rules. It is evaluated successfully across eight different scenarios.

Parallel code is a common target of API replacement techniques. Lee et al. [Lee et al., 2009] presents a system that automatically translates parallel regions of code annotated with OpenMP to CUDA-based General-Purpose Graphics Processing Units (GPGPU). The translation occurs in two phases. Initially, the technique applies compile-time optimizations to transform general CPU OpenMP programs into OpenMP programs optimized for GPGPUs. Second, the resulting optimized program is converted into CUDA GPGPU and it exploits CUDA-specific features. Concrat [Hong and Ryu, 2023] is a C to Rust translator

for concurrent programs that replaces lock API in C code with the equivalent API in Rust, which guarantees the correct use of locks with type checking. Concrat is a program transformation tool that takes a lock summary as an input and generates summaries of said locks with static analyses.

Both Desynchronizer [Gokhale et al., 2021] and Escapin [Kimura et al., 2018] rely on static analyses to identify synchronous API calls in JavaScript programs that can be safely replaced by asynchronous counterparts and suggest such change to the user. The main difference between the two techniques is that Desynchronizer focuses on semantic and control-flow restructuring analysis, inspecting program points that propagate asynchrony through the program; while Escapin heavily relies on API usage pattern recognition and abstraction lifting to transforming common usage idioms and anti-patterns into cleaner, safer constructs.

Most of the API replacement work for linear algebra is focused on dense operations. KernelFaRer [De Carvalho et al., 2021] works at intermediate representation level and is focused on general matrix-matrix multiplication and symmetric rank-2k (SYR2K) update API targets. KernelFaRer analyzes the order of access to matrices and SYR2K and uses pattern matching and loop information retrieved in LLVM (IR). A more robust approach is Source Matching and Rewriting (SMR) [Espindola et al., 2023], which works at MLIR level. However, SRM requires a idiom description as input in addition to the original program to perform the matching. It performs automaton-based tree-matching at control dependency (CDG) and data dependence (DDG) graphs (CDG, and DDG). It supports both MLIR dialects CIL for C and FIR for Fortran and successfully migrates code to use BLAS [Blackford et al., 2002].

There is considerably less research in matching sparse code to accelerator libraries due to the complexity of the task. LiLAC [Ginsbach, 2020] is a development of IDL[Ginsbach et al., 2018], but it is restricted to sparse matrix-vector multiplication (SpMV). It uses a vanilla implementation of the SpMV to generate constraints that are used to search the legacy code for suitable code regions and replace them with calls to different sparse libraries. More recently, SpEQ [Laird et al., 2024] proposes pattern-matching along with equality saturation to recognize sections that can be replaced with API call. It takes as input a sparse algebra kernel in LLVM IR and convert it to a functional IR called FIR. In FIR, SpEQ identifies the storage format inspecting the data dependence graph

of the code and matching it against reference implementations and identify a set of preconditions that represent the matched storage format and create a precondition-free version of the input code that is dense. Once SpEQ proves equivalence between the sparse and dense versions the FIR program is added to an E-Graph and equality saturation is used to find a equivalent form of expressed in the target backends. Nevertheless, SpEQ is limited to only a few operations (SpMV and Histograms).

As well as in lifting, rule-based approaches for API replacement are brittle and hard to adapt to new targets. Whenever the target API changes, such techniques require significant changes, making them non-portable.

3.2 Program Synthesis

Using program synthesis to generate one program that is equivalent to another has been explored in super-optimization [Phothilimthana et al., 2016], deobfuscation [Jha et al., 2010] and program lifting. We will focus on the later as it relates closely with the research targeted in this thesis. We start by introducing IO-based synthesis and then follow the layout described at the beginning of this chapter.

3.2.1 IO-based Synthesis

Early work on IO-based synthesis looked at generating string processing commands from a few examples [Gulwani, 2011]. The paper presents an algorithm that represents the search space as a directed acyclic graph (DAG) and uses user-provided IO example to refine the specification. The approach is very efficient time-wise and has been incorporated in Microsoft Excel as the `FlashFill` feature, allowing users to generate Excel commands by just providing examples.

The same concept has been used to predict the behavior of existing components or libraries. PRESYN [Collie et al., 2020b] is a technique that uses high-level control-flow structures to search for equivalent implementations of program components in a *black-box* fashion, that is, without any internal information other than API call or function signatures. It uses IO examples to assert behavioral equivalence between the original component and the synthesized program. A more robust approach is presented in HAZE [Collie and O'Boyle,

2021], where again the information available is limited to signatures. However, HAZE targets the synthesis problem using *gray-box* synthesis, extracting several runtime information from the original component, such as execution time, instructions and memory traces. Using this strategy, HAZE is able to synthesize more complex programs from different application domains.

SIMPL [So and Oh, 2017] proposed a new algorithm for efficiently synthesizing array manipulating programs from IO examples plus a partial program and a set of array/integer variables and constant values that it uses to help start the search. It uses bottom-up enumeration to fill the input partial program and static analyses to prune the search space by discarding states that will not lead to a valid solution. The analyses perform a combination of numeric and symbolic execution to determine abstract values of variable and use them to build a set of constraints. Any candidate program that does not satisfy the constraints according to the IO examples is ruled out during search. HOOGLE+ [James et al., 2020] is a tool that synthesizes Haskell programs based on either IO examples or type information. It uses type-directed program synthesis with heuristics based on property-based test to eliminate bad candidates during search. Additionally, HOOGLE+ automatically generates new inputs and outputs that demonstrate the behavior of each candidate solution, helping users to select the best one.

Alternative synthesis techniques use deep learning models to guide the generation of code from IO examples. Seminal work by Balog et al. [Balog et al., 2016] used a small Feed-Forward Network (FFN) to produce a probability distribution over the tokens of the target language. Such distribution specifies which tokens are more likely to be in the solution given the IO specifications. The paper shows that different search methods, like Depth-First Search (DFS) and sketch-based synthesis can leverage the distribution to decide the how to expand partial candidates during enumeration. In the paper, the authors target a DSL to manipulate arrays of integers. An evolution of this method is presented in PCCoder [Zohar and Wolf, 2018], a synthesis method that inputs the current program state to a neural network to predict the subsequent state. It represents the state including all intermediate variable values and the desired output across all examples and feeds this into the network to predict the next statement, variables which are safe to drop and functions type. During synthesis, it uses Beam Search to explore likely sequences of statements efficiently and a

garbage-collection prediction to manage memory to make the search scale to longer sequences.

Neural-guided IO search is also used in AutoPandas, [Bavishi et al., 2019] to generate programs using the Pandas data analysis API [McKinney et al., 2011]. AutoPandas implements a program candidate generator that encodes constraints of the Pandas API to produce well-formed programs. The generator uses neural models trained on example programs to rank and select values during search in a fine-grain fashion, meaning that the neural network makes specific choices during generation, rather than first predicting an entire program in a separate step. BUSTLE [Odena et al., 2020] employs a different modeling task to guided search. In BUSTLE's search, each program explored is an expression that can be executed on the input examples. Then, the technique uses a classifier trained to predict whether the output of the execution of a partial expression is part of an eventual solution. Using that prediction, BUSTLE performs bottom-up enumeration and is evaluated on SyGuS competition problems.

In the tensor application domain, TF-Coder [Shi et al., 2022] is a synthesizer that generates TensorFlow [Abadi et al., 2016] programs. It implements a weighted bottom-up enumerative search algorithm that builds expressions by composing operations in order of increasing weight and it uses with value- and type-based filtering to eliminate equivalent expressions based on observed values. TF-Coder uses two guiding models: the first is a language model conditioned on features of the IO tensors examples and the second is bag-of-words natural language model that takes as input an optional textual description of the task being performed by the desired program. Both models are used to adjust the weighs of each TensorFlow operator and aid search.

The rise of Large Language Models (LLM) has naturally led to neural-guided synthesis approaches that use these model as search aid. Although LLMs struggle to generate efficient code for domain-specific languages [Gu et al., 2025], several work has demonstrated success in combining LLMs with synthesis. Currently the main strategy is to use an LLM to produce a series of candidates solutions and from that build a probabilistic context-free grammar (pCFG) which is converted to a weighted grammar (wCFG) which serve as the syntactic specification for an enumerative synthesizer.

One of the earliest such efforts is due to Li et al. [Li et al., 2024] who propose two complementary integration strategies to solve SyGuS problems. This paper proposes two different methods of this integration. In the first approach, enumeration is guided toward promising regions of the program space using the wCFG build with LLM guesses. The second approach leverages incorrect candidates generated during enumeration: these partial or failing programs are fed back to the LLM, which is prompt to complete or repair them. The resulting LLM-generated helper functions are then used both to augment the grammar's production rules and to update their associated weights, further refining the synthesis process. HySynth [Barke et al., 2024] uses a very similar strategy. Given a IO examples, HySynth also prompts an LLM for a set of possible solutions. It builds a pCFG assigning probabilities for each rule in the grammar based on their frequency observed during the parsing of LLM guesses. HySynth is successfully evaluated on three different domains: grid-based puzzles, string and tensor manipulation problems.

Recently, STAGG [Li et al., 2025] proposes an LLM-guided synthesis technique to lift dense tensor algebra programs. Unlike other works, STAGG builds a grammar of templates and separates search into two stages. First, it uses A*-based search to explore template candidates, and then given a template candidate, STAGG searches for a valid substitution of template holes for values from the IO specification. Additionally, STAGG performs bounded verification to validate a completion as correct.

Some prior synthesis work derive metrics from the specifications to guide the search. The following two works construct programs by enumeration. Euphony [Lee et al., 2018] 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. 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 [Feser et al., 2023] enumerates the space of programs with bottom-up search and then clusters these programs into equivalence classes based on an expert-provided distance metric, which has to be redefined for each domain. The metric is focused on the *semantic* similarity of the input/output behavior of programs. Then, it proceeds to a local search that greedily takes the most representative program from a cluster as initial point and applies rewrite rules to mutate it until it finds a equivalent program.

3.2.2 Transpilation

There are cases where synthesis for transpilation is used with the same language being both the source and the target, but transpiling it to add new features such as paradigm changes or optimizations. [Zhang et al., 2021] presents a technique to translate SQL queries with User-Defined Functions UDFs into pure-SQL queries. Their approach introduces a lazy inductive synthesis which consists of using the dataflow graph representing the UDF to break it into subproblems and then solve each one of those subproblems using CounterExample-Guided Synthesis (CEGIS). A different problem is tackled in [Mariano et al., 2022], where inductive synthesis is used to transpile imperative JAVA and Python code to a functional versions in the same languages using the Stream and `functools` APIs. This technique builds upon the observation that both imperative and functional versions of the programs share expressions which take the same values in corresponding execution traces. Their top-down enumerative synthesis algorithm is guided by a novel neural network called cognate grammar network (CGN) trained on pairs of equivalent imperative and functional programs and it uses intermediate values of partial computations to prune the search space.

RDD2SQL [Zhang et al., 2023] is a transpilation tool to automatically translate functional queries written in Spark RDD to SQL queries. It uses classical CEGIS combined with a new technique named column-wise decomposition. Column-wise decomposition works under the observation that a query that produces a table with N columns can be divided into N queries each of which produces a column in the resulting table. RDD2SQL adapts that idea to decompose the synthesis problem into smaller ones, reducing the search space it needs to explore.

Lakeroad [Smith et al., 2024] uses sketch-based program synthesis to map hardware description language (HDL) designs to Field-Programming Gate Arrays (FPGA) primitives. Lakeroad takes as input a user design in HDL, the description of the FPGA architecture and a sketch primitive template. From the sketch template, it generates an architecture-specific sketch incorporating the semantics of the target. Lakeroad uses different solvers within the Rosette synthesis engine [Torlak and Bodik, 2014] to find a program equivalent to the input design, which is compiled to Verilog code that maps directly to FPGA hardware.

3.2.3 Lifting

Program synthesis has been used for lifting in many works. For lifting targeting specialized hardware, the work by Angstadt et al. [Angstadt et al., 2020] presents a method that lifts part of legacy code to a hardware description level in order to enable such code to efficiently run on FPGA platforms. This paper proposes an L^*-based algorithm to convert legacy functions into a automata that recognizes a language equivalent to input function. Their technique uses a combination of software verification with an SMT solver to guarantee the correctness of the automata, which is then converted to a HDL module. This solution is limited to functions that take strings as input and produce boolean values. Dyospiros [VanHattum et al., 2021] is a synthesis-based compiler that uses symbolic evaluation in search. Once the program is lifted, Dyospiros performs equality saturation to optimize the target program. This technique is used to automatically port legacy programs to digital signal processors.

A different search strategy is implemented in HeteroGen [Zhang et al., 2022]. HeteroGen is a framework to automatically lift C/C++ programs to high-level synthesis (HSL-C) modules to allow easier portability to program accelerators. Unlike traditional search-based synthesis, it uses evolutionary repair to iteratively explore the program space. It prunes the search space based on common development patterns using a LLVM-based pattern checker that discards candidates that do not conform with pre-established coding styles. This pruning allows for efficient lifting since it avoids the high cost of HSL compilation.

mlirSynth [Brauckmann et al., 2023] applies bottom-up enumeration in the MLIR IR space, lifting from lower- to high-level MLIR dialects that are optimized for linear algebra, such as Linalg. Rather than relying on manually defined rewriting rules, it builds the search space using available dialect definitions, making it extensible to different dialects. mlirSynth uses type constraints and IO equivalence to discard candidates and find equivalent programs. An alternative approach [Brauckmann et al., 2025] employs top-down sketching to lift legacy programs to a range of tensor DSLs. It runs legacy programs to obtain a symbolic execution trace as synthesis specification and automatically selects sketches from the target DSLs. Such specification is simplified using an algebraic solver, resulting in a lifted program that is correct by construction. The cost of this approach is of potentially finding sub-optimal solutions.

Dong et al. [Dong et al., 2025] proposes a neuro-symbolic lifting technique to lift tensor programs. The technique consists in breaking the lifting task into multiple transformation passes. Each pass is guided by an LLM that suggests code rewrites given some characteristics of the target. Each time a pass is applied, QiMeng-Xpiler performs constraint-based symbolic synthesis to automatically repair the output and guarantee correctness. It also includes a hierarchical auto-tuning component to select the best parameter choices for each target hardware, such as tiling sizes and the best order of transformation passes. This work also tackles tensor programs and translates among four different tensor DSLs.

Verified Lifting is a synthesis-based technique that produces lifted programs proven equivalent to their lower-level counterparts. Verified lifting computes a summary of the input program and uses it to generate verification conditions—typically expressed as Hoare triples [Hoare, 1969] consisting of a precondition, loop invariant, and postcondition. Preconditions are inferred, and program synthesis is used to generate loop invariants and postconditions The verification condition is then verified using an SMT solver. Once proven correct, the computation summary can be translated to the target language using compilation rules. Verified lifting is in principle domain-agnostic, and general frameworks like MetaLift [Bhatia et al., 2023] enable the development of task-specific lifters. Nevertheless, verified lifting remains very user-demanding, requiring the user to define the syntax and semantics of their target language as well as a set of compilation rules for each potential source and target.

Early work in verified lifting [Cheung et al., 2013] lifted JAVA code regions that access databases using object-relational mappings to SQL code. The technique identifies code fragments that use the Hibernate ORM API [Gregory and Bauer, 2015] and inlines methods that call and that are called by said fragment. For each candidate fragment, it identifies the variable that will contain results of a query and translates this code to an intermediate language. Then, it computes verification conditions containing unwon loop invariants and postconditions which are synthesized and verified by an SMT solver. Once the conditions are verified as correct, the candidate region is translated into SQL using syntax-driven rules and embedded back into JAVA code.

Verified Lifting has been successfully applied in diverse contexts. The same general framework is used, with some domain-specific solutions and optimiz-

ations. Kamil et al. [Kamil et al., 2016] developed STNG, where appropriate stencil-like loops in Fortran are lifted to their equivalent in Halide [Ragan-Kelley et al., 2013]. STNG starts identifying stencil loops and abstracts their behavior into a defined predicate language that captures the pattern of neighbor accesses and data dependencies explicitly. Due to multi-dimensional indexing and access patterns, the loop invariants in stencil domain are complex. STNG implements a combination of concrete and symbolic execution of the original code to capture the relation of the outputs with the inputs across the image grid, which is then used derive summaries and invariants needed by the verified lifting framework.

Continuing with image processing domain, DEXTER [Ahmad et al., 2019] lifts C++ to Halide. After selecting the candidate region, DEXTER translates that region into a Directed Acyclic Graph (DAG). Every node in the DAG corresponds to a loop-nest in the source code which are translated to functions in the target language. For each operation in the DAG, DEXTER summarizes three components: the Region of Interest (ROI), which describes the range and dimensionality over which the operation is executed; its terms, which is the set of all array reads, constants, and scalar variables used; and how each point in the output buffer is computed using the terms. Using these three components, DEXTER is able to generate IR code that goes that is verified.

CASPER [Ahmad and Cheung, 2017] is a compiler that uses verified lifting to automatically convert imperative code in JAVA to Spark [Lhoták, 2003], a DSL that uses the MapReduce paradigm [Dean and Ghemawat, 2008] to express computations. CASPER summarizes JAVA loops that are suitable for lifting and feeds them as input to a syntax-guided sketch synthesizer to generate the invariants and post-conditions. It uses Dafny [Leino, 2010] to verify the post-conditions. This technique is later extended [Ahmad and Cheung, 2018] to summarize JAVA programs into a language-agnostic MapReduce intermediate representation to target different DSLs. Since the performance of candidate programs may vary depending on characteristics of the input data, the synthesis algorithm generates different but equivalent programs for a given input and implements a monitor module to switch among the different candidates depending on runtime performance statistics. Besides Spark, this extension of CASPER supports Hadoop [White, 2012] and Flink [Carbone et al., 2015] as lifting targets.

The work by Zhan et al. [Zhan et al., 2024] proposes to lift deep learning operator implementations to higher-level mathematical abstractions. This approach extracts semantic information from the input code with symbolic execution, uses syntax-guided synthesis to search the space of possible mathematical expressions, and generates invariants that are given to an SMT solver along with the abstraction to verify correctness. A novel step is that this technique e-graph–based simplification with custom rewrite rules in the verified program to reconstruct the mathematical abstractions into a more intuitive high-level form. It has been evaluated on lifting kernels in Triton [Tillet et al., 2019] to mathematical formulas.

The Metalift verified lifting framework [Bhatia et al., 2023] has been applied to map matrix multiplication and convolution written in Python and C++ to the Gemmini DNN accelerator [Nishida et al., 2023]. This technique specifies the semantics of Gemmini operators and uses Metalift's synthesis engine to search and verify for equivalent programs. Metalift is also used by Tenspiler [Qiu et al., 2024a] to automatically lift sequential code to a range of different tensor processing languages. The core of Tenspiler is an IR (TensIR) that makes synthesis tractable by abstracting operations commonly used in tensor computations across multiple backends. Tenspiler also implements an underlying symbolic synthesizer that generates TensIR programs and invariants that prove the program is correct using formal methods. Tenspiler then generates backend-specific code for the proven correct TensIR program.

Recent work [Bhatia et al., 2024] integrates few-shot learning with GPT-4 [Achiam et al., 2023] within a verified lifting framework to suggest candidate solutions and invariants. Both the summary and the invariant are expressed in Python, which is used as a form of high-level IR used in traditional verified lifting. Python was chosen due to its large presence in the data used to train LLMs, which in principle boots GPT-4 ability. Here, the traditional verified lifting workflow is used, except that instead of using synthesis to generate computation summary and the loop invariants, these are generated by prompting the model. First, the model is prompted to generate the summary and then prompted to generate the invariant given an example. The invariant is translated to SMT-LIB and verified by a solver. Once proved correct, the model's output is translated to the target language using rewrite rules provided by the user.

Verified lifting has also been used to perform instruction selection. RAKE [Ahmad et al., 2022] automatically maps high-level expressions in Halide to DSP architectures. It proposes *uber-instructions*, which abstract groups of concrete DSP vector instructions like arithmetic and data movement, while preserving their semantic effects on data. RAKE searches for sequences of uber-instructions that are semantically equivalent to the source computation. A sequence that is verified correct is lowered to DSP machine instructions.

Synthesis-based approaches have also been used for binary lifting, or decompilation. A tree-based exploration framework is proposed in [Rodríguez et al., 2016] to reconstruct source-level affine loops from binary code. This technique searches over memory traces of programs, forming a space from which it constructs mathematical restrictions that constraint search. This technique was able to reconstruct static loops in the PolyBench benchmark suite. Helium [Mendis et al., 2015] is a lifting tool that correctly lifts stencil kernels from x86 binaries to Halide. It uses dynamic analyses to find performance-critical regions in a binary and reconstructs memory access buffers. Helium proposes a technique named expression forest reconstruction. From the buffers, Helium constructs dependency trees and searches for one that corresponds to all the input-dependent control-flow paths in the input binary. Expression forest reconstruction canonicalizes dependency trees, clusters them based on structure similarity and infers high-level Halide expressions by solving a set of linear equations based on the buffer accesses.

Current synthesis approaches for lifting 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. Verified lifting overcomes that by using SMT reasoning instead of straight enumeration. However, this technique requires the user to provide a compiler and decompiler from each potential source and target into the IR, which is not extensible.

3.2.4 API Replacement

M^3 [Collie et al., 2020a] address the API migration problem using probabilistic and SMT-based synthesis to migrate legacy code to string processing and mathematical APIs. The key idea is to learn semantic models of library functions by creating IO examples from observed calls. These models form a beha-

vioral representation of the original code and are used to drive an SMT-based search to find semantically corresponding patterns in large applications. Once matches are found, M³ can suggest or perform concrete API substitutions. M³ successfully learns correct implementations for legacy functions and discovers numerous migration opportunities in real-world code databases.

SOAR [Ni et al., 2021] combines Natural Language Processing (NLP) techniques with enumerative synthesis to migrate code . It learns a NLP model using the target API documentation. and to use error messages as feedback to the synthesizer. Once it finds the correct function to call, SOAR uses a synthesizer to correctly construct the method call. Additionally, SOAR uses NLP techniques to use error messages as feedback to the synthesizer to improve and refine the generated programs. It has been successfully evaluated in the data science domain, migrating code from TensorFlow [Abadi et al., 2016] and PyTorch [Imambi et al., 2021] to R's dplyr [Yarberry, 2021] and Pandas [McKinney et al., 2011] data manipulation libraries.

APIFix [Gao et al., 2021] is an IO-based synthesizer that adapts programs to newer versions of the same API, minimizing the problem of library modification. This technique automatically infers API usage transformation rules from both human-adapted migration examples and example usages of the new library in other clients. APIFix uses the additional outputs to produce transformation rules to guide the synthesis process. These additional examples help the synthesizer to produce more general programs and rules, avoiding overfitting to the examples. It has been able to fix API breaking changes from multiple widely used .NET libraries.

FACC [Woodruff et al., 2022] uses synthesis to fill the mismatch between user code and Fast Fourier Transforms (FFT) accelerator APIs. FACC uses input-output equivalence of code sections and is considerably more robust than pattern-based techniques, which cannot match structural differences in FFT code. Additionally, there are also data mismatches, such as user-defined complex number types that differ from the representation required by the accelerator's API. FACC synthesizes wrapper code that handles the differences those differences and it is able to port legacy code to different FFT accelerators, showing meaningful speedup gains. This approach is similar to ATC [Martínez et al., 2023], which targets dense general matrix-matrix multiplication and convolution programs. Given a kernel implementation, ATC explores a combinatorially large

search space using a heuristics based on program classification, dynamic analysis, variable constraint generation, and lexical-distance matching to discover semantically equivalent target calls.

3.3 Neural Machine Translation

Since the advent of sequence to sequence models [Sutskever et al., 2014b], neural machine translation has been applied diverse coding techniques, ranging from code style detection [Pizzolotto and Inoue, 2021], generating accurate variable names [Lacomis et al., 2019], correcting syntax errors and bugs [Santos et al., 2018; Hong et al., 2021] code completion [Katz et al., 2019] to and specification synthesis [Mandal et al., 2023]. In this section we will focus on techniques that use NMT for programming language translation tasks at different abstraction levels.

3.3.1 Transpilation

Phrase-based statistical translation [Koehn et al., 2003, 2007] is a framework to machine translation that uses segmentation of text rather than individual words to perform translation. Based on this framework, [Karaivanov et al., 2014] uses standard SMT training and decoding techniques to learn phrase translation tables that map source code to target code. This technique trained three different NMT models and show a case study translating programs from C# to JAVA. These models are a direct phrase-based one, a model with a grammar-aware extension that ensures the generated code parses correctly, and a final model that combines grammar constraints with custom rules to improve quality. Each model was trained on a large corpus of C#-JAVA program pairs. Still on C# $\rightarrow$ JAVA transpilation, Xinyun et al. [Xinyun et al., 2018] proposed a tree-to-tree model that divides the translation problem in sub-trees and try to translate each one independently. Their model contains also an attention mechanism that guides the decoder when expanding target trees by identifying the corresponding source tree whenever the decoder expands one non-terminal.

PLNMT [Kim and Kim, 2019] applies NMT to produce OpenCL programs from CUDA kernels. This approach extracts paired CUDA and OpenCL API usage examples from existing software to build a training dataset and include a

pre-processing step before the training and inference phases. PLNMT is evaluated on benchmark suites such as Polybench-gpu [Grauer-Gray and Pouchet, 2012] , NVIDIA SDK [NVIDIA, 2025d], and Rodinia [Che et al., 2009], showing that the neural model can translate many application kernels correctly. BabelTower [Wen et al., 2022] goes in the opposite direction and uses a Transformer model [Vaswani et al., 2017] to auto-parallelize C programs by translating them to CUDA. It builds a large dataset of compute-intensive C and CUDA pairs and uses back-translation augmented with a discriminative reranker to learn the mapping from C to CUDA without paired examples, enabling the model to generate candidate translations in both directions. BabelTower also implements a discriminative model to score reranks the beam search outputs to enforce parallel semantic fidelity.

The work by Armengol-Estapé and O'Boyle [Armengol-Estapé and O'Boyle, 2021] replicates traditional compilation form C to x86 assembly using a self-trained Transformer. Although there are challenges such as the semantic gap between C constructs and machine instructions, variable instruction counts per source line, and the difficulty of modeling low-level control and data encoding, this study demonstrates that NMT models can learn non-trivial patterns between high-level source and low-level machine instructions.

More recently, EvoC2Rust [Wang et al., 2025b] leverages a LLM to translate and post-process C code to Rust, but it uses the model as a helper for idiomatic local rewrites, rather than an end-to-end translator. This approach first performs static analysis to extract a project-wide Rust skeleton that contains features such as module boundaries, type definitions, and function signatures. Next EvoC2Rust uses a Transformer-based model to translate individual C function bodies into Rust. The skeleton strictly guides the NMT output by pre-determining types and interfaces, while post-processing and Rust's compiler enforce correctness.

NMT-based transpilation has also been proposed in unsupervised training settings. TransCoder [Roziere et al., 2020] is a model trained on a corpus of source code mined GitHub that translate programs among JAVA, Python and C++. Instead of traditional dataset of example pairs, this work uses monolingual corpora from each language and presents a series of innovations to handle source code in language models. TransCoder presents bi-

directional Transformer models with shared embeddings and back-translation. Back-translations allows the model to learn both translation directions simultaneously and generate synthetic parallel data translations that are used to train itself. The model also incorporates syntax-aware pre-processing and sub-tokenization specifically designed for programming languages. This work was later extended to operate on compiler IRs rather than source code [Szafraniec et al., 2023]. This technique leverages the semantic structure generated by a compiler in IR code and feed into the neural model alongside source code, instead of treating code purely as token sequences. Architecturally, the system uses multi-modal encoders that combine token sequences and graph embeddings extracted from the IR modules. Another extension is the integration of TransCoder into a unit-test–driven framework to guide the translation process [Rozière et al., 2022]. In this work, the Transformer model is trained jointly with test–validation feedback that reinforces test-correct translations according and discards incorrect ones, allowing the model to refine itself iteratively.

3.3.2 Lifting

NMT has been applied to lifting-related translation tasks, in particular decompilation, lifting binary code into an IR or higher level language. Katz et al. [Katz et al., 2018] uses a standard recurrent neural network (RNN) trained on a pairs of C and assembly to lift code. This network treats assembly instructions and their operands as sequences of tokens. It uses attention mechanism to learn correspondences between assembly instructions and high-level C constructs and coding patterns, which enables the network to generate code that is both readable and idiomatic.

NeurDP [Cao et al., 2022] proposes a new hierarchical neural architecture to lift optimized binaries. The proposed architecture integrates instruction-level decoders with two types of reasoning: block-level context model to capture the relation between basic blocks and function-level reasoning to capture control-flow structure. This technique uses combined multiple predictors to guess different program features, such as control flow and opcode semantics, which improves translation and outperforms vanilla sequence-to-sequence models.

SLADE [Armengol-Estapé et al., 2024] is lightweight neural decompiler that expands the lifting task to handle highly optimized assembly across multiple ar-

chitectures. It uses a small transformer model and overcome size limitations with new program representations and decoding strategies. SLADE introduces a normalized canonical representation for optimized assembly that abstracts away certain register naming and scheduling differences to reduce vocabulary and pattern sparsity. It also implements contextual decoding with structural constraints to ensure that generated high-level code respects control flow and data dependencies. SLADE is more powerful than previous NMT-based binary lifters, lifting assembly code in two different instruction set architectures, x86 and ARM optimized with different levels of compile optimization back to equivalent C.

Despite the recent advances, studies still show that these models are prone to errors in their translation [Pan et al., 2024]. As a result, a number of error correction techniques have been developed. [Katz et al., 2019] augments the NMT translation with post-generation repair steps. Namely, this approach applies language grammar constraints to fix syntax-invalid code, static analysis to recover correct variable names and types, and semantic patching where the model outputs are compared against execution traces to correct behavioral inconsistencies. Coda [Fu et al., 2019] uses a different approach and integrates the repairs in the NMT learning loop. Model outputs that fail verification feed back into reinforcement or fine-tuning steps, improving future predictions. Coda fixes syntax errors using grammar-aware constraints and parser feedback, while symbolic execution is used to detect behavioral mismatches in relation to the original binary. Additionally, Coda applies type inference, signature recovery, and control-flow validation to ensure the generated code respects original calling conventions and program invariants.

In [Lee et al., 2023], a language model is combined with symbolic solvers to automatically translate between ARMv8 and RISC-V assembly. This technique uses a trained language model that outputs *alignment* and *confidence information* together with the lifted program. It uses low confidence on translated tokens to initiate an SMT-based sketching process to search for a better solution. Alignment is a mapping between sequence of instructions in the input and sequences of instructions on the output, since there is no one-to-one correspondence, and confidence information is the probability with which each token was predicted. It builds a sketch with the alignment sequences to build a sketch in which the tokens predicted with low probability are the initial holes.

This sketch is then passed as input to a solver that will fill the holes and find the correct translation. Another approach is to use NMT to learn repairs. Sequencer [Chen et al., 2019]is a sequence-to-sequence method that is proposed to fix bugs in JAVA programs. It trains a model on pairs of wrong and fixed code mined from version control histories to learn repairs purely from data rather than explicit rules.

NMT is also used to lift assembly code back to intermediate representations. A machine-learning based solutions is proposed in [Hasabnis and Sekar, 2016] to lift assembly code to a architecture-neutral intermediate language. The key idea is to leverage existing compiler backends—such as GCC or LLVM, which encode rich instruction semantics for many architectures in their code generation. The authors present an approach that automatically extracts instruction semantics from compilers and uses this extracted knowledge to drive the lifting of machine instructions to a generic IR, enabling support for several different ISA, like x86, ARM, and AVR. Forklift [Armengol-Estapé et al., 2024] extends this idea in a token-level encoder-decoder Transformer-based approach. It introduces an incremental learning strategy where the decoder is shared and frozen while new ISA encoders are fine-tuned, enabling scalable support for multiple architectures with minimal retraining effort. Forklift is a much more robust neural lifter that is able to lift optimized x86, ARM, and RISC-V assembly to LLVM IR across multiple optimization levels.

3.3.3 API Replacement

There has been some research effort aiming to perform neural API replacement on programs. [Phan et al., 2017] introduces JV2CS, a tool that uses Word2Vec [Mikolov et al., 2013] to model the APIs of Java JDK and C# into vector spaces and automatically learn pairs of mappings from one to the other. Then, JV2CS uses phrase-based translation to build tables mapping source API patterns to target API patterns, and uses a decoder to replacing sequences of API calls and control structures with statistically likely equivalents. This approach is flexible and can generalize beyond exact matches because it stores the statistical correspondence between tokens instead of a strict mapping.

DeepAM [Gu et al., 2017] also targets JAVA $\rightarrow$ C# and presents a the multimodal sequence-to-sequence learning architecture that learns joint semantic

representations of APIs in both languages to perform migration. The proposed model is an end-to-end encoder–decoder that embeds API call sequences together with context information. Said information consists of program state elements, such as parameter types and calling context, as well the API signature information taken from the documentation.

An LLM-based approach is proposed in [Almeida et al., 2024]. Here the authors use ChatGPT [OpenAI, 2022] to migrate software to use a newer version of SQLAlchemy [Bayer, 2012] , an Object Relational Mapping (ORM) Python library evaluating different prompting strategies and few-shot examples. This solution leverages the fact that LLMs are pretrained on large amounts of code and can inherently generalize API usage patterns and produce migration suggestions with minimal training data. APIRecX [Kang et al., 2021] proposes a neural network to recommend target API calls given a source code context and old library usage. It breaks the training task in two parts, first pre-training a GPT-based subword language model with data not containing calls to the target API, and second it fine-tunes the pre-trained model with examples that include the API to be recommended. Although these techniques can be generalized, they demand carefully designed prompts and examples or fine-tuning. Besides, they can present a lower performance in APIs which are not widely publicly available as training data.

3.4 Tensor Algebra Compilers

There exists a wide range of compilers for tensor algebra code generation. Most of them are based on einsum notation [Klaus et al., 2023; Raje et al., 2024; Dias et al., 2022] Tools like TACO [Kjolstad et al., 2017] enable the generation of fast sparse tensor kernels. In addition to generating high-performance CPU code, it has been extended to compile to GPUs [Senanayake et al., 2020], CGRAs [Hsu et al., 2022], compute-in-memory systems [Drebes et al., 2020], and distributed systems [Yadav et al., 2022]. Other extensions include Mosaic [Bansal et al., 2023], which extends TACO code generation to support other APIs specified by the users, and WACO [Won et al., 2023], which automatically optimizes the format and the schedule of a tensor program.

Several mainstream machine learning tools and compilers provide support for einsum notation via modules. Python frameworks like Pytorch [Imambi et al.,

2021], TensorFlow [Abadi et al., 2016], JAX [Bradbury et al., 2018], NumPy [Harris et al., 2020] and CuPy [Nishino and Loomis, 2017] have native support, allowing users to write einsum expressions directly and running them via an API call. The TVM compiler [Chen et al., 2018] also provides a module that supports natively written einsum programs. The MLIR compiler framework [Lattner et al., 2021] enables einsum-like contractions to be expressed via the Linalg dialect. Other programming languages such as Julia and R also have native einsum packages for tensor algebra programs.

Einsum notation is also used by Sigma [Zhao et al., 2023], which compiles einsum to dataflow architectures. High-level tensor operations expressed in einsum enable compilation to accelerators from GPUs [Weng et al., 2021] and StructTensor [Ghorbani et al., 2023] explores compiling tensors with different sparsity patterns. Einsum can also be compiled to other DSLs for optimization and use within their systems: work exists to compile einsum to SQL [Blacher et al., 2023]. Furthermore, einsum-like DSLs also provide an excellent basis for co-design [Jia et al., 2021, 2023], enabling hardware designers to obtain high-performance hardware for their tensor programs.

Linear algebra libraries like the Basic Linear Algebra Subroutines (BLAS) [Blackford et al., 2002], the Linear Algebra Pack (LAPACK) [Netlib, 2025b] and Eigen [Guennebaud et al., 2010] provide highly-optimized implementations of common dense matrix-based operations, such as general matrix multiplication and convolutions. NVIDIA provides a version of BLAS called cuBLAS [NVIDIA, 2025a] with implementations optimized for the GPU architecture. cuTensor [NVIDIA, 2025c] is another library provided by NVIDIA but for higher-order tensor operations on GPU, also limited to dense tensors.

There are a number of high-performance APIs that target sparse matrix operations. The Math Kernel Library (MKL) [Intel, 2025], provided by Intel, is a highly optimized library for high performance numerical computing on CPUs which supports many sparse linear algebra functions. The Python library sciPy [Virtanen et al., 2020] contains very efficient methods for sparse operations that use MKL kernels as underlying implementation. Similarly, the CUDA sparse matrix library cuSPARSE [NVIDIA, 2025b] provides GPU-accelerated basic linear algebra subroutines for handling sparse matrices, supporting a variety of sparse operations.

Table 3.1: Comparison of the techniques discussed in this literature review with in terms of extensibility, scalability, and correctness. The ✗ denotes the major weakness of each technique group.

	Extensibility	Scalability	Correctness
Rule-based	✗	✓	✓
Program Synthesis	✓	✗	✓
Neural Machine Translation	✓	✓	✗
This thesis	✓	✓	✓

3.5 Summary

This chapter presents a literature review of the research relevant to this thesis. We covered the main techniques used to automatically generate code, rule-based techniques, program synthesis and neural machine translation; and an overview of tools for highly-optimized tensor algebra. Although we shown several examples of successful applications, the techniques discussed have their own limitations when it comes to program lifting.

Rule-based approaches are have limited extensibility. They require retooling whenever the target API changes, which makes such approaches non-portable. Program synthesis is precise, but its enumerative modus operandi struggles to scale to large search spaces, often failing to generate complex programs within practical time bounds. Neural machine translation requires a large amount of training data which is not available for emerging DSLs. Additionally, languages models are prone to hallucinations which produces incorrect code.

This shows that there still a need to develop program lifting techniques which can overcome the issues of extensibility, scalability, and correctness. Table 3.1 compares the main approaches discussed in this chapter with the contributions of this thesis in terms of this criteria. The contributions presented in this thesis aim to overcome each of those issues. The following technical chapters (4, 5, 6) present novel lifting techniques that address those limitations with compiler technology to lift both dense and sparse tensor algebra programs to optimized backends.

.

Chapter 4

Lifting Tensor Code with Guided Enumerative Synthesis

As stated in Chapter 1, achieveing high performance for linear algebra is difficult with current compiler technology. This language/compiler failure has led to the growth of domain-specific languages (DSLs) aimed at efficient linear algebra. However, migrating legacy programs to these DSLs is costly and error-prone, which presents a serious barrier to existing applications to harness their performance. Furthermore, existing approaches such as API Migration, neural machine translation and simple program synthesis are brittle and do not scale well in face of complex real-world implementations.

This chapter presents C2TACO, a synthesis tool for synthesizing TACO [Kjolstad et al., 2017], a well-known tensor DSL, from C code. We propose a guided enumerative synthesis method to generate TACO programs based on automatically generated IO examples. We use source code analysis to retrieve features from the original programs and use them as search aids during synthesis.

We compared the performance of C2TACO against a neural machine translation approach and two state-of-the-art existing schemes, TF-Coder [Shi et al., 2022] and ChatGPT [OpenAI, 2022]. When evaluated on a suite of tensor benchmarks, C2TACO is able to synthesize 95% of the programs, demonstrating considerably higher accuracy than the other techniques (10%, 32% and 24% respectively). Our lifted TACO programs are portable and achieve significant performance improvements over the original implementation when evaluated on a multi-core (geomean 1.79x) and GPU (geomean 24.1x) platform.

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

Figure 4.1: C implementation of matrix vector product and summation.

We start by presenting a motivating example in section 4.1. We then summarize the pipeline of C2TACO in section 4.2 and describe the key components in sections 4.3 and 4.4. Finally, we present the setup used in evaluation in section 4.5, followed by experimental results in section 4.6 and a conclusion 4.7.

4.1 Overview

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

4.1.1 TACO

TACO [Kjolstad et al., 2017] is a high-level programming language for tensor contractions. A tensor is a generalization of a matrix (order 2) to higher orders. The core TACO language is based on einsum notation, and it supports tensor expressions of unbounded length and tensors of unbounded order. TACO has been used in other frameworks including TVM [Chen et al., 2018].

Consider the matrix-vector product and summation example: $a_i = \Sigma_k X_{i,k} b_k + c_i . \forall i$. In C a simple sequential implementation would result in the code shown in Figure 4.1. While straightforward, targeting this code for different platforms such as multi-cores or GPUs would require significant code restructuring. Writing the example in TACO gives:

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

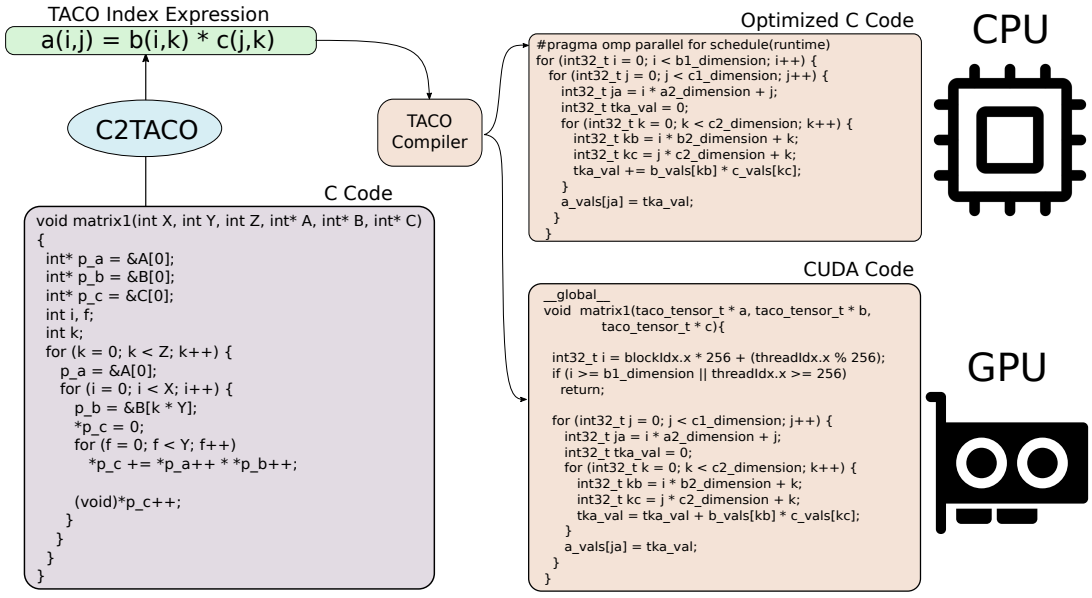


Figure 4.2: Lifting C code to TACO using C2TACO. Given a program implemented in C, C2TACO generates a equivalent program written in TACO tensor index notation which the TACO compiler can use to produce high-performance code targeting a variety of hardware platforms.

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

4.1.2 Example

We take existing legacy C code, lift it to TACO, and then use TACO's code generation abilities to target diverse, high-performance platforms. Consider the program in Figure 4.2. This is a C function from the `DSPStone` benchmark suite [Zivojnovic, 1994] which makes use of post-increment pointer arithmetic to target the addressing modes found in DSP processors. Although the pointers are a hindrance to understanding, this program is in fact matrix multiplication.

C2TACO uses automatically-generated input/output examples as a specification for an enumerative synthesis algorithm. C2TACO uses information about the C program to guide a search through the TACO grammar in a type-directed

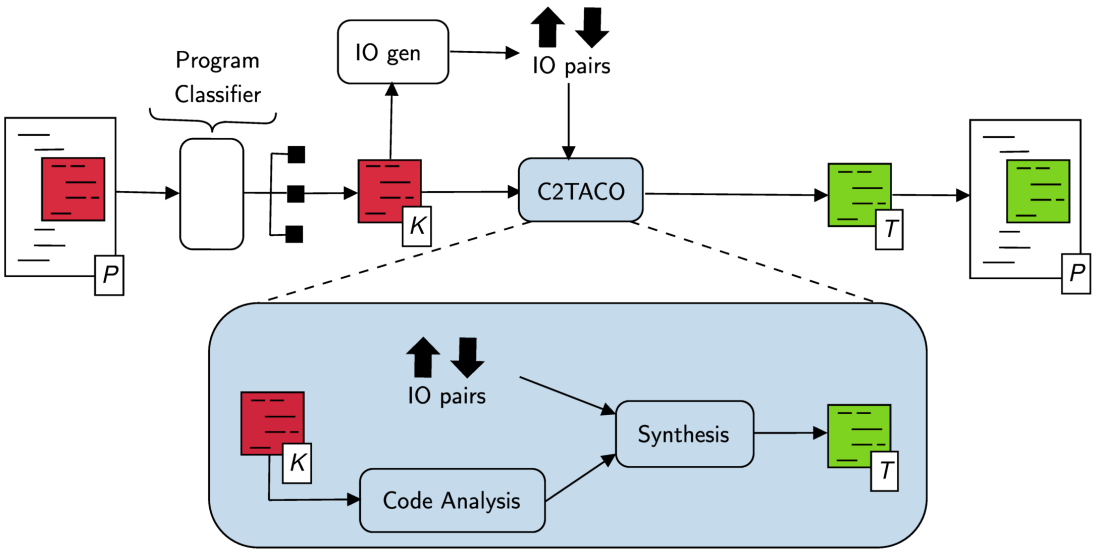


Figure 4.3: Architecture of C2TACO.

template-based enumerative fashion and produces the TACO code shown in Figure 4.2. As well as being higher-level and easier to read than the original C code, the synthesized TACO program can be optimized and targeted at different platforms.

Figure 4.2 shows the code generated from tensor index notation for a multi-core CPU and an NVIDIA GPU. For the CPU, the TACO compiler generates OpenMP code with a dynamic runtime schedule policy. So in effect, lifting is an automatic parallelization method for certain C programs. For the NVIDIA GPU, the TACO compiler generates CUDA code (also shown in Figure 4.2). Although, the code is syntactically distinct from the OpenMP version, the TACO compiler again exploits parallelism with implicit concurrence across all of the threads executing the shown kernel.

4.2 C2TACO Architecture

Figure 4.3 shows our overall approach. Given a program P written in C, we first detect the program sections K that are suitable for lifting using neural program classification. Once we have extracted the candidate regions, we generate input-output (IO) examples which are then used as a specification for our synthesis scheme. Our system performs a series of static code analysis to extract relevant features from K. We then search the TACO grammar for equivalent

programs that satisfy the IO specification using the features of K to prune the program space. Once we have identified a suitable equivalent TACO program T, we lower it to the target platform and insert it into the original program for execution.

4.2.1 Classification

We take as input general-purpose programs that perform varied computations and perform lifting to a domain-specific language for tensor contractions. Because we cannot express general computation in TACO, there is a need to identify the code regions that can be lifted and accelerated. We use prior work in neural program classification [Woodruff et al., 2022] to determine which parts of the program represent tensor operations. In case of false positives, lifting will fail because no equivalent program exists in TACO. There are two possible failing scenarios in face of misclassifications. First, the code analysis may produce features that overly constrain the search space, leading to exhaustion the search space without finding a solution. Alternatively, the search space may not be sufficiently restricted, which results in a timeout before an equivalent program is generated.

4.2.2 IO Generation

Our synthesizer is driven by a specification of observational equivalence (i.e., randomly generated input-output examples). We generate 10 input-output examples. Whilst this means that we cannot *guarantee* absolute equivalence of the synthesized and source code, it allows our synthesis to scale to programs too complex to be reasoned about by the SMT solvers that drive other lifting techniques [Bhatia et al., 2023].

4.2.3 Lifting via Synthesis

Once we have the IO examples of the code to lift, we explore the space of TACO programs using enumeration of templates over TACO's grammar to generate programs that may be equivalent to the original C program. We execute each candidate on the IO samples to see if it is equivalent. The Enumerative Template Synthesis algorithm is described in Section 4.3. Given the unbounded

size of the TACO program space, this can lead to excessive synthesis time. We, therefore, introduce a compiler tool that extracts a set of features from the original C program and use it to guide search, as described in Section 4.4.

4.2.4 Validity

Our synthesized TACO programs are demonstrated to have observational equivalence with the original programs in C. We generate IO pairs executing the original program with randomly generated inputs and test whether the TACO program produces the same output. This generation process is explained in detail on section 4.3.2. We also manually inspect the synthesized code.

4.2.5 Exporting

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

4.3 Enumerative Template Synthesis

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

We use a bottom-up enumerative synthesis algorithm to enumerate *template* TACO programs, i.e., TACO programs that use symbolic variables in place of all tensors and constants. We then check whether there is a valid substitution of inputs and constant literals for these symbolic variables that satisfies the specification. The enumeration of our algorithm is based on classic algorithms in the literature [Albarghouthi et al., 2013; Udupa et al., 2013], while the use of a sub-procedure to instantiate concrete variable names and constant literals is based on CEGIS(T) [Abate et al., 2018].

⟨*PROGRAM*⟩ ::= ⟨*TENSOR*⟩ **=** ⟨*EXPR*⟩

⟨*TENSOR*⟩ ::= ⟨*ID*⟩ **(** ⟨*INDEX-EXPR*⟩ **)** | ⟨*ID*⟩

⟨*INDEX-EXPR*⟩ ::= ⟨*INDEX-VAR*⟩
| ⟨*INDEX-VAR*⟩**,** ⟨*INDEX-EXPR*⟩

⟨*INDEX-VAR*⟩ ::= i | j | k | l

⟨*EXPR*⟩ ::= ⟨*EXPR*⟩ **+** ⟨*EXPR*⟩
| ⟨*EXPR*⟩ **-** ⟨*EXPR*⟩
| ⟨*EXPR*⟩ ***** ⟨*EXPR*⟩
| ⟨*EXPR*⟩ */* ⟨*EXPR*⟩
| ⟨*CONSTANT*⟩
| ⟨*TENSOR*⟩

⟨*ID*⟩ ::= T_0 | T_1 | T_2 | ...

⟨*CONSTANT*⟩ ::= C_0 | C_1 | C_2 | ...

Figure 4.4: TACO grammar.

4.3.1 The Grammar

Our synthesis algorithm requires a grammar as definition of the TACO language to enumerate programs. We enumerate through the grammar G, shown in Figure 4.4, which defines a search space of possible template TACO programs. The grammar G is defined as a set of non-terminal symbols NT, terminal symbols, and production rules R. For each rule $r \in R$, $|NT|$ indicates the number of non-terminal symbols in the rule. We refer to the non-terminal symbols on the right-hand side of a rule in the order they appear as $NT_0, NT_1, \ldots$. For example, for the production rule ⟨*PROGRAM*⟩ ::= ⟨*TENSOR*⟩ $=$ ⟨*EXPR*⟩, the non-terminals are $NT_0 =$ ⟨*TENSOR*⟩ and $NT_1 =$ ⟨*EXPR*⟩, and $|NT| = 2$.

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

In addition to the grammatical structure define by G, C2TACO assumes other restrictions in the original programs, including the absence of aliases, undefined behavior, and affine indexing expressions for tensor access.

4.3.2 Specification

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

We extend the method set out in FACC [Woodruff et al., 2022], where inputs are randomly generated according to manually given constraints dictating the length of arrays and favoring smaller values to make evaluation faster. We constrain arrays to be of size 4096, and fix tensor-dimensions to be equal (e.g., a 2-dimensional tensor is of size 64×64).

A single input-output example I, O consists of a set of randomly generated arguments $I = (i_1, \ldots i_m)$, corresponding to the input parameters $\vec{x} = (x_1, \ldots, x_m)$, and an output $O = P_C(i_1, \ldots, i_m)$, . We generate 10 input-output examples which form a specification: $\phi_{IO} = \{(I, O)_1, \ldots, (I, O)_{10}\}$

A program P_T satisfies the specification ϕ_{IO} iff $\forall (I, O) \in \phi_{IO}.P_T(I) = O$. To determine this in practice, we run P_T using the TACO Python API, checking if the behavior matches the corresponding outputs.

4.3.3 Template Enumeration

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

We enumerate templates as shown in Algorithm 4, by iterating through production rules until we have found all possible complete templates of length 1 in the grammar. We then increase the length and repeat the process, using the previously enumerated templates as building blocks, until we have hit the maximum user-given length. Each time the length increases, we add a new

tensor ID and a new symbolic constant to the set of candidate templates. This is shown in Algorithm 3.

We discard any invalid candidates during enumeration, i.e., templates that do not type check or are unsupported by TACO. More specifically we discard:

- any candidate that iterates over two different dimensions with the same index variable (e.g., $T_0(i, i)$);
- any candidate where the same tensor appears more than once in a program with different orders (e.g.: $T_0(i) = T_1(i) * T_1(i, j)$); and
- any candidate where the same tensor appears on both sides of an assignment (e.g.: $T_0(i, j) = T_0(i, j) + T_1(j, k)$).

4.3.4 Instantiating Templates

After we have generated all templates of length L, we check whether any of these templates generate programs that satisfy the specification, ϕ_{IO} (see Section 4.3.2). To do this, we enumerate through all substitutions that map all symbolic constants in the candidate program to concrete values, and all tensor IDs to inputs in the specification, until we find a substitution that gives us a TACO program that satisfies the specification. This is shown in Algorithm 4. We limit the concrete constant values to constants present in the source program.

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

4.4 Synthesis Guided by Static Code Analysis

The search space of possible TACO templates is large, and so, in C2TACO, we use program analysis to focus the scope of the synthesis search, prioritizing

Algorithm 3: Enumerative Template Synthesis. The subprocedures *instantiate* and *completeRule* are shown in Algorithm 4.

input : source code P_C, grammar G, max length L
output : candidate program, or no solution

```
1  Algorithm synthesize(P_C, G, L)
2      short ← ∅;                          // set of short candidates
3      long ← ∅;                           // set of long candidates
4      φ_IO ← generateSpec(P_C);
5      for l in 1...L do
6          short ← short ∪ newTensor() ∪ newCons();
7          while true do
8              nS ← ∅ ;                    // new short candidates
9              nL ← ∅ ;                    // new long candidates
10             for Rule ∈ G do
11                 for p ∈ completeRule(short, Rule) do
12                     if Length(p) = L ∧ valid(p) then
13                         nL ← nL ∪ p;
14                     else if Length(p) < L ∧ valid(p) then
15                         nS ← nS ∪ p ;
16                 end
17             end
18             if nS ⊆ short ∧ nL ⊆ long then break;
19             if l = L then
20                 long ← long ∪ nL;
21             else
22                 short ← short ∪ nS ∪ nL ;
23             end
24         end
25     end
26     for p ∈ long, short do
27         P_T, result ← instantiate(p, φ_IO);
28         if result then return P_T ;
29     end
30     return no solution
```

Algorithm 4: Subprocedures. Note $e.\{x \mapsto y\}$ denotes the result of the proper substitution of the expression x by the expression y in the expression e.

```
1  Procedure completeRule(short, Rule)
2      completions ← ∅;
3      for p ∈ short do
4          candidate ← Rule.{NT_0 ↦ p};
5          if |NT| ∈ Rule = 2 then
6              for q ∈ short do
7                  candidate ← Rule.{NT_1 ↦ q};
8          completions ← completions ∪ candidate;
9      return completions
10 Procedure instantiate(p, φ_IO, P_C)
11     X ← getInputParams(P_C);
12     K ← getConstants(P_C);
13     T ← getTensors(p);
14     C ← getConstantSymbols(p);
15     for x, t ∈ cartesianProduct(X, T) do
16         for k, c ∈ cartesianProduct(K, C) do
17             result ← check(p.{t ↦ x}.{c ↦ k}, φ_IO)
18             if result then
19             return result, p.{t ↦ x}.{c ↦ k} ;
```

candidates that are more likely to be correct. In particular, we use heuristics to estimate the correct TACO template length (section 4.4.1), the correct dimensions of each tensor (section 4.4.2) and the arithmetic operators likely to be in the TACO program (section 4.4.3).

4.4.1 TACO Program Length

The length of a TACO program is related to the number of array/pointer references and constants in the original C code. However, temporary variables to capture common sub-expressions and mutable arrays mean that there is no

direct correspondence. Fixing the size of the target TACO program reduces the search space because we only have to enumerate candidates once.

To determine the range of sizes C2TACO explores, we focus on the definition of the *output* array and examine the number of input arrays, or *uses* [Cooper and Torczon, 2011]. At each definition, we iteratively build a set of variables used by that definition. We use reaching analysis to disambiguate between different references to the same (mutable) variables. We then reduce the constructed set in the presence of summations or reductions. In C, when writing a reduction or summation, a variable appears on both sides of an assignment but only once in the TACO program. For this reason, we apply simple data dependence analysis to check if there is a recurrence. If there is, we do not count it twice.

For example, in Figure 4.1, we have the use set `sum, X, b, c` for the the output array `a`. This is reduced to `X, b, c` after detecting the reduction on `sum` to give 4 (`a, X, b, c`) as the predicted number of tensors in the TACO program.

4.4.2 Tensor Dimensions

C programs frequently contain linearized arrays: where a single pointer is used to represent a multi-dimensional tensor. However, in TACO dimensions are explicit, and searching over all possible dimensions is costly. To address this, we apply the dataflow analysis defined in [Franke and O'Boyle, 2003] to recover arrays from pointer structures, and then apply delinearization [O'Boyle and Knijnenburg, 2002] and determine the highest dimension array.

As an example, consider the program in Figure 4.2. After applying dataflow analysis to `*p_c`, we get `p_c`$[Z * k + i]$. Let n be the dimensionality of the recovered C array and m be the dimension of the enclosing loop nest J. Here $n = 1, m = 3$. The loop iterators are represented by a column vector $J = [k, i, f]^T$, and UJ is the affine expression for the array access $[Z * k + i]$:

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

We now delinearize by constructing a transformation S, such that SU gives a matrix of 1's and 0's. For our example $S = [()/Z, ()\%Z]$. For details of this step,

we refer the reader to the paper [O'Boyle and Knijnenburg, 2002]. We apply the transformation to give:

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

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

4.4.3 Operator Analysis

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

4.5 Experimental Setup

This section describes in detail the experimental setup used in C2TACO's evaluation, including environment (section 4.5.1), competitors (section 4.5.2), and methodology (section 4.5.3).

4.5.1 Environment

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

The real-world benchmarks originate from different applications. We selected a subset of the programs used by previous synthesis work [Collie et al., 2020b]. Our benchmark suite includes `blas`, a baseline implementation of functions from the BLAS [Blackford et al., 2002] linear algebra library; `DSP`,

which consists of signal processing functions adapted from the TI [Instruments, 2023] library; `makespeare`, programs that manipulate arrays of integers originally from [Rosin, 2019]; `mathfu`, mathematical functions from the MathFu [Google, 2023b] library; and `simpl_array`, a set of programs performing computation on arrays of integers, originally from [So and Oh, 2017].

In addition to those, we extracted benchmarks from other suites that contain tensor manipulations, specifically neural network operations from the `Darknet` deep learning framework, as well as `DSPStone` and `UTDSP` kernels targeting digital signal architectures from the DSPStone [Zivojnovic, 1994] and UTDSP [Saghir, 1998] suites.

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

Software: ETS and C2TACO are implemented in Python version 3.8.10. The NMT Transformer model is implemented using Fairseq [Ott et al., 2019] 0-12.2 with Google's SentencePiece [Kudo and Richardson, 2018] as the tokenizer. The analyses described on Section 4.4 are implemented as plugins for the `clang` compiler version 14.0.0. Operating system is Ubuntu 20.04.6 LTS.

Hardware. We evaluate C2TACO on a multi-core CPU and GPU platform. The targeted CPU is an 8-core Intel i5-1135G7 at 2.40GHz with 16 GB of RAM (LPDDR4) at 4267 MT/s. The GPU is an NVidia GeForce GTX 1080 Ti using driver version 535.54.03 and CUDA runtime version 12.2.

4.5.2 Competitive Approaches

To evaluate C2TACO we compared its performance against other techniques. We implemented a simple version of the synthesis process described in Section 4.3 and an alternative approach based on neural machine translation. In addition, we consider an existing large language model ChatGPT and IO-based synthesizer, TF-Coder.

4.5.2.1 ETS

C2TACO uses the synthesis algorithm described in Section 4.3 combined with the heuristics described in Section 4.4. To evaluate the contribution of the

heuristics in C2TACO, we compare to the most basic enumerative template synthesis algorithm described in Section 4.3 (without any heuristics), which we refer to as ETS.

4.5.2.2 Neural Machine Translation

NMT converts text sequences from one language to another by means of a deep neural network and has shown positive results on code tasks. We therefore frame the task of lifting C to TACO as a neural machine translation problem. More formally, given a data set D with N pairs of programs (P_C^i, P_T^i), where P_C^i is a program in C, and P_T^i is a semantically equivalent program written in TACO, We train a Transformer [Vaswani et al., 2017] that given a C input sequence minimizes the edit distance between the predicted and ground-truth TACO. Once trained, then, given an unseen C program, the model will generate the most likely equivalent TACO program.

The main challenge for any new DSL is the availability of training data. To overcome this, we generate a synthetic dataset based on the TACO grammar shown in Figure 4.4. We compile the synthetically generated TACO programs to generate the equivalent C programs. We limit our synthetic dataset to programs that contain a maximum of 5 tensors of no more than 4 dimensions, and where all datatypes are integers.

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

We generate 800K pairs of C program and TACO expressions. The bottom-up enumeration ensures that the generated programs are are syntactically distinct. We further discard semantically equivalent programs to ensure that each data sample is unique within the datasets. Of the 800K pairs, we randomly separated 5K for validation, 5K for testing, and the remaining for training. The trained model is a Transformer with 6 encoders and 6 decoders with 16 attention heads and an embedding size fixed at 1024. The model was trained using Adam optimizer [Kingma and Ba, 2017] with a learning rate of $1e$-3.

4.5.2.3 TF-Coder

TF-Coder [Shi et al., 2022] is an open-source publicly available program synthesizer. It takes a single input-output example as source and generates a corresponding TensorFlow program. Although the search space of TF-Coder is not defined by the same grammar we considered in our synthesis methods, we compare C2TACO against TF-Coder because both synthesize programs from IO examples and operate on the domain of tensor computations. We use one of the IO examples automatically generated by our synthesis scheme, but limit it to less than 100 elements as required by TF-Coder.

4.5.2.4 ChatGPT

ChatGPT [OpenAI, 2022] is large-scale language model based on GPT 3.5. It has been used for a wide number of tasks including code generation. We used version 3.5 in our experiments. As its accuracy depends on the quality of its prompts, we experimented with various formats and found the following to be the most effective, followed by the original source code:

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

4.5.3 Methodology

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

4.6 Evaluation

In this section, we evaluate against four criteria: coverage (Section 4.6.1), error rate (Section 4.6.2), synthesis time (Section 4.6.3), and speedup (Section

Table 4.1: Synthesis coverage of different approaches on the artificial dataset.

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

4.6.4).

4.6.1 Coverage

Figure 4.5 shows the lifting coverage of each of the five schemes described in section 4.5.2 across the two benchmark suites: artificial and real-world.

Artificial. As described in section 4.5.1, these are C kernels generated by the TACO compiler guaranteed to have an equivalent in the TACO language. The coverage of each scheme is shown in Table 4.1 and Figure 4.5.

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

NMT achieves higher accuracy, translating seven of the benchmarks. The Transformer model was trained using TACO-generated kernels, which have a similar structure to the synthetic programs. Unlike synthesis methods, it always

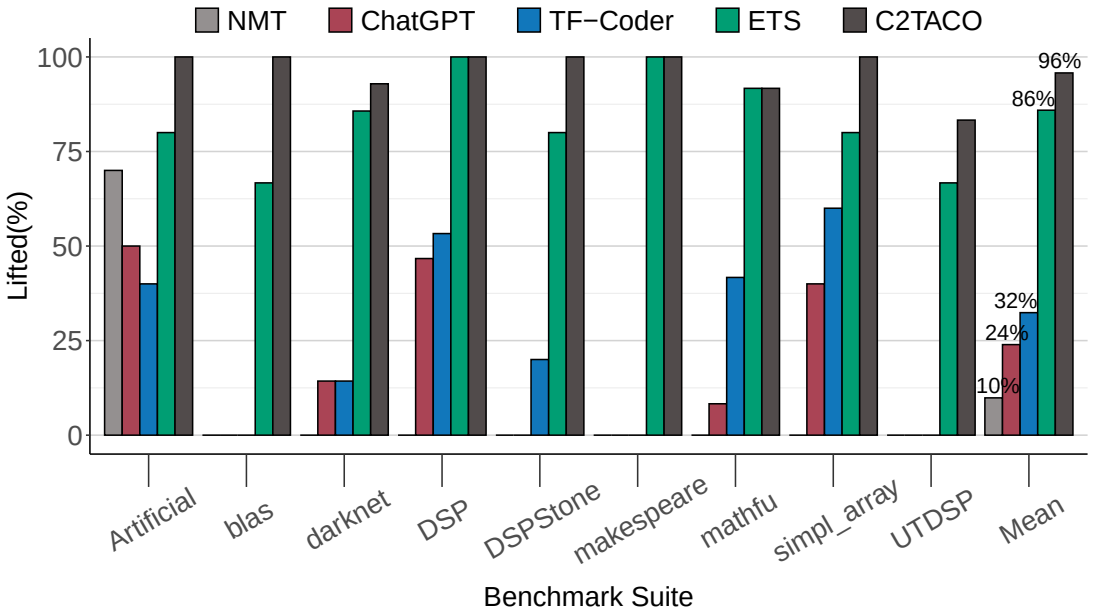


Figure 4.5: Overall lifting coverage across benchmarks suites.

produces a result even though it may be inaccurate and does not timeout. In the three cases where NMT fails, it correctly guesses the number of tensors but misorders them in the resulting programs.

ChatGPT is able to correctly predict 5 of the 10 benchmarks, hallucinating the remainder. In four cases it produces syntactic invalid programs. The syntax errors include wrong indices, multiple assignments, and duplication. In one case a tensor was treated as having different orders in the same program. In another, ChatGPT produces a program that is syntactically valid but incorrectly refers to the same tensor twice.

Real-World. Real-world benchmarks are more challenging as shown in Figure 4.5. Both ETS and C2TACO are able to achieve high coverage of 85% and 95% respectively. ETS times out on 5 out of 61 while the sole instance of failure for C2TACO is the presence of program features not contemplated in our implementation of the grammar 4.4. TF-Coder manages to correctly synthesize 31% of the benchmarks. Along with timeouts, TF-Coder also produces programs that are semantically incorrect. We further discuss these in Section 4.6.2.

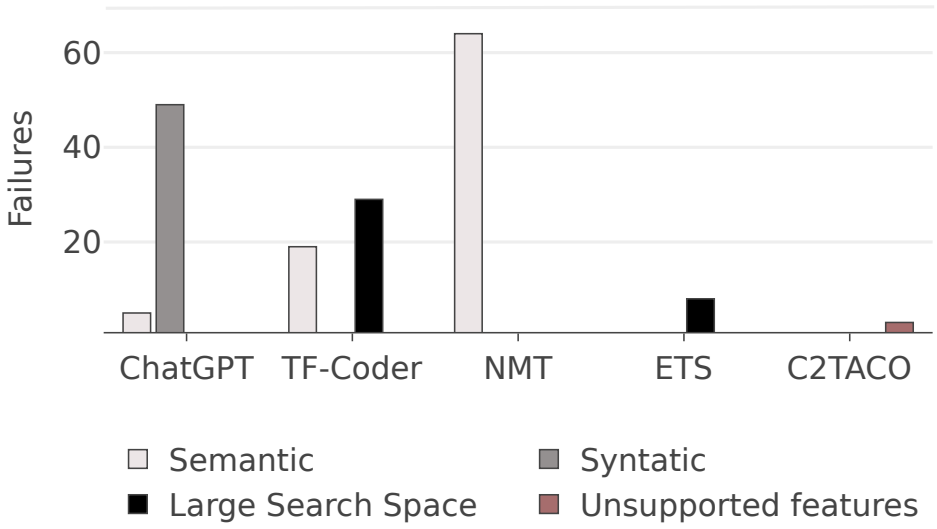


Figure 4.6: Distribution of failure causes for the different approaches evaluated.

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

4.6.2 Error Analysis

We identify several different reasons for failure: a large search space causing time out; syntactic and semantically wrong solutions. Figure 4.6 depicts a summary.

Large Search Space. Enumerative synthesis techniques explore a large search space, which grows as program length increases. This causes 60.42% of TF-Coder's failures and all failures for ETS. Neural translation approaches, ChatGPT and NMT, always find a solution in time due as they translate a program in

a sequence-to-sequence fashion and do not perform an extensive search. Although C2TACO is also based on enumeration, it never times out as program analysis restricts the search space sufficiently.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

$$tf.add(arr, arr)$$

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

4.6.3 Generation Time

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

Despite performing a search, TF-Coder is fast, taking an average of 1.18 seconds to find the solution. Nevertheless, TF-Coder is only able to correctly lift 40% of the artificial benchmarks (Section 4.6.1). ETS is the slowest method

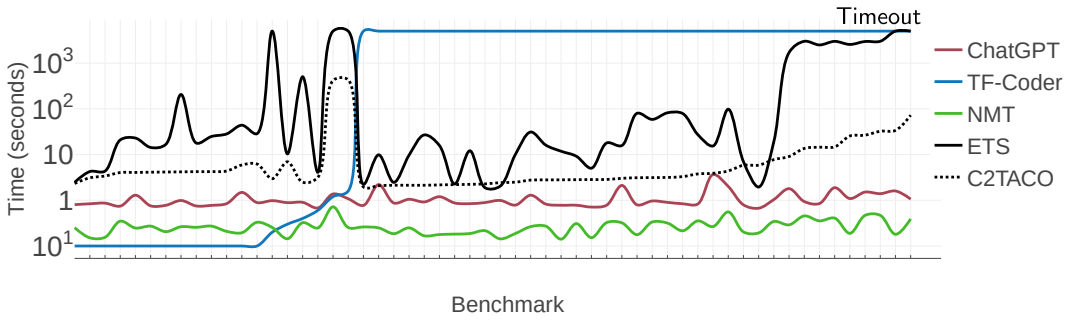


Figure 4.7: Lifting time on real-world benchmarks. Y-axis is on logarithmic scale.

with an average of 238 seconds to find the solution. In contrast, C2TACO takes an average of 21 seconds. That result shows the impacts of the program features obtained by syntactic analyses in guiding the synthesizer to find the correct answer.

Real-World. Figure 4.7 shows the synthesis times for each of the five approaches across the real-world collection. Numbers are on a logarithmic scale. As expected both the neural approaches NMT and ChatGPT are fast and stable across all programs. NMT always returns a program in less than 1 second and ChatGPT takes a maximum of 4 seconds to find a solution. However, as shown in Section 4.6.1, this speed comes at the expense of frequently generating wrong code.

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

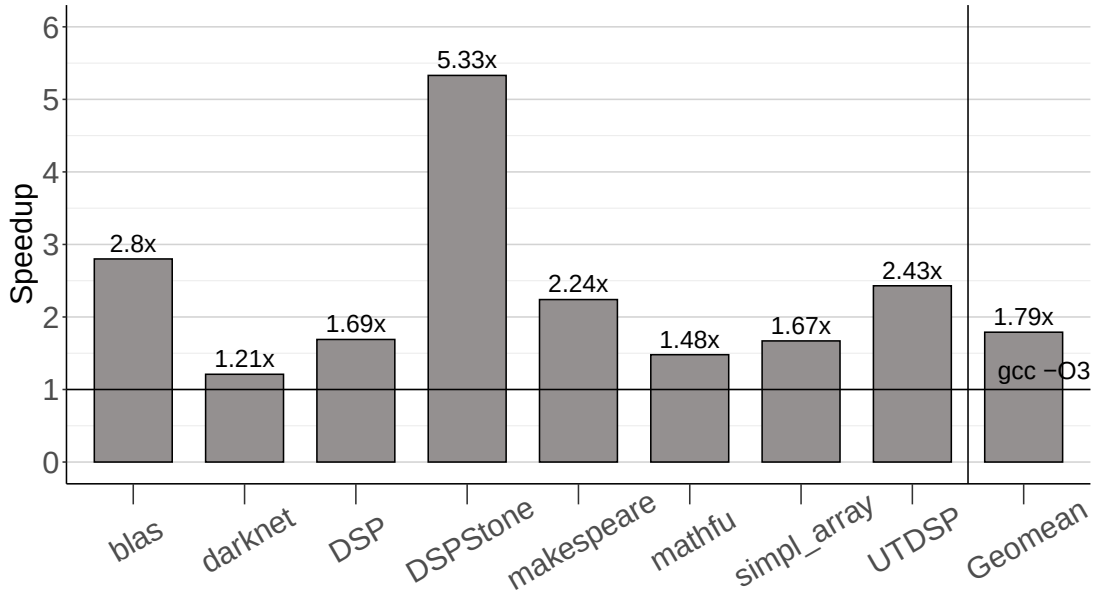
whole suite. C2TACO is also faster than fully manual development, as reasoning about Einsum notation in structurally complex programs is time-consuming for humans, whereas C2TACO automates this task with code analysis.

4.6.4 Performance of Lifted Code

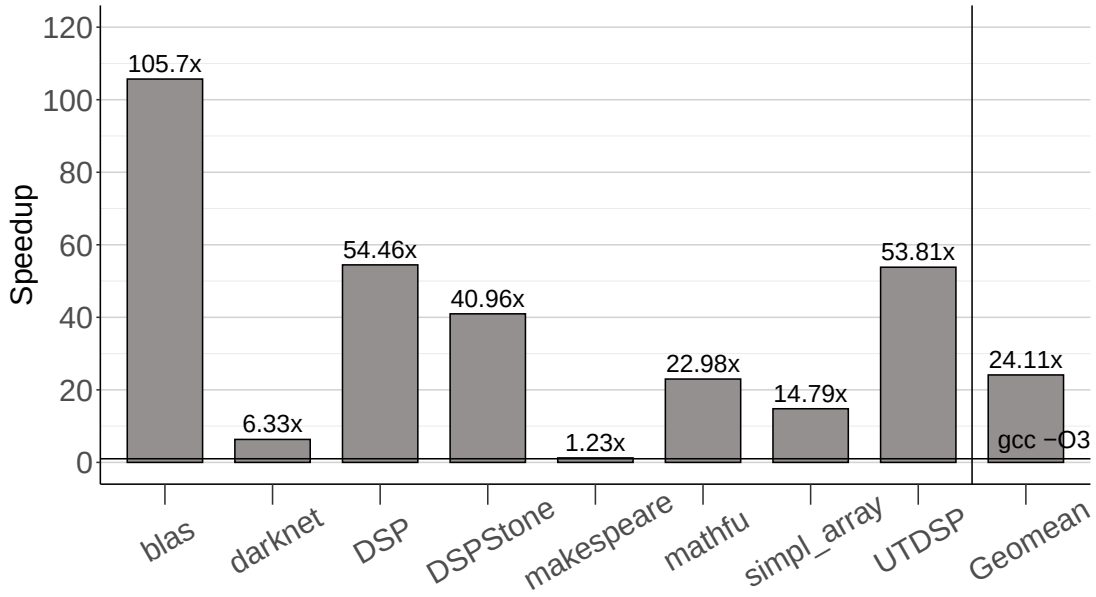
The main reason we wish to lift code to TACO is to exploit its portable performance. We generated C and CUDA versions of the programs generated by C2TACO and measured their performance on a multi-core CPU and GPU respectively. Figure 5.13 shows the speedup across the benchmark suite achieved running lifted programs. Baseline is the original implementation compiled with `gcc -O3`. Only the real-world benchmarks are considered as the artificial ones are directly derived from the TACO compiler and the synthesized version corresponds to the original.

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

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



(a) CPU



(b) GPU

Figure 4.8: Speedup obtained by the synthesized TACO programs on different hardware platforms. The baseline is the average running time of the original implementations when compiled with `gcc -O3`.

Table 4.2: Speedup obtained given different tensor dominant orders. We consider the highest order among the tensors in a program as dominant.

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

Speedup by Program Complexity. We further evaluated the impact of lifting on the performance of programs when such programs become more complex. In our domain, we consider programs more complex as they manipulate tensors with higher orders. We define the concept of dominant order as the highest order among the tensors in a program. For example, the program shown in Figure 4.1, manipulates tensors of 3 different orders: vectors (order 1), a matrix (order 2) and a scalar variables (order 0). The dominant order for that program is therefore 2.

Table 4.2 shows the overall speedup obtained on programs with different dominant orders. We observed two categories of dominant orders in the real-world benchmarks, 1 and 2. Programs that handle two-dimensional tensors benefit more from being lifted than the ones operating on one-dimensional ones. The speedup goes from 1.41x to 3.20x on the multi-core and from 20.19x to 36.97x on the GPU. These results show that the impact of lifting is even higher for programs that are more complex in the sense that they manipulate multi-dimensional tensors.

4.6.5 Summary

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

4.7 Conclusions

This chapter presented a solution for the lifting problem in the context of dense tensor algebra. We develop C2TACO, a synthesis tool for lifting C tensor code to TACO. C2TACO uses equivalence behavior and program analysis to generate code and it is shown to lift more programs in a shorter time with greater accuracy when compared to an alternative NMT and simpler synthesis approaches. C2TACO also outperforms existing techniques, lifting 95% of the benchmarks.

We demonstrate that the synthesis of equivalent TACO programs is feasible for a range of C programs taken from software libraries and benchmark suites. We also show that we can obtain significant performance improvement over the original source. Using C2TACO we are able to synthesize TACO programs that are 1.79x faster when ported to a multi-core CPU and 24.1x when ported to a GPU platform.

Chapter 5

Guess, Measure & Edit: Using Lowering to Lift

Current lifting techniques use language models or program synthesis to translate code. As shown in the previous chapter, language models are prone to hallucinations and, while accurate, program synthesis approaches do not scale to complex tensor DSLs. Although successful, C2TACO and other synthesis methods cannot scale because of the fundamental limits of bottom-up enumerative synthesis. In practice, these approaches cannot synthesize programs with tensors of greater than 2 dimensions.

This chapter 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 4.07x and 38.30x when ported to a multi-core CPU and GPU respectively.

We introduce the Guess, Measure & Edit framework in section 5.1. Next, we explain in detail how we apply this framework to lift tensor code and develop KONRUL in section 5.2. Section 5.3 describes the experimental setup used, followed by KONRUL's evaluation in section 5.4. We present the conclusions in section 5.5.

5.1 Guess, Measure & Edit

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 infrastructure 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.

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 $\forall x.IR_P(x) = IR_E(x)$. The existence of such lowering mechanism is underlying for our framework to measure the distance between lifting candidates and the original program. 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.

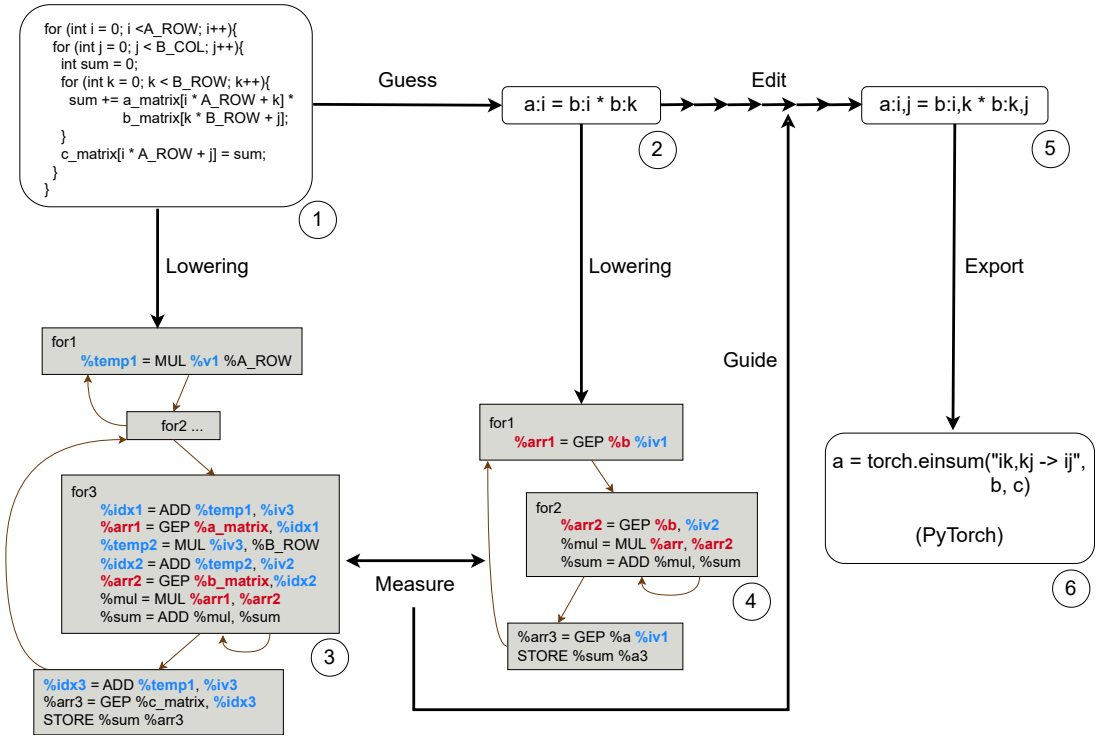


Figure 5.1: KONRUL takes as input a C program ① and tries to guess an equivalent einsum program ②. It lowers both to LLVM IR ③, ④ and checks similarity between them. KONRUL uses the result of this check to iteratively edit the guess until it finds an einsum program that is equivalent to the input ⑤.

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 5.1 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.

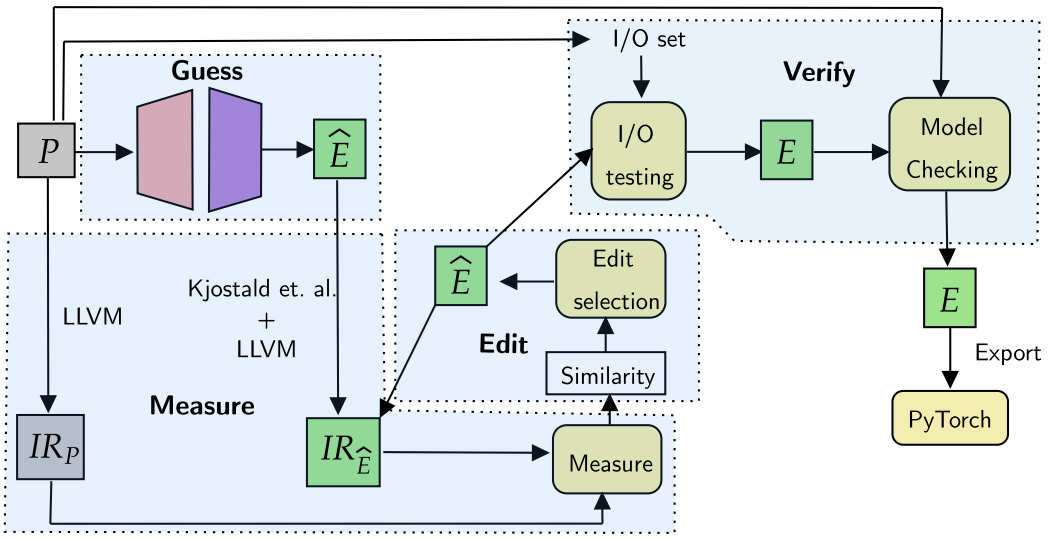


Figure 5.2: Overall architecture of KONRUL. Kernel extraction and IO generation are based on prior-work.

- 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 edit is applied to the guess to generate a new einsum candidate which is again lowered and measured against the original program. This process is repeated to eventually produce the correct lifted einsum notation program shown in ⑤.

- This einsum program is then executed with IO examples and checked for equivalence. If successful, it is then verified by model checking before being exported into PyTorch as shown in ⑥.

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

Figure 5.3: 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.

5.2 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 5.2 and the search process described in Algorithm 5. KONRUL uses a Transformer model [Vaswani et al., 2017] to *guess* a solution, compiler lowering to infer a specification for the guess (section 5.2.1), program similarity metrics to *measure* the difference between the specifications (section 5.2.2) and guide a search of edit rules (section 5.2.3), and finally testing and model checking to verify the solution (section 5.2.4).

Target language. We use extended einsum notation described in section 2.4.3 as our target language. The grammar we support, G, is depicted in Figure 5.3, and our aim is to find a solution E that is in the language of G. This grammar is fully represents all programs that can be expressed in extended einsum notation. Once we have a valid solution, it can then be exported to any DSL that supports einsum notation, e.g., PyTorch [Imambi et al., 2021].

Algorithm 5: KONRUL search and edit algorithm. Procedure $Edit$ is described in Algorithm 6.

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*$)$;
  end
  for $c \in$ *candidates* **do**
    if *Check*$(c, \phi_{IO}, P) \wedge$ *Verify*(c, P) **then**
      return *c*
  end
  return *no solution*

Algorithm 6: 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 $\hat{E}$

5.2.1 Language Model: Guess

The first phase of KONRUL's pipeline uses a standard Transformer to translate the input C code into an einsum program. As training data for DSLs is scarce, we use a compiler [Kjolstad et al., 2017] to generate C code from einsum programs which have been selected by searching the einsum grammar space.

5.2.1.1 Data generation

Transformers require a large data set that captures the domain of interest. 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 einsum grammar shown in Figure 5.3. The generated programs are then exported to an einsum compiler which generates equivalent C code. As the program space is unbounded, we limit both the number of tensors and their dimensionality to 5. This gives a data set of approximately 800k $\langle P, E \rangle$ pairs.

```c
void f(float *X, float *Y, float z, int n) {
  for (int i = 0; i < n; i++) {
    X[i] = Y[i] + z - 2.0;
  }
}
```

(a) The original code (Step 1).

```c
void f(tensor *X, tensor *Y, tensor *Z) {
  int X1_dimension = (int)(X->dims[0]);
  float* X_vals = (float*)(X->vals);
  ... // More definitions

  #pragma omp parallel for
  for (int32_t i = 0; i < Y1_dimension; i++) {
    X_vals[i] = Y_vals[i] + Z_val - 2.0;
  }
}
```

(b) Compiler-generated code (Step 2).

```c
void f(tensor *a, tensor *b, tensor *c) {
  for (int32_t i = 0; i < DIM; i++) {
    a[i] = b[i] + c - CONS;
  }
}
```

(c) The post-processed code (Step 3).

Figure 5.4: Overview of the post-processing pipeline. (a) Original C code. (b) Compiler-generated tensor code. (c) Post-processed tensor code used to train the Transformer.

Out of distribution data. 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. Consider the code in Figure 5.4(a) corresponding to the simple einsum program $x : i = y : i + z - 2.0$. Figure 5.4(a) represents a standard hand-written implementation in C, while Figure 5.4(b) shows a highly condensed version of the code generated by the compiler. It also contains wrapper code and memory duplication and is considerably longer than the original hand-written program illustrating the out-of distribution nature of synthetic data.

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 gives the code shown in Figure 5.4(c).

5.2.1.2 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 5.3 were part of the training data. The programs were then tokenized using byte-pair-enconding (BPE). [Sennrich et al., 2015] before used for training. The Transformer has 6 encoder layers and 6 decoder layers, with 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 [Kingma and Ba, 2017].

5.2.1.3 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 4 and use the best prediction of einsum output as our initial guess. We replace any constant tokens such as $CONS$ with a small random integer before passing on to the next stage.

5.2.2 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 5.2, to facilitate the measure task, we lower both programs to a common abstraction level. We lower the expression $\hat{E}$ using an einsum compiler [Kjolstad et al., 2017] 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.

5.2.2.1 Variable Similarity

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

5.2.2.2 Indexing Similarity

The second metric, Sim_{index}, considers the index expressions used in the program. We use Polly's [Grosser et al., 2012] 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 [Levenshtein, 1966] 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 [Navarro, 2001].

5.2.2.3 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.

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. In the example depicted in Figure 5.1, the guess (2), `a:i = b:i * b:k`, contains two variables while the original program (1) has 3. Therefore the variable similarity metric is to $1 - (\frac{3-2}{3}) = 0.67.$, i.e two-thirds of the way to matching.

To check indexing similarity, the indexing tuples are respectively $Indices(IR_{\hat{E}}) = (1, 1, 0, 0, 0, 1)$ and $Indices(IR_P) = (1, 1, 0, 2, 0, 2, 0, 2, 1)$, which results in distance value $D = 3$ and a similarity value of also $1 - (\frac{3}{9}) = 0.67.$ Although both (1) and (2) have a single multiplication operator, both perform implicit summation of indices. At the LLVM IR level, that results in a the presence of multiplication and addition instructions. The operators list for (1) and (2) are respectively $Ops_1 = \{App_1 = (Mul, b, b), App_2 = (Add, b, b)\}$ and $Ops_2 = \{App_1 = (Mul, b, c), App_2 = (Add, b, c)\}$, which results in a high arithmetic similarity of 0.8.

5.2.3 Edit

Consider the box labeled `Edit` in Figure 5.2. 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. To avoid cycles, KONRUL keeps a set with every candidate visited during search. If a candidate has already been explored, KONRUL discards it and select a different edit rule to continue exploration.

The set of parameterizable edit rules is shown in Figure 5.5. 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 ex-

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

Figure 5.5: The set R 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 5.3. An asterisk (*) indicates that the expression has been introduced or changed by the edit rule.

pression. At each iteration, we select a similarity metric, and apply edit rules targeting that specific similarity metric, as shown in Algorithm 6.

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

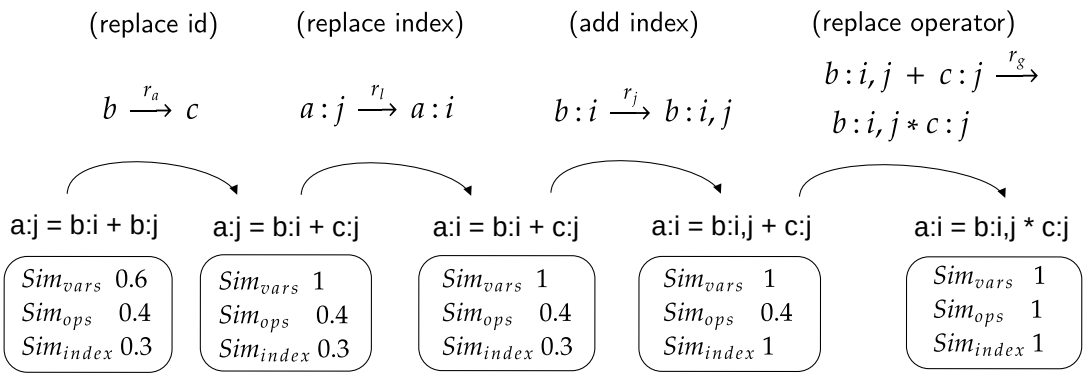


Figure 5.6: Edit rule selection based on program similarities. KONRUL first selects rules based on the variable similarity, secondly it analyzes indexing similarity and finally apply rules to edit arithmetic operators.

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.

Termination. It is possible that the edit and search process will reach a fixed time-out without finding a perfectly similar program. In this case, the testing phase will test all programs it encountered during the search with a high similarity score. The successful candidate will, as before, be then model checked. During our evaluation, we observe that timeouts are more frequent in the presence of candidates involving unary negation, for which the arithmetic similarity metric fails to adequately distinguish from subtraction. Timeouts also tend to occur when the initial guess is far from the solution in therms of variable similarity, requiring a larger number of edits to correct the size of the program.

Example. Figure 5.6 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.

5.2.3.1 Expressivity of Edit Rules

The synthesis problem we are solving is in general undecidable [Caulfield et al., 2015]. Thus, our algorithm is, unsurprisingly, not complete (i.e., it is not guaranteed to find a solution if a solution exists). However, given an einsum guess $E_0 \in \mathcal{L}(G)$, and the correct solution $E \in \mathcal{L}(G)$, there always exists a finite sequence of edit rules $r_1 \dots r_n \in R$ that can be applied to generate a sequence of Einsum expressions $S(E_0, E) = E_0 \xrightarrow{r_1} \dots \xrightarrow{r_n} E$, that terminates in the expression E.

Treat each expression as a sequence of terminal symbols in grammar G, i.e., let E_0 and E each be a sequence of symbols in Σ, where $\Sigma = \{T_0, T_1, \dots, C_0, C_1, \dots, i, j, k, l, m, n, :, -, +, *, /, =\}$. Any symbol in the sequence corresponding to E_0 can be deleted, mutated or inserted using an edit rule in R: Any tensor or constant symbol T_0, T_1, C_0, C_1 can be mutated by r_a and r_b, inserted by r_c and r_d and deleted by r_e and r_f. Any index symbol i, j, k, l, m, n can be mutated by r_m, inserted by r_j and r_k and deleted by r_l. Any operator symbol $+, -, *, /$ can be mutated by r_g, inserted by r_c, r_d or r_h and deleted by r_e, r_f or r_i. By definition, both expressions are in the form $tensor = expr$, so there is no need to modify the $=$ symbol. Thus, there exists a sequence of rewrite rules to convert any E_0 to E.

Note that, whilst a sequence of edit rules always exists, we are not guaranteed to find it by selecting rules that give intermediate expressions with increasing similarity. This is because, although our similarity metrics are inspired by distance metrics like Levenshtein distance [Levenshtein, 1966] which do guarantee a minimum number of edits needed to transform one sequence to another, our metrics operate on a lower level representation than the search process.

Our metrics are not perfect due to artifacts introduced by the lowering process. For example, auxiliary variables may be introduced, which affect Sim_{vars} and Sim_{index}, and compiler optimizations in the lower may remove operators, affecting Sim_{ops}. If the search reaches a timeout without finding a solution, we check the closest intermediate candidates. This enables KONRUL to find a solution, provided it was in the sequence of expressions explored.

5.2.4 Check (Testing and Verification)

The final stage of the process is to check for correctness denoted by the `Verify` box in Figure 5.2. Once $IR_{\hat{E}}$ is determined to be sufficiently similar to IR_P, according to the similarity metrics, the *Check* call in Algorithm 5 tests that the result is correct. We do this using observational equivalence [Feser et al., 2023] — 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 [Magalhães et al., 2023]. 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. 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 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. This gives the programmer guarantees that the lifted code will be exactly the same as the code they wrote, within the limitations described above. This step is important for efficient integration of KONRUL into large-scale projects, where the volumes of lifted code are high and the effort required to manually verify the lifting is equally high.

We lower from the original C code into MLIR [Lattner et al., 2021], and we lower the einsum into MLIR using JAX [JAX, 2024]. We then generate a C file, which initializes two copies of a set of non-deterministically 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 [Kroening and Tautschnig, 2014], a bounded model checker for C programs, to verify that this assertion is never violated.

Given a input C program, CBMC constructs a formula that represents all paths in the input and is satisfiable if and only if the assertion at the end is violated. To do this, it must unroll all the loops in the program, but we note that, since we fix the size of the input matrices, the number of times a loop must be unrolled is also fixed. The formula is then passed to an SMT solver (we use cvc5 [Barbosa et al., 2022]), which determines if the formula is satisfiable (i.e., the input C code is not equivalent to the lowered einsum code) or unsatisfiable (i.e., the input C code is semantically equivalent to the lowered einsum code).

CBMC supports full IEEE floating-point semantics, and initially, we attempt to verify the correctness of all solutions using floating-point semantics. We use a timeout of 3 minutes for this. However, verifying equivalence with floating point is challenging [Barbosa et al., 2022], and a small number of benchmarks exceed this timeout (this does not mean that the solutions are incorrect, simply that the SMT solver has not managed to prove that they are correct). We thus extend CBMC to support arrays of real numbers, which we internally represent as rationals. Programs that are proven equivalent when all floating-point datatypes are replaced with reals are proven to be performing mathematically equivalent operations on the inputs, but the semantics of floating-point operations and rounding errors are not taken into account in the verification. This could cause small differences in the output, for example, if the order of floating-point operations is changed [Goldberg, 1991].

5.3 Experimental Setup

This section describes in detail the environment (section 5.3.1), alternative approaches (section 5.3.2) and methodology (section 5.3.3) used to evaluate KONRUL.

5.3.1 Environment

Benchmarks. We extended the benchmark suite from Chapter 4 with the image processing benchmarks from `blend` [Ahmad et al., 2019] and C implementations of kernels from the deep learning framework `llama` [Touvron et al., 2023b], both evaluated in [Qiu et al., 2024a]. Additionally, we include higher-dimensional tensor contraction programs described in [Chelini et al., 2021]. These are show in table 5.1.

Software. KONRUL's language model is implemented using Fairseq [Ott et al., 2019] version 0-12.2 with Google's SentencePiece [Kudo and Richardson, 2018] tokenizer. We use TACO version 0.1, LLVM version 14.0, PyTorch version 2.3.0, and `gcc` version 9.4.

Table 5.1: Tensor contraction benchmarks taken from [Chelini et al., 2021] and the correspondent einsum counterparts.

Benchmark	Einsum
ab-ac-cd	a:i,j = b:i,k * c:k,j
ab-acd-dbc	a:i,j = b:i,k,l * c:l,j,k
ab-cad-dcb	a:i,j = b:k,i,l * c:l,k,j
abc-bda-dc	a:i,j,k = b:j,l,i * c:l,k
abc-ad-bdc	a:i,j,k = b:i,l * c:j,l,k
abc-acd-db	a:i,j,k = b:i,k,l * c:l,j
abcd-aebf-dfce	a:i,j,k,l = b:i,m,j,n * c:l,n,k,m
abcd-aebf-fdec	a:i,j,k,l = b:i,m,j,n * c:n,l,m,k

Hardware. 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.

5.3.2 Competitive Approaches

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

- **TF-Coder** [Shi et al., 2022]: neural-guided bottom-up synthesizer for TensorFlow programs. We convert its output program into einsum to allow direct comparison.

- **C2TACO** [Magalhães et al., 2023]: bottom-up enumerative synthesizer for TACO [Kjolstad et al., 2017] programs that uses static program analysis to drive search. Described in full detail in Chapter 4

- **Tenspiler** [Qiu et al., 2024a]:verified-lifting-based approach that lifts tensor code written in low-level languages such as C++ and Python to different tensor API/DSLs. It initially lifts the original code to a specialized intermediate representation that can be used by an SMT solver to formally verify its correctness. Once equivalence is proved, the intermediate program is translated to the specified target DSL.

- **GPT-4** [Achiam et al., 2023]: 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 as variations of its search.

- **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.

5.3.3 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 IO example, enabling it to significantly lift more candidates than reported in previous work [Magalhães et al., 2023]. Tenspiler was evaluated using the hand-written grammars provided in the reproducibility artifact [Qiu et al., 2024b]. 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.

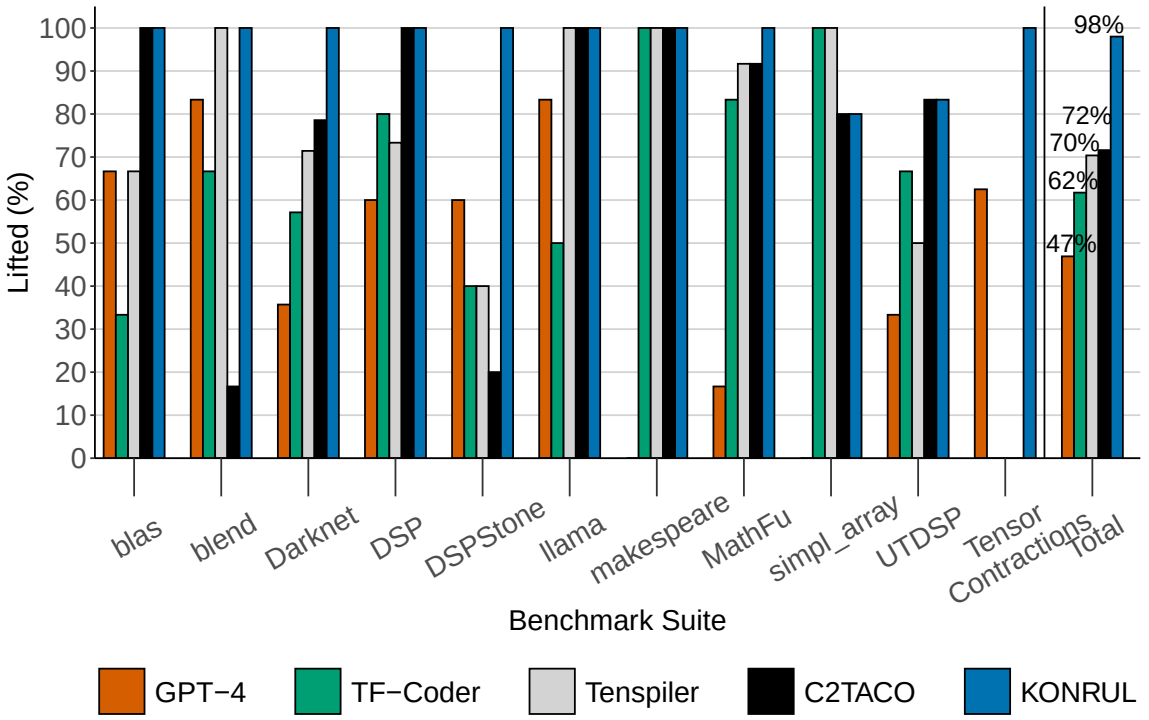


Figure 5.7: Lifting coverage across different benchmark categories. Y-axis shows the percentage of programs lifted by each approach in each category listed on the X-axis. The TOTAL group shows the average across the whole benchmark suite.

5.4 Evaluation

This section evaluates KONRUL against the alternative methods across three metrics: success rate, correctness (section 6.5.1) and efficiency of exploration of the program space (section 6.5.2). This is followed by a breakdown of KONRUL's lifting time (Section 5.4.3) and verification effectiveness (section 5.4.4). Finally, we examine the speedup obtained by lifting to Einsum notation (section 5.4.5).

5.4.1 Coverage

Figure 5.7 shows the percentage of programs successfully lifted by each technique across benchmark categories. KONRUL lifts 98% of the programs across the entire suite, lifting 100% in 9 categories. It fails to lift 2 programs due to LLVM common sub-expression elimination during 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`, `DSP-Stone`, 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 IO examples as specifications of the target. C2TACO generates syntactically correct but semantically incorrect code in 2 cases due to insufficient coverage by its IO examples. As an example, in one case it generates $C = AB$ rather than $C = AB + C$, assuming 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.

5.4.1.1 Coverage by Program Complexity.

Figure 5.8 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.

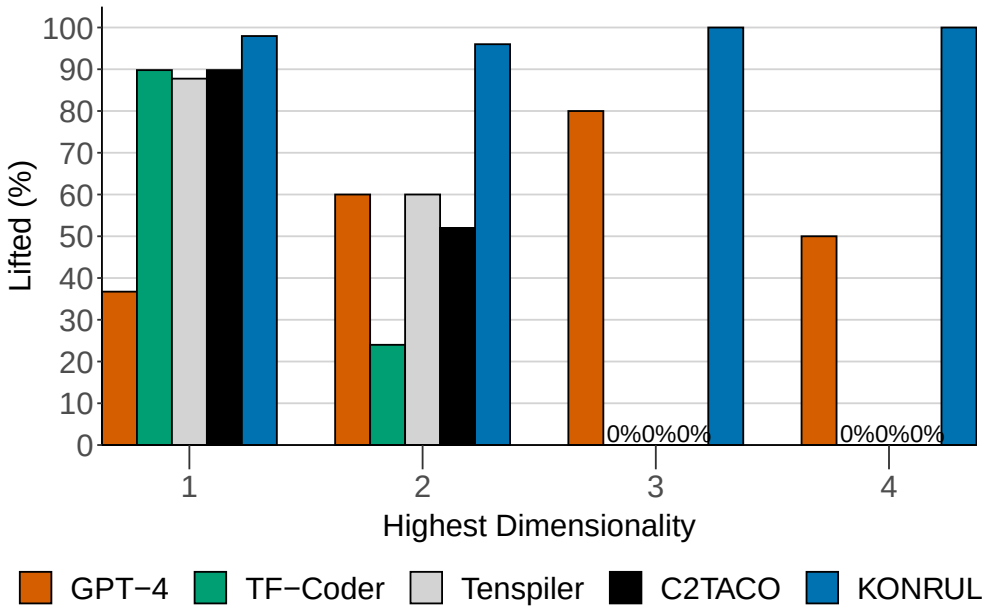


Figure 5.8: Lifting coverage by program complexity. Y-axis shows the percentage of programs lifted given different values of highest-dimensionality listed on the X-axis.

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.

5.4.2 Program Space Exploration

This section evaluates efficiency of different techniques 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 the underlying SMT solver considers. Instead, we evaluate KONRUL against Tenspiler in terms of lifting time.

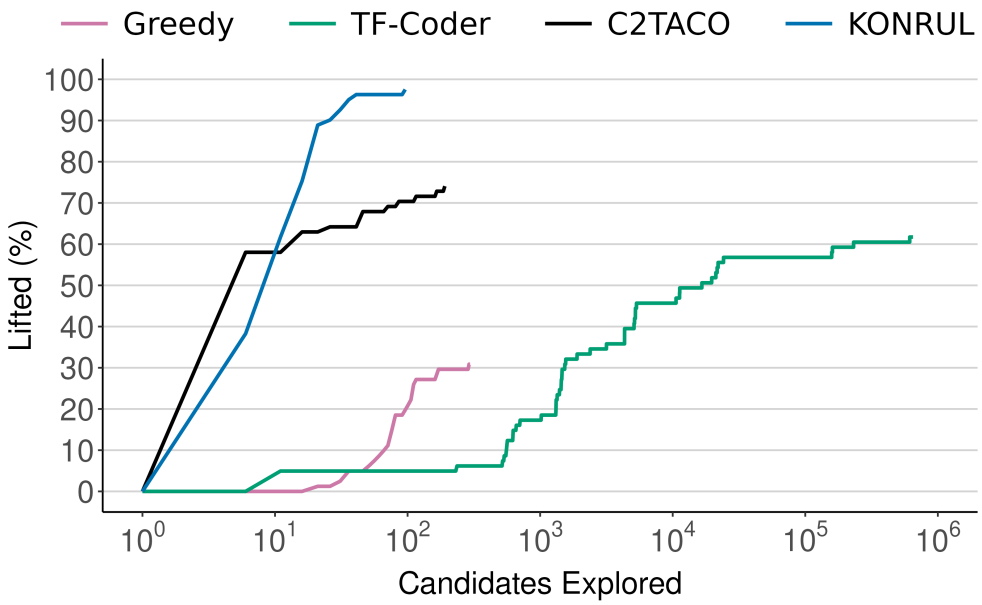


Figure 5.9: Cumulative number of candidates explored during lifting by different IO-based techniques. X-axis shows cumulative number of candidates explored and Y-axis shows the number of programs lifted. X-axis is on logarithmic scale.

5.4.2.1 IO-based Techniques

Figure 5.9 evaluates the efficiency of the various IO-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, 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.

Table 5.2: Average number of candidates explored vs candidates tested with IO across different benchmark categories.

Category	Explored	IO 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

5.4.2.2 The Impact of Similarity Metrics.

As explained in Section 5.2, 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 5.2 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 IO 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.

5.4.2.3 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.

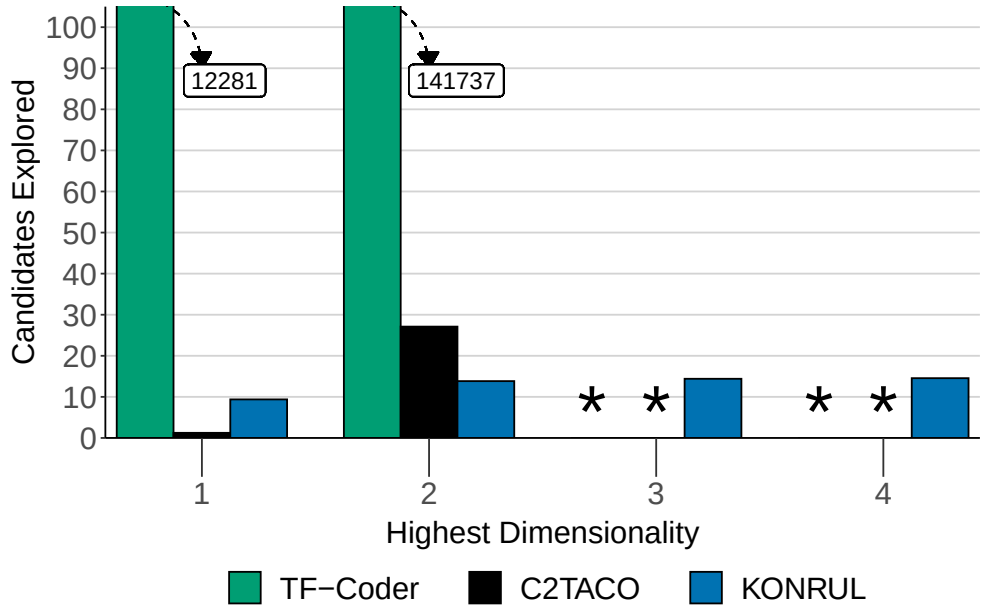


Figure 5.10: Search space exploration by program complexity. Y-axis shows the average number of candidates visited during search given different values of highest-dimensionality listed on the X-axis. An asterisk (*) means that the method could not lift any program in that complexity class.

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.

5.4.2.4 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.

Table 5.3: Average number of candidates explored by KONRUL vs candidates tested with IO by highest dimensionality.

Highest Dimensionality	Explored	IO Tested
1	9.3	2
2	13.8	2.9
3	14.4	2.6
4	14.55	2.5

5.4.2.5 Program Space Exploration by Complexity

Similarly to coverage, we evaluated space exploration against program complexity. For the IO-based techniques, we plot the average number of candidates explored, shown in Figure 5.10. All approaches need to visit more candidates as the programs to be lifted become more complex.

C2TACO is the best method for benchmarks with dimensionality 1 exploring an average of 1.2 candidates. Nevertheless, that number rapidly increases to 29.4 on dimensionality 2. C2TACO cannot lift any programs with dimensionality greater than 2. Although KONRUL needs to explore more candidates for programs with high-dimensionality tensors, it is the the only search-based method able to lift them.

We again compare KONRUL and Tenspiler in terms of average lifting time. Tenspiler is faster for programs with highest-dimensionality 1 and 2, however, it is not able to lift the benchmarks containing tensors with 3 or 4 dimensions, whereas KONRUL lifts all the programs taking 71 and 78 seconds respectively.

Finally, we evaluated the KONRUL's number of candidates tested in contrast to the number of candidates visited. As we can see in table 5.3, the number of candidates explored increase as the programs become more complex. However, the number of candidates actually tested remains stable. KONRUL always tests fewer candidates than it explores regardless of the highest-dimensionality.

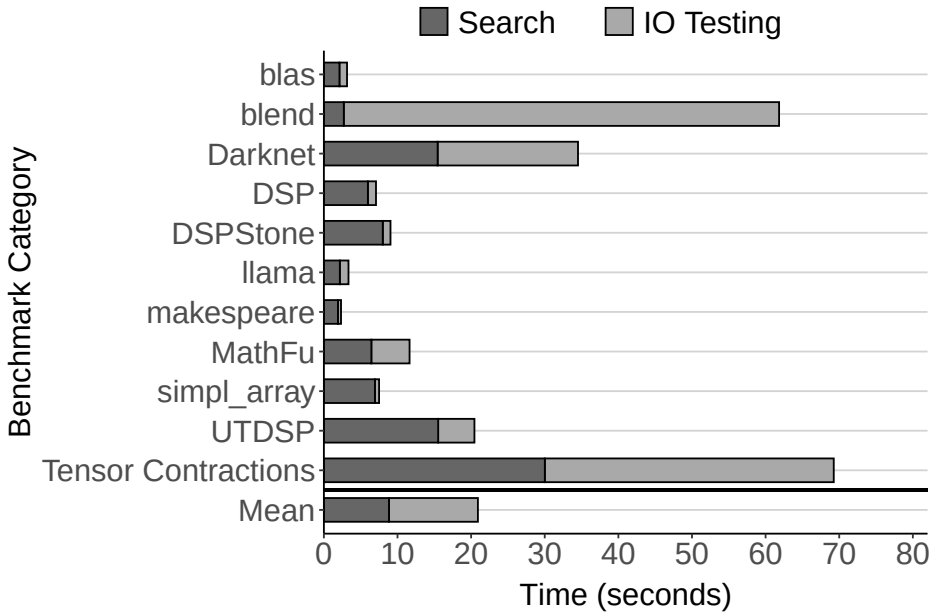


Figure 5.11: Lifting time breakdown of KONRUL. Y-axis shows different benchmarks categories with the average value at the bottom. X-axis shows the time taken by KONRUL to search for candidates and test the ones near to the original C program.

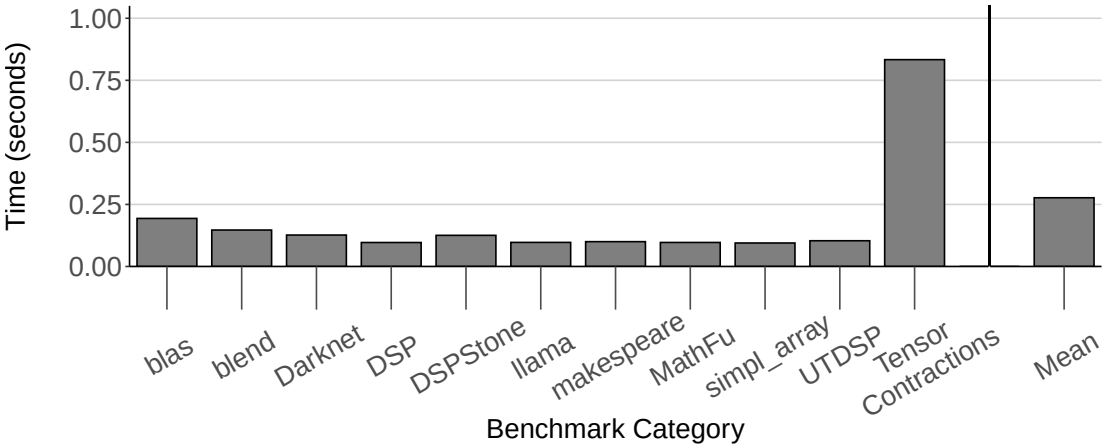
5.4.3 Detailed Evaluation of Lifting Time

Figure 5.11 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.

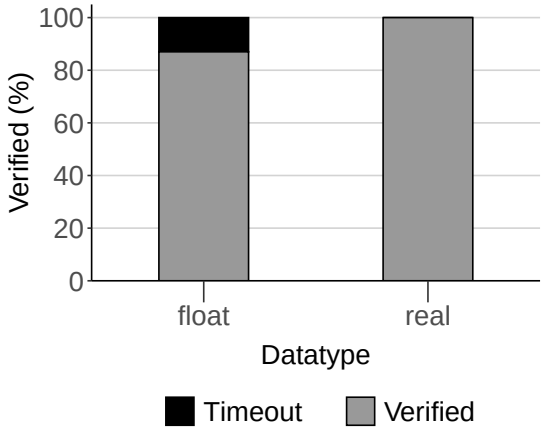
KONRUL takes an average of 23 seconds to lift a program, of which 10 seconds correspond to search and 13 to testing. Search time is usually longer than testing except for `blend`, `Darknet`, and `Tensor Contraction` programs. For `blend` and `Darknet` are the categories in which KONRUL IO tests more than one candidate more often, 83% and 27% respectively. The `Tensor Contraction` benchmarks perform heavy computation on large tensors and the inputs are constant sizes. Testing time can be reduced by using smaller inputs.

5.4.4 Verification

We measure the time taken by CBMC to verify a lifted program. As shown in Figure 5.12a, the overall time is small with an average of 0.27 seconds.



(a) Average verification time across benchmark categories.



(b) Verification coverage using floating-point and real datatypes.

Figure 5.12: CBMC verification results. 5.12a reports average verification time across benchmark categories. 5.12b reports verification coverage using floating-point and real datatypes for all programs lifted by KONRUL.

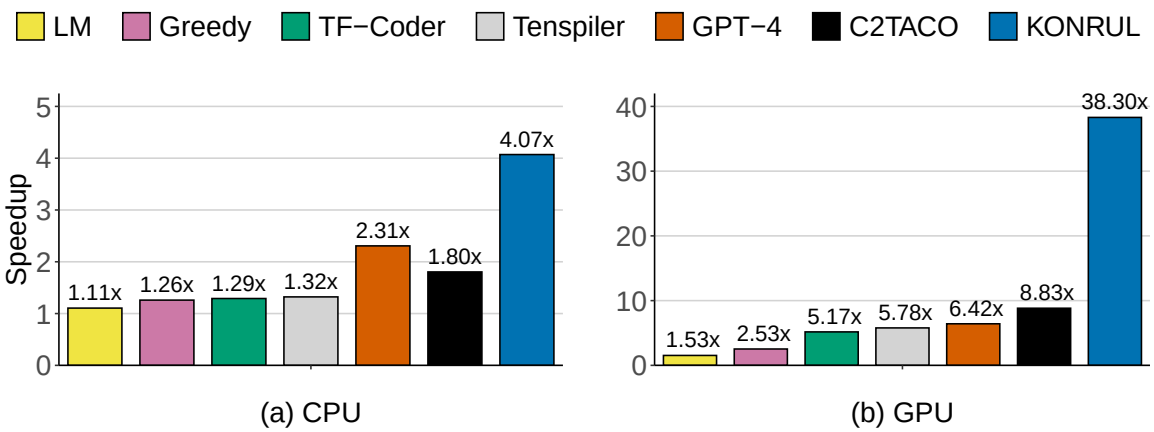
The longest time taken is 0.83 seconds on the `Tensor Contractions` programs. These programs contain higher-dimensional tensor operations in which bounded model checking can take up to 5.2 seconds. The fastest verification occurs `DSP`, `llama` and `simpl_array` categories with an average time of 0.09 seconds.

We verify lifted programs using two different datatypes: floating points and reals. However, verifying equivalence with floating point is challenging [Barbosa et al., 2022], 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 5.12b 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.

5.4.5 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 5.13 depicts a summary of the geomean 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 4.07x on the CPU and 38.3x 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 1.29x, but improves on the GPU providing a 5.17x speedup. Tenspiler performs similarly, providing 1.32x and 5.78x speedups respectively. Finally, C2TACO achieves 1.80x speedup on CPU and 8.83x on GPU while GPT-4 reaches 2.31x and 6.42x 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 improvement. The variation baselines



(a) CPU (b) GPU

Figure 5.13: 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.

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

5.5 Conclusions

This chapter 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 methodology, 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 4.07x on a multi-core CPU and 38.30x on a GPU platform.

Chapter 6

Sparse Lifting with Neuro-Guided Sketch Synthesis

The two previous chapters have focused on dense linear algebra. This chapter tackles the more challenging problem of porting legacy *sparse* linear algebra code to libraries and DSLs. We present a neural-guided sketch-based lifting technique that 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 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.

We present the challenges of lifting sparse code with a motivating example in section 6.1.1. We then summarize the pipeline of SLEB in section 6.2 and describe the technique details in section 6.3. We present the experimental setup and an extensive evaluation in sections 6.4 and 6.5 respectively. Section 6.6 concludes this chapter.

6.1 Sparse Lifting

There has been little progress in lifting sparse code given the complexity of this problem. The reason is that analyzing legacy sparse computation is difficult [Ginsbach, 2020]. Unlike dense operations, there are many different data

storage formats employed [Won et al., 2023] and idiosyncratic ways for programmers to code sparse linear algebra. Furthermore, while tensor DSLs are able to express and support a wide range of higher-order algebraic operations [Ahrens et al., 2025], 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.

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.

6.1.1 Example

To illustrate our approach, Figure 6.1 presents a running example of SLEB The original source code in ① is sparse matrix–matrix multiplication (SpMM) from the SpComm3D framework [Abubaker and Hoefler, 2024], 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.

For this example, 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.

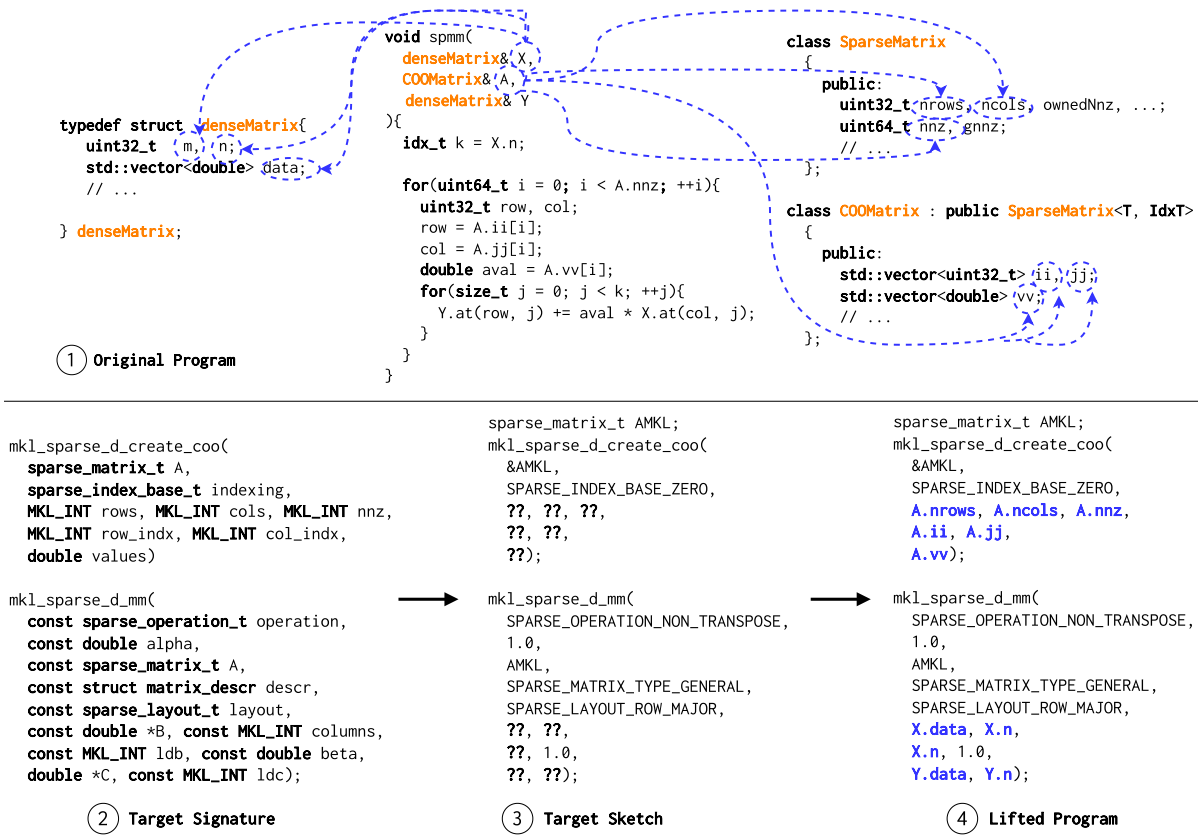


Figure 6.1: Example of sparse code lifting. The original program `spmm` ① is replaced by the lifted program ④ which consists of two calls to the MKL library with parameters originating as variables in the original program. The signature for these calls is shown in ② and partially completed with known 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.

6.1.1.1 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 combinatorial

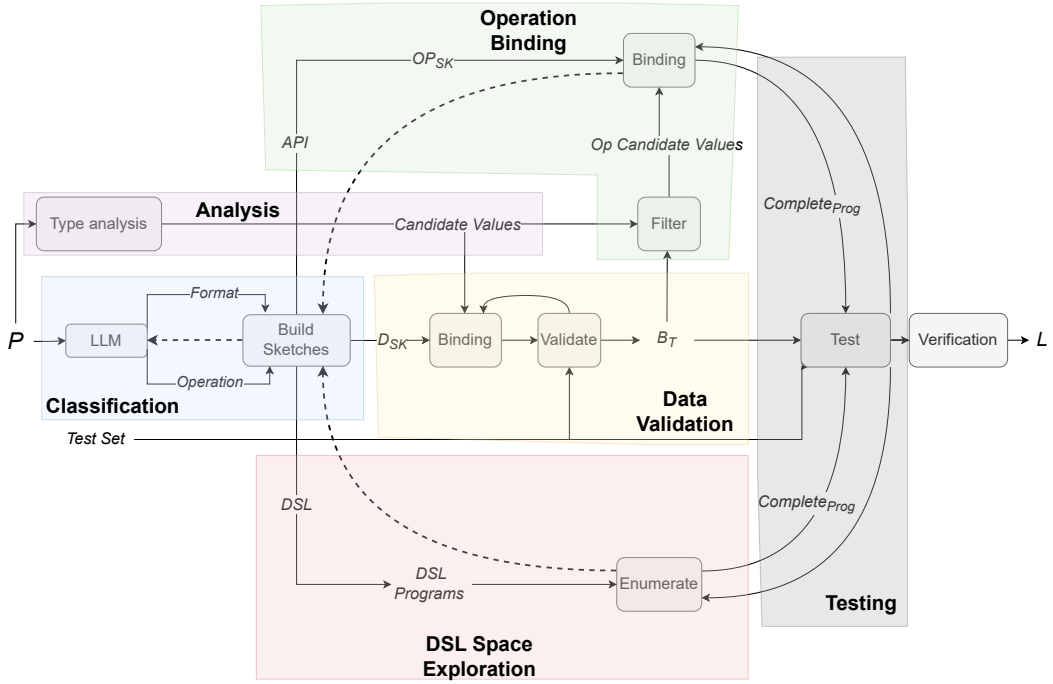


Figure 6.2: The overall pipeline of SLEB. A program P is classified by operator and data format from which a data and operator sketches are generated. Program analysis is performed on P to determine candidate variables t match the holes in the generated API or DSL sketch. Data binding finds the correct 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

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.

.

6.2 SLEB Overview

Figure 6.2 provides a high level overview of SLEB's components. It combines an LLMs, program analysis and smart synthesis to generate API or DSL code L from an input program P.

6.2.1 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 skectch is a function call with holes in as illustrated in Figure 6.2. For DSLs, the operation sketch is a smaller grammar expressing the space of possible matching programs.

6.2.2 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: integers, floating points, as well as doubles and arrays of integers, floats and doubles. These are all candidates for the next stage data binding.

6.2.3 Data Binding

In the third sate 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.

6.2.4 Operator Binding

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

6.2.5 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.

6.2.6 Testing

The final stage performs IO testing on the complete candidates to check equivalence. Although data validation discards many candidates without running them, SLEB still executes complete programs during the operation binding or DSL exploration stages. Hence, the efficiency of SLEB depends on reasonable execution time for the candidate programs. Unlike the previous technique, SLEB does not perform formal verification due to the complexity of verifying sparse code. This complexity is due to two main reasons. First, sparse implementations exhibit complex code structure and control flow, with a large number of possible execution paths which are difficult for SMT-based tools to reason about. Second, sparse code uses different storage formats to store data, and the resulting data dependencies across memory locations introduce additional challenges for such tools. Nevertheless, a program that passes all tests can be passed onto different external verification tools such as [Dyer et al., 2019] to perform this process.

6.3 Implementation

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 7. 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 6.3.3. The `bind` function is described in detail in Algorithm 8. 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.

6.3.1 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* for the input program. We use GPT-4.o [Achiam et al., 2023] as the LLM, giving it the prompt below, followed by the source code of P.

Example prompt. ”`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`

Algorithm 7: SLEB lifting algorithm

input : source computation kernel P, set of tests ϕ

output : lifted MKL/TACO program, or no solution

```
1  Algorithm lift(P, φ)
2      format, Op, N, orders ← queryLLM(P);
3      candidates ← analyze(P);
4      D_SK ← sketch(format);
5      DB ← bind(D_SK, candidates);
6      for b ∈ DB do
7          if validate(b, P, φ) then
8              if Op is API supported then
9                  Op_SK ← sketch(Op);
10                 candidates_Op ← filter(b, candidates);
11                 OPB ← bind(Op_SK, candidates_Op);
12                 for op_b ∈ OPB do
13                     L ← compile(b, op_b);
14                     if test(L, φ) then
15                         return L
16                 end
17             else if Op is DSL supported then
18                 E ← genProgSpace(Op, N, orders);
19                 for e ∈ E do
20                     L ← compile(b, e);
21                     if test(L, φ) then
22                         return L
23                 end
24     end
25     return no solution
```

```
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.

6.3.2 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 6.1 ①, variable X cannot be directly bound to a sketch hole. We can, however, determine that

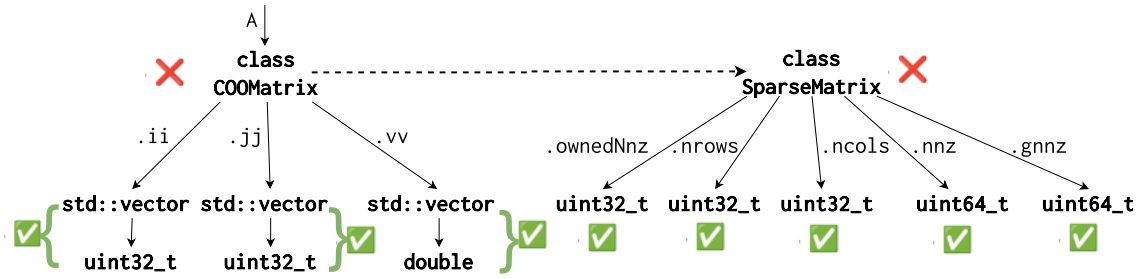


Figure 6.3: Example of program analysis on a type tree. Dashed arrow represents inheritance relation.

`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.

Example. Figure 6.3 shows how SLEB performs type analysis. Given the variable `A` from the code in Figure 6.1, we analyze its type and detect that `COOMatrix` does not type check with the holes in none of the sketches, therefore `A` cannot be a candidate for binding on its own. We proceed to analyze the children of `COOMatrix` and check that the subtree `std::vector` $\rightarrow$ `uint32_t` is type-compatible since it can represent a pointer of integers, therefore we consider `A.ii` and `A.jj` as candidates. Similarly, we check that `std::vector` $\rightarrow$ `double` is type-compatible so `A.vv` can be a candidate. `COOMatrix` is a subclass of `SparseMatrix`, hence we proceed to analyze that subtree and select `A.OwnedNnz`, `A.nrows`, `A.ncols`, `A.nnz`, and, `A.bnnz` as binding candidates.

6.3.3 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:

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

We perform binding using type-constrained enumeration as described in Algorithm 8. SLEB implements a recursive depth-first based algorithm to generate all constraint-valid bindings. 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 three constraints. The *type compatibility* constraint states that only combinations of variables whose types match the expected input types of the API call are considered. The *unique pointer binding* constraint express that A pointer variable may be bound to at most one placeholder, i.e., a hole, in the sketch. Finally, the *naming* constraint restricts the bindings based on variable identifiers. 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.

6.3.3.1 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

Algorithm 8: Binding generation process for a given sketch SK.

Procedure *genBindings*$(i, H, V, \mathcal{B}, current, Seen)$
  if $i = |H|$ **then**
    $b \leftarrow \{h \mapsto v \mid (v, h) \in current\}$;
    if $b \notin Seen$ **then**
      $\mathcal{B} \leftarrow \mathcal{B} \cup \{b\}$;
      $Seen \leftarrow Seen \cup \{b\}$;
    end
    return
  $h \leftarrow H[i]$;
  for $v \in V$ **do**
    if $\langle h, v \rangle$ *type-checks and obeys all rules* **then**
      $current \leftarrow current \cup \langle h, v \rangle$;
      genBindings$(i + 1, H, V, \mathcal{B}, current, Seen)$;
      $current \leftarrow current \setminus \langle h, v \rangle$;
  end
Procedure *bind*$(SK, \mathcal{V}, Seen)$
  $\mathcal{H} \leftarrow holes(SK)$;
  $\mathcal{B} \leftarrow \emptyset$;
  genBindings$(0, \mathcal{H}, \mathcal{V}, \mathcal{B}, \emptyset, Seen)$;
  return $\mathcal{B}$;

storage-format constraints for F. These constraints represent the semantics of each format and are used to discard invalid bindings. We support four different sparse formats, CSR, CSC, COO and JDS.

In the Compressed Sparse Row (CSR) format, the number of nonzero elements (NNZ) is implicitly defined by the last entry of the *rows_index* array. The *rows_index* array must either have length $n_{\text{rows}} + 1$, or alternatively be represented using two arrays of length n_{rows} that store the start and end offsets of each row. The *cols_index* and *values* arrays must both have length equal to NNZ. Additionally, the *rows_index* array must be monotonically non-decreasing, with its first entry equal to zero.

The Compressed Sparse Column (CSC) format is analogous to CSR but organized by columns. In this representation, NNZ is implicitly given by the last

Algorithm 9: Filter out values already used in a binding.

```
1 Procedure FILTER(b, V):
2     S ← {v' ∈ V | v' ∉ {v | (h, v) ∈ b}};
3     return S;
4 end
```

entry of the *cols_index* array. The *cols_index* array must have length $n_{\text{cols}}+1$, or be represented by two arrays of length n_{cols} that store the starting and ending offsets of each column. The *rows_index* and *values* arrays must both have length equal to NNZ. Furthermore, the *cols_index* array must be monotonically non-decreasing, with the first value equal to zero.

In the Coordinate List (COO) format, the sparse matrix is represented explicitly by three arrays: *rows_index*, *cols_index*, and *values*, all of which must have length equal to NNZ. Each entry in *rows_index* and *cols_index* specifies the row and column indices of the corresponding nonzero value. These indices must satisfy the bounds $0 \leq$ *rows_index*$[i] < n_{\text{rows}}$ and $0 \leq$ *cols_index*$[i] < n_{\text{cols}}$.

In the Jagged Diagonal Storage (JDS) format, the number of nonzero elements (NNZ) is implicitly determined by the last value of the diagonal pointer array. As a result, NNZ does not need to be explicitly stored elsewhere in the representation.

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.

6.3.4 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 $v \in B_T.V$ as shown in Algorithm 9. 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 para-

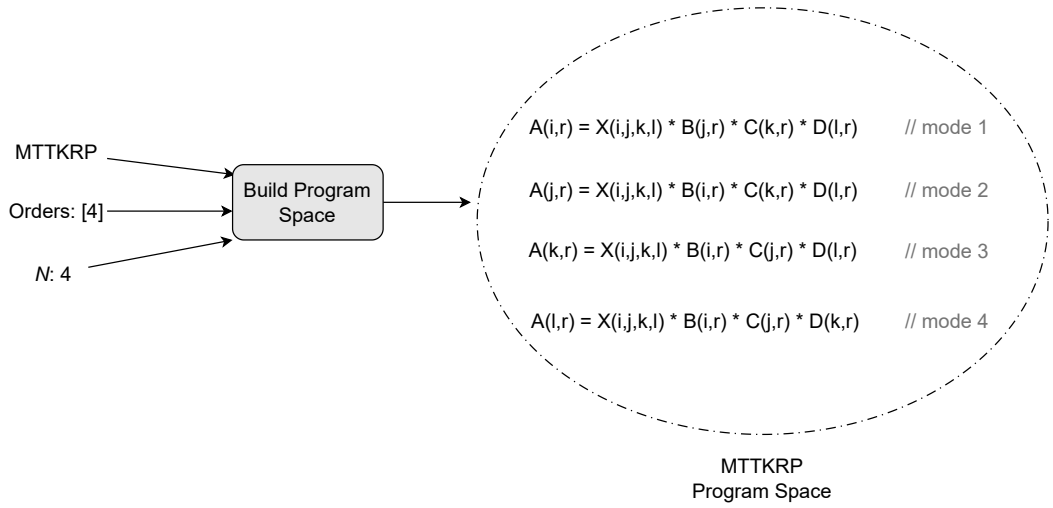


Figure 6.4: Example of TACO program space generation for a MTTKRP operation on a 4D sparse tensor.

meters 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.

6.3.5 DSL Exploration

Operations involving high-order sparse tensors are not well-supported in sparse API's. 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 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. SLEB supports the same grammar described in Figure 4.4.

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 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.

Example. Figure 6.4 illustrates an example of a TACO program space built by SLEB. Given an operation (MTTKRP), orders of sparse elements (4) and the number of elements in the operation (4), SLEB generates all possible programs and tests them sequentially. In this example, each generated program corresponds to the MTTKRP operation executed on one of the modes of sparse tensor X.

6.3.6 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.

6.4 Experimental Setup

This section describes the environment (section 6.4.1), alternative approaches (section 6.4.2) and methodology (section 6.4.3) used to evaluate SLEB.

6.4.1 Environment

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. There are benchmarks implemented in 3 different languages (C, C++, and Fortran77) and use 4 different sparse storage formats (CSC, CSR, COO, JDS). Table 6.1 depicts every benchmark in detail.

One of the most commonly used sparse operations is sparse matrix-vector multiplication (SpMV). This operation multiplies a sparse matrix for a dense

vector and store the result in a dense vector. Our evaluation includes fourteen SpMV benchmarks drawn from a diverse set of sources, including CSPARSE [Davis, 2006], DOLFINx [Baratta et al., 2023], GinkGO [Anzt et al., 2022], Netlib [Dongarra et al., 2001], the NAS Parallel Benchmarks [Löff et al., 2021], Parboil [Stratton et al., 2012], QuantLib [QuantLib, 2025], SciMark [Pozo and Miller, 2025], and a TACO-generated implementation from the SpEQ artifact [Laird, 2024],

We included sparse matrix-matrix multiplication (SpMM) benchmarks sourced from SuperLU [Li et al., 2011], Sextans [Song et al., 2022], and SpComm3D [Abubaker and Hoefler, 2024]. These benchmarks multiply a sparse matrix with a dense one and store the result as dense. We have also include sparse general matrix-matrix multiplication (SpGEMM) obtained from CSeg [An and Çatalyürek, 2021] and GinkGO [Anzt et al., 2022]. Unlike SpMM, SpGEMM multiply two sparse matrices and store the result as sparse.

Analogous to SpGEMM, sparse matrix addition (SpMAdd) sum two sparse matrices and store the result as sparse. We have one benchmark drawn from CSPARSE [Davis, 2006]. We gathered one sampled-dense-dense matrix multiplication (SDDMM) benchmark from SpComm3D [Abubaker and Hoefler, 2024]. This operation is commonly used in graph analytics and machine learning, where dense computations are restricted to the sparsity pattern of a matrix.

Our benchmark suite contains also high-order tensor benchmarks. The tensor-element-wise (TEW) category includes four benchmarks from the PASTA benchmark suite [Li et al., 2019], covering tensor element-wise addition, subtraction, multiplication, and division. We also selected two tensor–scalar benchmarks from PASTA, tensor–scalar addition (TSA) and tensor–scalar multiplication (TSM). Another tensor benchmark category contains programs that perform contractions. We included a sparse tensor–times vector (SpTTV) and a tensor– times matrix multiplication (SpTTM) both from PASTA. Finally, we evaluate on two Matricized Tensor Times Khatri-Rao Product (MTTKRP) benchmarks, sourced from PASTA and from the Splatt library [Smith and Karypis, 2016].

Table 6.1: Benchmark suite description. The ^{T} symbol means the benchmark operates on the transpose of the sparse input.

Benchmark	Format	Language	LOC	Source	Domain
Sparse Matrix Addition (SpMAdd)	CSC	C	26	CSPARSE	Sparse Linear Algebra
SpMV	CSC	C	28		
SpMV	CSR	C	19		
SpMV^{T}	CSR	C	19		
SpMV	COO	C	18		
SpMV^{T}	COO	C	18		
SpMV	CSR	C++	40	DOLFINx	PDE Solvers
SpMV	CSR	Fortran	16	Netlib	Numerical Libraries
SpMV^{T}	CSR	Fortran	16		
SpMV	CSR	C	11	NAS Parallel Benchmarks	HPC Benchmarks
SpMV	JDS	C	15	Parboil	HPC Benchmarks
SpGEMM	CSR	C++	180	GinkGO	Linear Algebra Libraries
SpMV	CSR	C++	29		
SpMM	CSC	C	106	SuperLU	Direct Solvers
MTTKRP	CSR	C	39	Splatt	Tensor Factorization
SpMM	CSR	C++	23	Sextans	Linear Algebra

Continued on next page

Benchmark	Format	Language	LOC	Source	Domain
SpGEMM	CSR	C++	414	CSeg	Column-segmented Matrix-Matrix Multiplication
SDDMM	COO	C++	15	SpComm3D	Communication Kernels
SpMM	COO	C++	14		
MTTKRP	COO	C	51	PASTA	Tensor Algebra
SpTTM	COO	C	43		
SpTTV	COO	C	41		
TEW addition	COO	C	66		
TEW division	COO	C	15		
TEW product	COO	C	39		
TEW subtraction	COO	C	65		
TSA	COO	C	17		
TSM	COO	C	10		
SpMV	CSR	C++	19	QuantLib	Quantitative Finance
SpMV	CSR	C	16	Scimark	HPC Benchmarks
SpMV	CSC	C	10	TACO-generated (from SpEQ artifact)	Compiler-generated

Software. 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.

Hardware. 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.

6.4.2 Competitive Approaches

We compared against three alternative approaches

- **LiLAC** [Ginsbach et al., 2020]: pattern matching approach that uses constraints over LLVM IR to detect SpMV and replace with calls to MKL and cuSPARSE.
- **SpEQ** [Laird et al., 2024]: lifter 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** [Achiam et al., 2023]: a popular LLM, which is provided with the original program and prompted to provide equivalent library or TACO code.

6.4.3 Methodology

We give a timeout of 10 minutes to each technique to lift a benchmark. We evaluate LiLAC and SpEQ with their respective artifacts [Ginsbach, 2020] [Laird, 2024]. GPT.4o was repeatedly tested with different prompts to find the best recall. To ensure a fair evaluation, when evaluating the performance, we use `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 [Kjolstad et al., 2017] in their evaluation. These datasets are described in Table 6.2.

Table 6.2: Real-world matrices and higher-order tensors used for performance experiments

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

6.5 Evaluation

This section evaluates SLEB's performance in terms of coverage (section 6.5.1 and lifting time 6.5.2. We also evaluate the speedup obtained by lifting the benchmarks to sparse backends (section 6.5.3).

6.5.1 Coverage

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

6.5.1.1 Coverage by Kernel

Figure 6.5 shows the percentage of programs successfully lifted by each technique grouped by operation. LiLAC has the lowest coverage across 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

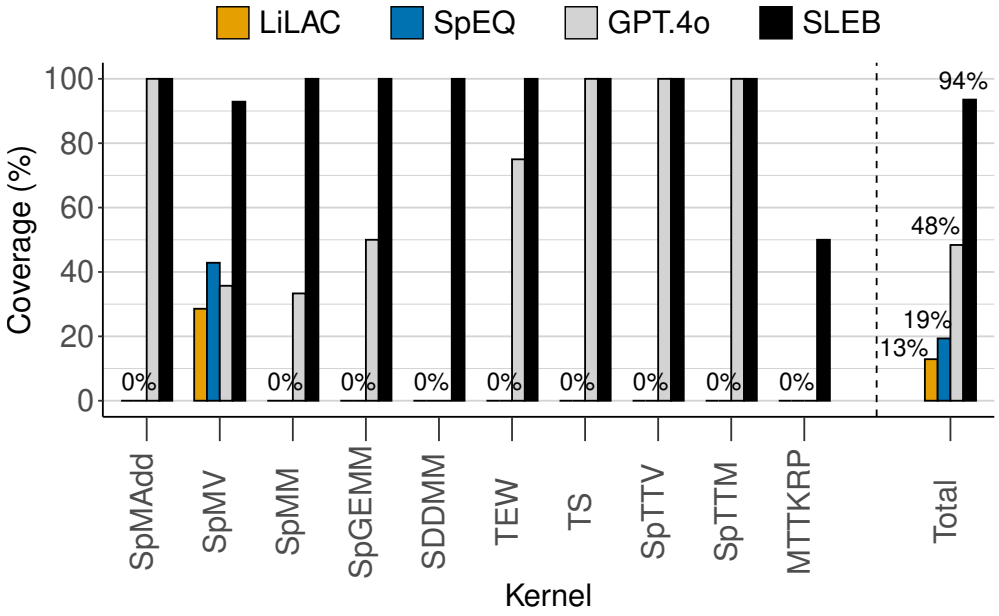


Figure 6.5: Coverage by different kernels.

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 struts/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 dif-

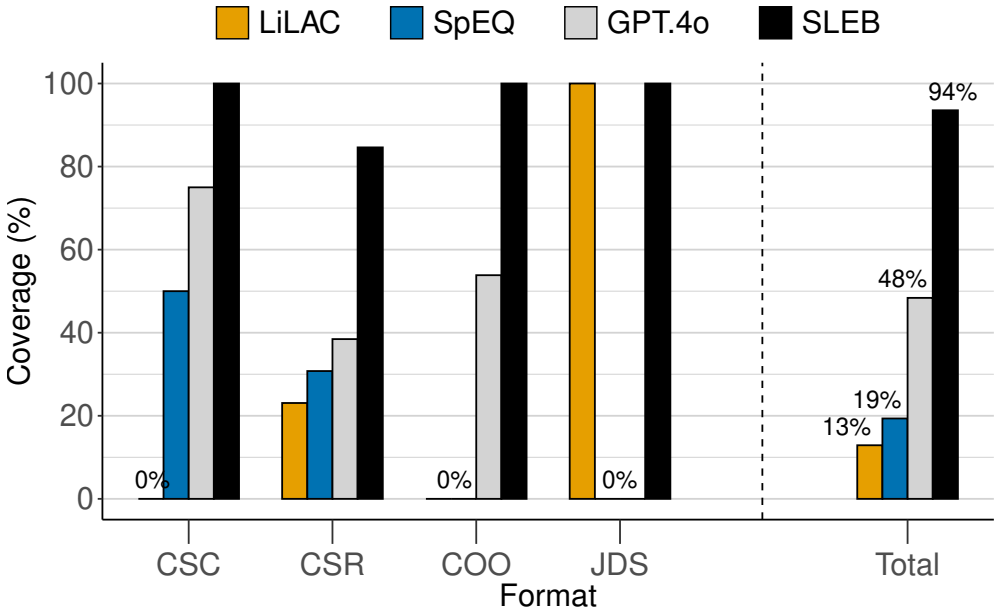


Figure 6.6: Coverage by different sparse storage formats.

ferent sparse operations, achieving 100% of coverage in all categories except for SpMV and MTTKRP. SLEB fails to lift the SpMV from GinkGO 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. 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.

6.5.1.2 Coverage by Format

Figure 6.6 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 and GinkGO SpMV as described above.

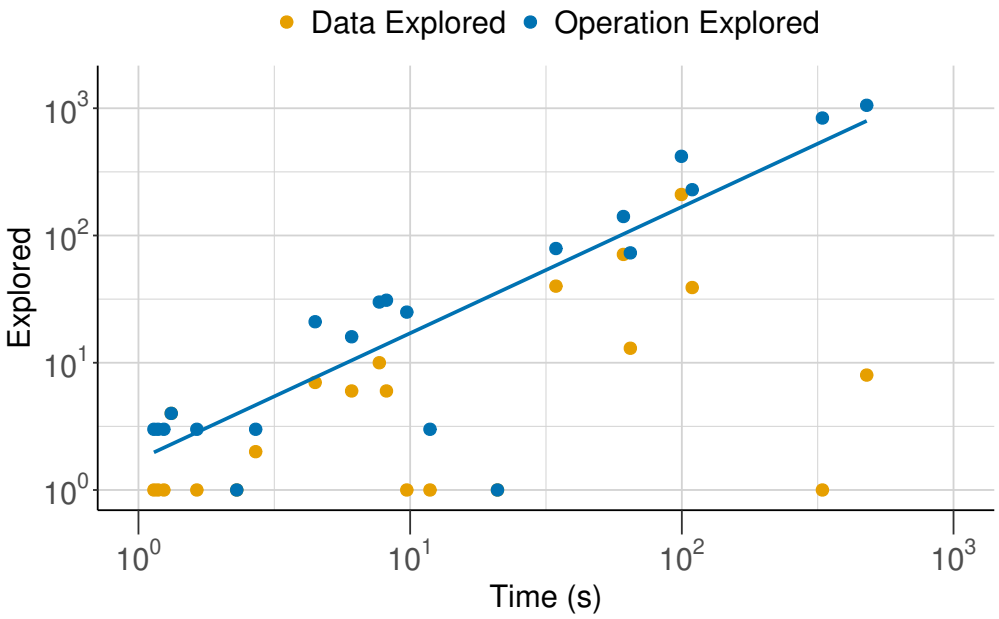


Figure 6.7: Relation between candidates explored and lifting time.

6.5.2 Lifting Time

Table 6.3 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 6.7. 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 SpMV^T, 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 6.3 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.

Table 6.3: Lifting time results. The ^{T} symbol means the benchmark operates on the transpose of the sparse input.

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

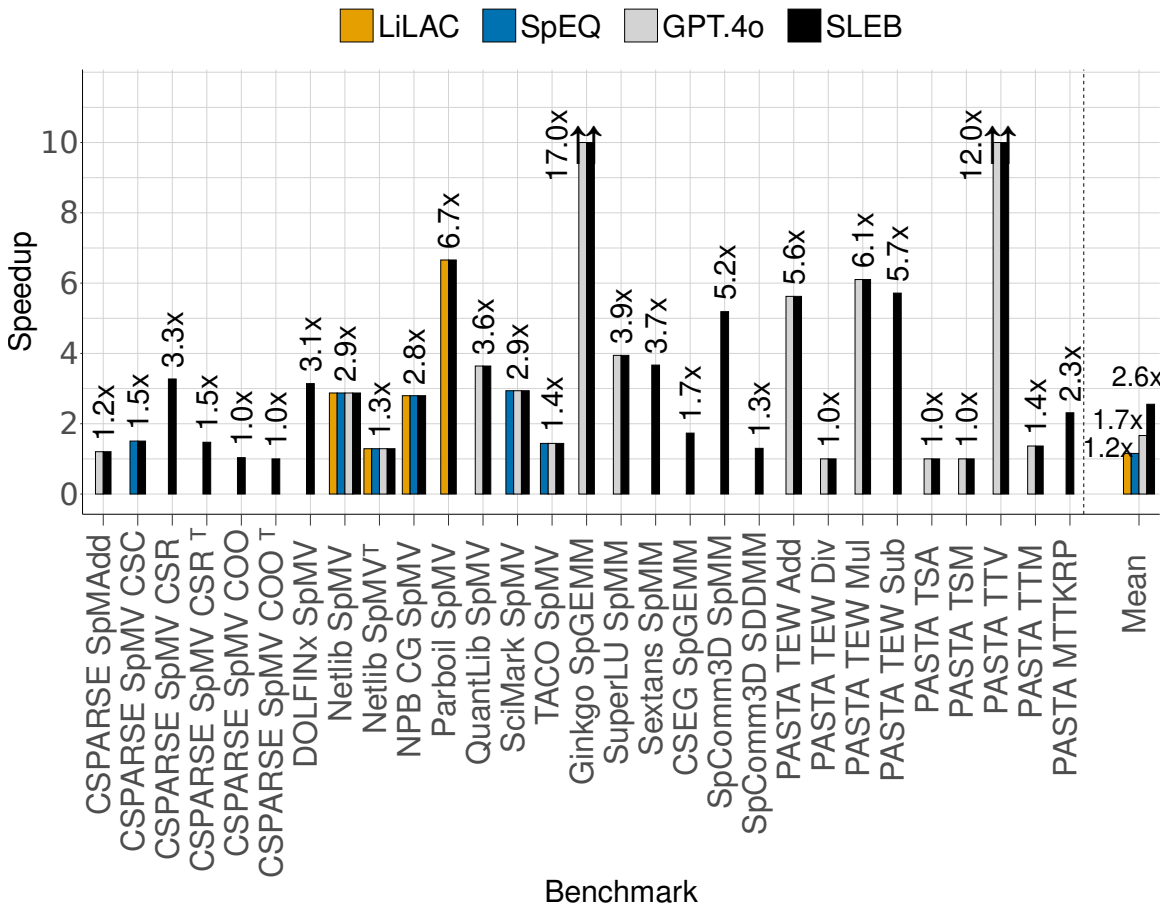


Figure 6.8: Overall speedup of benchmarks lifted to CPU. X-axis lists the benchmarks. Y-axis shows the average speedup over the baseline.

6.5.3 Speedup

6.5.3.1 Speedup on CPU

Figure 6.8 shows the speedup achieved by each method on the CPU for each benchmark together with a geometric mean. For each benchmark, we only show the bars of the methods that can lift it. 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 the PASTA SpTTV and GinkGO SPGEMM kernels, with speedups of 12x and 17x respectively. Programs lifted to MKL achieve good speedup for CSR and CSC. We do not get improvements for SpMV in COO due to MKL not being very optimized for that format.

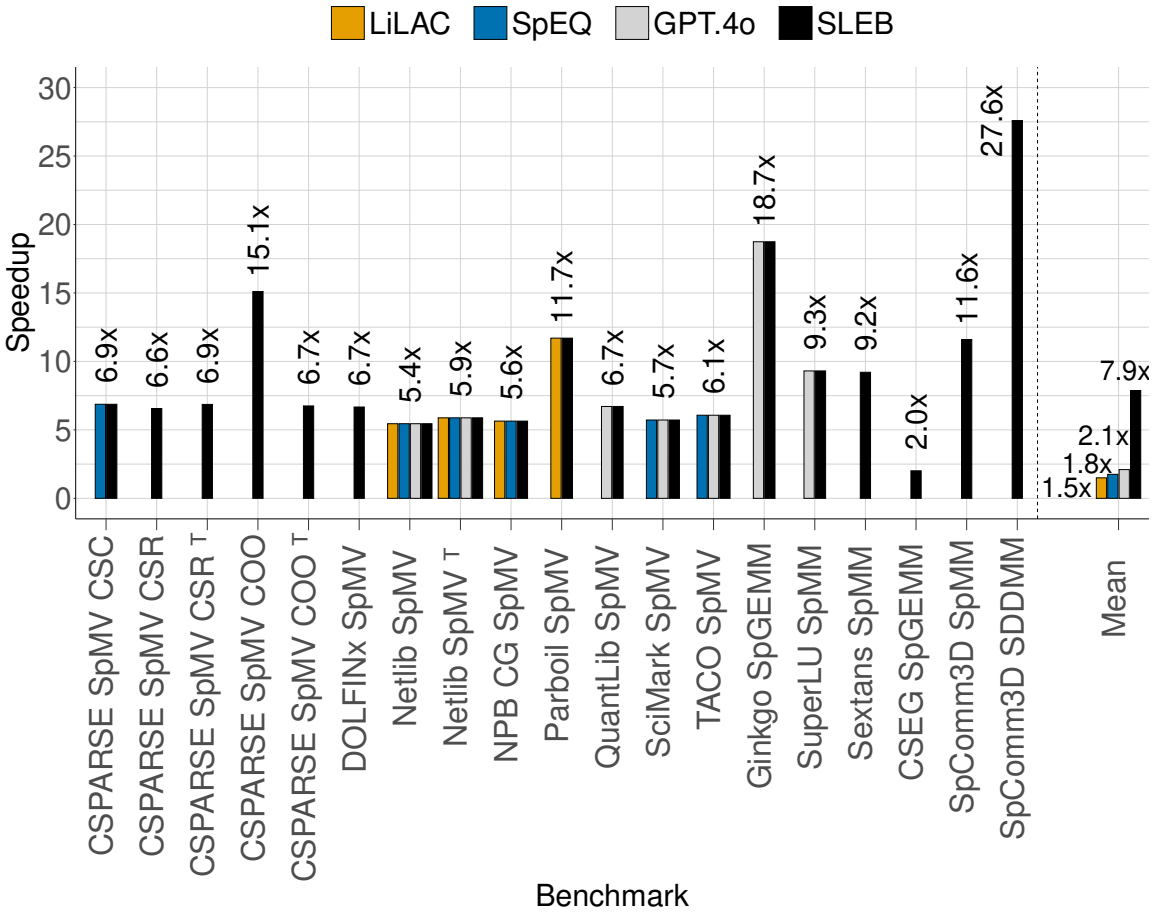


Figure 6.9: Overall speedup of benchmarks lifted to GPU. X-axis lists the benchmarks. Y-axis shows the average speedup over the baseline.

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 output is stored in CSR. We do not achieve speedup for tensor-scalar as the original TSA and TSM benchmarks from PASTA 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 improvements are higher, such as tensor contraction and matrix-matrix kernels.

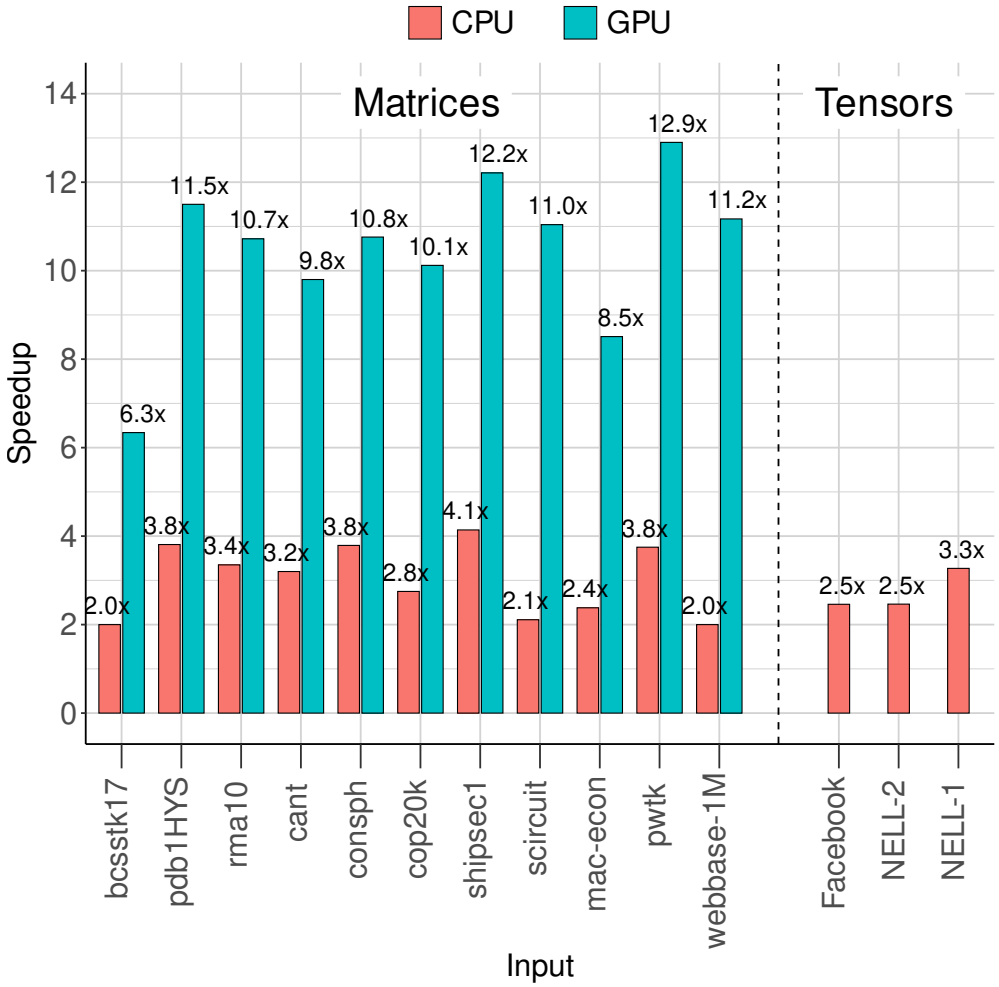


Figure 6.10: Overall speedup of benchmarks lifted to CPU and GPU. X-axis show different inputs sorted by size. Y-axis shows the average speedup over the baseline.

6.5.3.2 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 6.9 depicts the results. Speedup gains are always higher in the GPU than the CPU. The lowest improvement happens in the SpGEMM from CSeg [An and Çatalyürek, 2021], 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.

6.5.3.3 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 6.10 illustrate those results. For the CPU matrix benchmarks, the speedup ranges from 2x on `bcsstk17` and `webbase-1M` to 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`.

6.6 Conclusions

This chapter presents SLEB, a tool that tackles the challenging problem of porting legacy sparse linear algebra code. SLEB uses 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 an LLM.

Chapter 7

Conclusion

This thesis investigated how program lifting can be used to address the code migration challenge by translating legacy programs to domain-specific languages. It presented a series of contributions that combine the strengths of program synthesis, neural machine translation and compiler technology to accelerate legacy tensor programs by porting them to specialized hardware.

We showed that each lifting contribution enables big performance improvement. C2TACO lifts dense tensor programs that become 1.79x and 24.1x faster when ported to multi-core CPU and GPU respectively. The Guess, Measure & Edit framework improves dense lifting and KONRUL, the implementation targeting tensor code is able to lift more complex contractions with higher-order tensors, achieving geomean speedups of 4.07x and 38.30x on CPU and GPU. Finally, SLEB solves the more challenging problem of lifting sparse tensor code to both high-level libraries and DSLs, enabling performance gains of 2.6x and 7.2x on a multi-core CPU and GPU platforms. Summaries of each contribution are provided below.

7.1 Contributions Summary

7.1.1 C2TACO: Lifting with Guided Enumerative Synthesis

C2TACO showcases the use of enumerative bottom-up synthesis to lift C programs to TACO, a tensor DLS. C2TACO implements compiler static analyses to extract features from the legacy programs and use such features as aid during synthesis, restricting exploration. It uses automatically generated input-

output examples to check for correctness. The lifted programs leverage the TACO compiler code generation capabilities to produce optimized dense code for multi-core CPU and GPU and achieve significant results. Although C2TACO analysis are efficient in pruning the search space, its enumerative nature still does not scale for programs with high-order tensors, which motivated a new lifting approach that explores fewer candidates to find the solution.

7.1.2 Guess, Measure & Edit: a Methodology to Lift code Using Compiler Lowering

The Guess, Measure & Edit approach overcomes C2TACO's search limitations with a new method leveraging neural machine translation and compiler lowering. Given a C program, this method uses a language model to guess a initial point in the target search space. Then, both the original program and the guess are lowered to intermediate representation level and are compared with program similarity metrics. Based on this metrics, the initial guess is iteratively edited and compared againt the reference program until deemed correct. We implemented this framework in a tensor lifter named KONRUL. Aside from outperforming C2TACO's search, KONRUL improves dense tensor lifting in two other ways. First, instead of targeting a single language, it lifts the programs to Einsum notation, an agnostic format that can be exported to a variety of languages. Second, KONRUL improves lifting correctness, formally verifying the lifted programs with bounded model checking.

7.1.3 SLEB: Lifting Sparse Code with Neuro-Guided Sketch Synthesis

Both C2TACO and KONRUL handle tensor programs operating on dense-stored data. SLEB (Sparse LiftEr with Binding) moves to a new problem and lifts sparse tensor algebra programs to both libraries and DSLs. Analyzing legacy sparse code is difficult as there are many different data storage formats and idiosyncratic ways of implementing these operations, which makes this lifting task more challenging than handling dense code. SLEB tackles this problem using a neuro-guided approach combining a large language model (LLM) with sketch-based synthesis. Given a legacy implementation of a sparse operation,

SLEB queries an LLM to classify the original code regarding the storage data format and tensor operation. Using this class, SLEB builds an API or DSL program sketch and implements type-based binding to match source code values to the holes in the sketch. This process produces complete programs in highly optimized sparse backends, and the LLM classification combined with sketching makes this technique scalable for complex real-world programs.

7.2 Critical Evaluation

While providing significant contributions to the state-of-the-art of lifting methods, the contributions of this thesis have their own limitations. This section discusses these limitations.

7.2.1 Hard-Wired Heuristics

All the techniques presented in this thesis employ task-specific heuristics to prune the synthesis search space. C2TACO program analysis, KONRUL similarity metrics, and SLEB synthesis constraints are tailored to the tensor domain. All those decisions affect the contributions extensibility to other domains. While this has been demonstrated to be highly effective in the case study of this thesis, these heuristics would have to be adapted to lift programs in other domains. Automatically learning forms of pruning search space remains for future work.

7.2.2 Correctness

C2TACO and SLEB assert correctness via behavior equivalence. Both methods extensively test the lifted programs and the fact the typically tensor programs traverse the inputs completely provide enough coverage to determine the original and the lifted program as correct. However, IO testing is limited to the set of inputs used, and it does not provide formal guarantees that the programs are equivalent for every possible input. KONRUL improves said guarantees with model checking. Still, it proves equivalence in bounded scenario, and the model checker is not able to verify floating-point equivalence for all the benchmarks. Full verification for floating-point benchmarks is a extremely

difficult problem which current verification tools cannot guarantee for complex tensor programs.

7.2.3 Benchmarks

A recurrent problem in program lifting is finding benchmarks. With lifting targets languages that do not support general computation, complete programs cannot be used as source benchmarks. In this thesis we resorted to program classification to identify code regions that can be represented in the DSLs we target. Some tools that were used have limitations which required some benchmarks to be manually adapted. For example, KONRUL uses LLVM's Polly to compute indexing similarity, but Polly cannot reason about constant fields within C structs, hence we had to manually augment augment where necessary. For SLEB, we had change the output type of some benchmarks from sparse to dense because TACO cannot generate code for some operations if the output is sparse.

We were able to gather large-scale and varied sets of benchmarks and strongly evaluated each contribution of this thesis. Nevertheless, even though we used automated classification methods, manual revision and adjustments were still required. This post-process is extremely time-consuming and represents a significant fraction of time for this research. This evidences a need for more automated techniques to search and build benchmarks for program lifting, as the number of such benchmarks is relatively low. A possible direction for future work is automatic generation of benchmarks, however, further research is needed to ensure that such synthetic benchmarks resemble real-world code written by human developers.

7.3 Future Work

The limitations of the contributions of this thesis combined with continuous rapid evolution of computer architecture and program lifting techniques create several research directions for the incoming future.

7.3.1 Expansion to Other Domains

Future research should explore different lifting targets. This thesis presented case studies for tensor algebra, but there exist various distinguished domains that would benefit from lifting. Image processing, graph algorithms, physics-based simulation kernels and security software are some examples of computation that are becoming increasingly important, which creates a demand to deliver high performance for those applications.

7.3.2 Generalization

A clear direction for future work is the development of search aids that are not domain-specific. Currently, program lifting techniques employ heuristics specialized to the target domain. Future work will investigate generic analysis, metrics and cost functions that go beyond tensor DSLs. A promising direction is to use neural networks to automatically learn how to guide search towards the solution, which is in principle domain-agnostic

7.3.3 Targeted Architectures

This thesis shows that enormous speedup gains are available once we port legacy tensor code to multi-core processors and Graphic Process Units. Still, there is a wide range of other architectures that can be targets for lifting. For example, for tensor domains there exists Neural Processing Units (NPU) which also have been shown to provide increased performance. It is an interesting line of work to port code to hardware platforms that are even more specialized and optimized for a given domain.

7.3.4 Usage of Language Models

The rise of language models, including LLMs, has drastically changed research in automatic program translation. Yet, while LLMs are powerful, they are expensive and often ill-suited to low resource languages like DSls. To solve this problem new neural-based lifting techniques will look at replacing LLMs with more self-trained model. Another approach is to fine-tuning large models with task-specific examples, which has been show to boost accuracy.

Since DSL code is scarce, especially for emerging ones, another interesting strand of work is to improve synthetic data generation to produce artificial programs that are representative of human implementations. This would enable automatically creation of training datasets for language models and enhance their usage for code translation.

Nevertheless, the thesis has shown the neural machine translation and language models cannot be relied on their own for lifting tasks, but that they can aid when combined with determinist techniques. Future research should continue to explore novel combinations for efficiently lifting complex programs.

7.4 Summary

This thesis has presented new contributions in program lifting to solve the code migration challenge. Using novel combinations of neural machine translation, program synthesis and compiler technology, this thesis developed techniques able to automatically accelerate legacy programs by translating them to specialized languages and hardware accelerators. C2TACO demonstrates that enumerative synthesis can effectively lift dense tensor programs using guidance. KONRUL improves this lifting overcoming scalability issues with a new methodology, Guess, Measure & Edit, that iterative edit a misdirected guess to quickly synthesize a equivalent program. Finally, SLEB goes beyond dense lifting and employs neural-guided sketch synthesis to lift sparse tensor programs. Together, these contributions significantly advance the program lifting state-of-the-art and demonstrate that lifting is a powerful technique that can build a bridge between legacy code and modern hardware architectures.

Bibliography

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., et al. (2016). Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*.

Abate, A., David, C., Kesseli, P., Kroening, D., and Polgreen, E. (2018). Counterexample guided inductive synthesis modulo theories. In *CAV (1)*, volume 10981 of *Lecture Notes in Computer Science*, pages 270–288. Springer.

Abubaker, N. and Hoefler, T. (2024). Spcomm3d: A framework for enabling sparse communication in 3d sparse kernels.

Achiam, J., Adler, S., Agarwal, S., Ahmad, L., Akkaya, I., Aleman, F. L., Almeida, D., Altenschmidt, J., Altman, S., Anadkat, S., et al. (2023). Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

Ahmad, M. B. S. and Cheung, A. (2017). Optimizing data-intensive applications automatically by leveraging parallel data processing frameworks. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1675–1678.

Ahmad, M. B. S. and Cheung, A. (2018). Automatically leveraging mapreduce frameworks for data-intensive applications. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1205–1220.

Ahmad, M. B. S., Ragan-Kelley, J., Cheung, A., and Kamil, S. (2019). Automatically translating image processing libraries to halide. *ACM Transactions on Graphics (TOG)*, 38(6):1–13.

Ahmad, M. B. S., Root, A. J., Adams, A., Kamil, S., and Cheung, A. (2022). 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.

Ahrens, W., Collin, T. F., Patel, R., Deeds, K., Hong, C., and Amarasinghe, S. (2025). Finch: Sparse and structured tensor programming with control flow. *Proc. ACM Program. Lang.*, 9(OOPSLA1).

Albarghouthi, A., Gulwani, S., and Kincaid, Z. (2013). Recursive program synthesis. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, pages 934–950.

Albrecht, P. F., Garrison, P. E., Graham, S. L., Hyerle, R. H., Ip, P., and Krieg-Brückner, B. (1980). Source-to-source translation: Ada to pascal and pascal to ada. *ACM SIGPLAN Notices*, 15(11):183–193.

Almeida, A., Xavier, L., and Valente, M. T. (2024). Automatic library migration using large language models: First results. In *Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 427–433.

Alur, R., Bodík, R., Juniwal, G., Martin, M. M. K., Raghothaman, M., Seshia, S. A., Singh, R., Solar-Lezama, A., Torlak, E., and Udupa, A. (2013). Syntax-guided synthesis. In *FMCAD*, pages 1–8. IEEE.

An, X. and Çatalyürek, U. V. (2021). Column-segmented sparse matrix-matrix multiplication. In *HiPC21: 28th IEEE International Conference on High Performance Computing, Data, & Analytics*.

Angstadt, K., Jeannin, J.-B., and Weimer, W. (2020). Accelerating legacy string kernels via bounded automata learning. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 235–249.

Anzt, H., Cojean, T., Flegar, G., Göbel, F., Grützmacher, T., Nayak, P., Ribizel, T., Tsai, Y. M., and Quintana-Ortí, E. S. (2022). Ginkgo: A Modern Linear

Operator Algebra Framework for High Performance Computing. *ACM Transactions on Mathematical Software*, 48(1):2:1–2:33.

Armengol-Estapé, J. and O'Boyle, M. F. P. (2021). Learning C to x86 translation: An experiment in neural compilation. *CoRR*, abs/2108.07639.

Armengol-Estapé, J., Rocha, R. C., Woodruff, J., Minervini, P., and O'Boyle, M. F. (2024). Forklift: An extensible neural lifter. *Conference on Language Modeling*.

Armengol-Estapé, J., Woodruff, J., Cummins, C., and O'Boyle, M. F. (2024). SLaDe: A portable small language model decompiler for optimized assembler. *CGO*.

Backus, J. W. (1959). The syntax and the semantics of the proposed international algebraic language of the zurich acm-gamm conference. In *ICIP Proceedings*, pages 125–132.

Bahdanau, D., Cho, K., and Bengio, Y. (2014). Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*.

Baillie, C. (1986). A general fortran to c translator. *Computer Physics Communications*, 41(2-3):409–414.

Balog, M., Gaunt, A. L., Brockschmidt, M., Nowozin, S., and Tarlow, D. (2016). Deepcoder: Learning to write programs. *arXiv preprint arXiv:1611.01989*.

Bansal, M., Hsu, O., Olukotun, K., and Kjolstad, F. (2023). Mosaic: An interoperable compiler for tensor algebra. *PLDI*.

Baratta, I. A., Dean, J. P., Dokken, J. S., Habera, M., HALE, J., Richardson, C. N., Rognes, M. E., Scroggs, M. W., Sime, N., and Wells, G. N. (2023). Dolfinx: the next generation fenics problem solving environment.

Barbosa, H., Barrett, C., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A., et al. (2022). cvc5: A versatile and industrial-strength smt solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 415–442. Springer.

Barke, S., Anaya Gonzalez, E., Kasibatla, S. R., Berg-Kirkpatrick, T., and Polikarpova, N. (2024). Hysynth: Context-free llm approximation for guiding program synthesis. *Advances in Neural Information Processing Systems*, 37:15612–15645.

Bavishi, R., Lemieux, C., Fox, R., Sen, K., and Stoica, I. (2019). Autopandas: neural-backed generators for program synthesis. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–27.

Bayer, M. (2012). Sqlalchemy. *The architecture of open source applications*, 2.

Behnel, S., Bradshaw, R., Citro, C., Dalcin, L., Seljebotn, D. S., and Smith, K. (2010). Cython: The best of both worlds. *Computing in Science & Engineering*, 13(2):31–39.

Benabderrahmane, M.-W., Pouchet, L.-N., Cohen, A., and Bastoul, C. (2010). The polyhedral model is more widely applicable than you think. In *International Conference on Compiler Construction*, pages 283–303. Springer.

Benzaken, V., Contejean, É., Hachmaoui, M. H., Keller, C., Mandel, L., Shinnar, A., and Siméon, J. (2022). Translating canonical sql to imperative code in coq. *Proceedings of the ACM on Programming Languages*, 6(OOPSLA1):1–27.

Bhatia, S., Kohli, S., Seshia, S. A., and Cheung, A. (2023). Building code transpilers for domain-specific languages using program synthesis (experience paper). In *ECOOP*, volume 263 of *LIPIcs*, pages 38:1–38:30. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

Bhatia, S., Qiu, J., Hasabnis, N., Seshia, S. A., and Cheung, A. (2024). Verified code transpilation with llms. *Advances in Neural Information Processing Systems*, 37:41394–41424.

Blacher, M., Klaus, J., Staudt, C., Laue, S., Leis, V., and Geisen, J. (2023). Efficient and portable einstein summation in SQL. *Proceedings of the ACM on Management of Data*, 1:1–19.

Blackford, L. S., Petitet, A., Pozo, R., Remington, K., Whaley, R. C., Demmel, J., Dongarra, J., Duff, I., Hammarling, S., Henry, G., et al. (2002). An up-

dated set of basic linear algebra subprograms (blas). *ACM Transactions on Mathematical Software*, 28(2):135–151.

Blech, E., Grishchenko, A., Kniazkov, I., Liang, G., Serebrennikov, O., Tatarnikov, A., Volkhontseva, P., and Yakimets, K. (2021). Patternika: A pattern-mining-based tool for automatic library migration. In *2021 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 333–338. IEEE.

Bradbury, J., Frostig, R., Hawkins, P., Johnson, M. J., Leary, C., Maclaurin, D., Necula, G., Paszke, A., VanderPlas, J., Wanderman-Milne, S., and Zhang, Q. (2018). JAX: composable transformations of Python+NumPy programs.

Brauckmann, A., Jaulmes, L., de Souza Magalhães, J. W., Polgreen, E., and O'Boyle, M. F. (2025). Tensorize: Fast synthesis of tensor programs from legacy code using symbolic tracing, sketching and solving. In *ACM/IEEE CGO*.

Brauckmann, A., Polgreen, E., Grosser, T., and O'Boyle, M. F. (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)*, pages 39–50. IEEE.

Calotoiu, A., Ben-Nun, T., Kwasniewski, G., de Fine Licht, J., Schneider, T., Schaad, P., and Hoefler, T. (2022). Lifting c semantics for dataflow optimization. In *Proceedings of the 36th ACM International Conference on Supercomputing*, pages 1–13.

Cao, Y., Liang, R., Chen, K., and Hu, P. (2022). Boosting neural networks to decompile optimized binaries. In *Proceedings of the 38th Annual Computer Security Applications Conference*, pages 508–518.

Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S., and Tzoumas, K. (2015). Apache flink: Stream and batch processing in a single engine. *The Bulletin of the Technical Committee on Data Engineering*, 38(4).

Caulfield, B., Rabe, M. N., Seshia, S. A., and Tripakis, S. (2015). What's decidable about syntax-guided synthesis? *CoRR*, abs/1510.08393.

Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J. W., Lee, S.-H., and Skadron, K. (2009). Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE international symposium on workload characterization (IISWC)*, pages 44–54. Ieee.

Chelini, L., Drebes, A., Zinenko, O., Cohen, A., Vasilache, N., Grosser, T., and Corporaal, H. (2021). Progressive raising in multi-level ir. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 15–26. IEEE.

Chen, T., Moreau, T., Jiang, Z., Zheng, L., Yan, E., Cowan, M., Shen, H., Wang, L., Hu, Y., Ceze, L., et al. (2018). Tvm: an automated end-to-end optimizing compiler for deep learning. In *Proceedings of the 13th USENIX conference on Operating Systems Design and Implementation*, pages 579–594.

Chen, Z., Kommrusch, S., Tufano, M., Pouchet, L.-N., Poshyvanyk, D., and Monperrus, M. (2019). Sequencer: Sequence-to-sequence learning for end-to-end program repair. *IEEE Transactions on Software Engineering*, 47(9):1943–1959.

Cheung, A., Solar-Lezama, A., and Madden, S. (2013). Optimizing database-backed applications with query synthesis. *ACM SIGPLAN Notices*, 48(6):3–14.

Cho, K., Van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., and Bengio, Y. (2014). Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*.

Chomsky, N. (1956). Three models for the description of language. *IRE Transactions on information theory*, 2(3):113–124.

Collie, B., Ginsbach, P., Woodruff, J., Rajan, A., and O'Boyle, M. F. (2020a). M3: Semantic api migrations. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pages 90–102.

Collie, B. and O'Boyle, M. F. (2021). Program lifting using gray-box behavior. In *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 60–74. IEEE.

Collie, B., Woodruff, J., and O'Boyle, M. F. (2020b). 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.

Cooper, K. D. and Torczon, L. (2011). *Engineering a compiler*. Elsevier.

Cordy, J. R. (2006). The txl source transformation language. *Science of Computer Programming*, 61(3):190–210.

Cover, T. and Hart, P. (1967). Nearest neighbor pattern classification. *IEEE transactions on information theory*, 13(1):21–27.

Cross, A. W., Bishop, L. S., Smolin, J. A., and Gambetta, J. M. (2017). Open quantum assembly language. *arXiv preprint arXiv:1707.03429*.

Davis, T. A. (2006). *Direct methods for sparse linear systems*. SIAM.

De Carvalho, J. P., Kuzma, B., Korostelev, I., Amaral, J. N., Barton, C., Moreira, J., and Araujo, G. (2021). Kernelfarer: replacing native-code idioms with high-performance library calls. *ACM Transactions On Architecture And Code Optimization (TACO)*, 18(3):1–22.

de Moura, L. M. and Bjørner, N. S. (2008). Z3: an efficient SMT solver. In *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer.

Dean, J. and Ghemawat, S. (2008). Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113.

Dennard, R., Gaensslen, F., Yu, H.-N., Rideout, V., Bassous, E., and LeBlanc, A. (1974). Design of ion-implanted mosfet's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268.

Dias, A., Sundararajah, K., Saumya, C., and Kulkarni, M. (2022). SparseLNR: Accelerating sparse tensor computations using loop nest restructuring. *ICS*.

Dong, S., Wen, Y., Bi, J., Huang, D., Guo, J., Xu, J., Xu, R., Song, X., Hao, Y., Li, L., Zhou, X., Chen, T., Guo, Q., and Chen, Y. (2025). Qimeng-xpiler:

transcompiling tensor programs for deep learning systems with a neural-symbolic approach. In *Proceedings of the 19th USENIX Conference on Operating Systems Design and Implementation*, OSDI '25, USA. USENIX Association.

Dongarra, J., Eijkhout, V., and Vorst, H. v. d. (2001). An iterative solver benchmark. *Scientific Programming*, 9(4):223–231.

Drebes, A., Chelini, L., Zineko, O., Cohen, A., Corporaal, H., Grosser, T., Vadivel, K., and Vasilache, N. (2020). TC-CIM: Empowering tensor comprehensions for computing-in-memory. *IMPACT*.

Dyer, T., Altuntas, A., and Baugh, J. (2019). Bounded verification of sparse matrix computations. In *2019 IEEE/ACM 3rd International Workshop on Software Correctness for HPC Applications (Correctness)*, pages 36–43.

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

Elliott, A. S., Ruef, A., Hicks, M., and Tarditi, D. (2018). Checked c: Making c safe by extension. In *2018 IEEE Cybersecurity Development (SecDev)*, pages 53–60. IEEE.

Emre, M., Schroeder, R., Dewey, K., and Hardekopf, B. (2021). Translating c to safer rust. *Proceedings of the ACM on Programming Languages*, 5(OOPSLA):1–29.

Esmaeilzadeh, H., Blem, E., St. Amant, R., Sankaralingam, K., and Burger, D. (2011). Dark silicon and the end of multicore scaling. In *Proceedings of the 38th annual international symposium on Computer architecture*, pages 365–376.

Espindola, V., Zago, L., Yviquel, H., and Araujo, G. (2023). Source matching and rewriting for mlir using string-based automata. *ACM Transactions on Architecture and Code Optimization*, 20(2):1–26.

Fedyukovich, G., Ahmad, M. B. S., and Bodik, R. (2017). Gradual synthesis for static parallelization of single-pass array-processing programs. *ACM SIGPLAN Notices*, 52(6):572–585.

Feldman, S. I. (1990). A fortran to c converter. In *ACM SIGPLAN Fortran Forum*, volume 9, pages 21–22. ACM New York, NY, USA.

Feser, J. K., Dillig, I., and Solar-Lezama, A. (2023). Inductive program synthesis guided by observational program similarity. *Proc. ACM Program. Lang.*, 7(OOPSLA2):912–940.

Fix, E. (1985). *Discriminatory analysis: nonparametric discrimination, consistency properties*, volume 1. USAF school of Aviation Medicine.

Fox, P., Hall, A., and Schryer, N. L. (1978). The port mathematical subroutine library. *ACM Transactions on Mathematical Software (TOMS)*, 4(2):104–126.

Franke, B. and O'Boyle, M. (2003). Array recovery and high-level transformations for dsp applications. *ACM Transactions on Embedded Computing Systems (TECS)*, 2(2):132–162.

Fu, C., Chen, H., Liu, H., Cheng, X., and Tian, Y. (2019). Coda: An end-to-end neural program decompiler. *NeurIPs*.

Gage, P. (1994). A new algorithm for data compression. *The C Users Journal*, 12(2):23–38.

Gao, X., Radhakrishna, A., Soares, G., Shariffdeen, R., Gulwani, S., and Roychoudhury, A. (2021). Apifix: output-oriented program synthesis for combating breaking changes in libraries. *Proceedings of the ACM on Programming Languages*, 5(OOPSLA):1–27.

Gerard, B., Grosser, T., and Kong, M. (2022). Qrane: lifting qasm programs to an affine ir. In *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction*, pages 15–28.

Ghorbani, M., Huot, M., Hashemian, S., and Shaikha, A. (2023). Compiling structured tensor algebra. *OOPSLA*.

Ginsbach, P. (2020). `https://github.com/ginsbach/llvm/tree/linearalgebra`, https://github.com/ginsbach/clang/tree/research.

Ginsbach, P., Collie, B., and O'Boyle, M. F. (2020). Automatically harnessing sparse acceleration. In *Proceedings of the 29th International Conference on Compiler Construction*, pages 179–190.

Ginsbach, P., Remmelg, T., Steuwer, M., Bodin, B., Dubach, C., and O'Boyle, M. F. (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*, pages 139–153.

Gokhale, S., Turcotte, A., and Tip, F. (2021). Automatic migration from synchronous to asynchronous javascript apis. *Proceedings of the ACM on Programming Languages*, 5(OOPSLA):1–27.

Goldberg, D. (1991). "what every computer scientist should know about floating-point arithmetic". *ACM Comput. Surv.*, 23(3):413.

Google (2023a). Google gemini. `https://gemini.google.com/app`.

Google (2023b). Mathfu. `https://github.com/google/mathfu`.

Grauer-Gray, S. and Pouchet, L.-N. (2012). Polybench/gpu: Implementation of poly-bench codes for gpu processing. *URL: http://www. cs. ucla. edu/-pouchet/software/polybench*.

Gregory, G. and Bauer, C. (2015). *Java persistence with hibernate*. Simon and Schuster.

Grosser, T., Groesslinger, A., and Lengauer, C. (2012). Polly—performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters*, 22(04):1250010.

Gu, X., Chen, M., Lin, Y., Hu, Y., Zhang, H., Wan, C., Wei, Z., Xu, Y., and Wang, J. (2025). On the effectiveness of large language models in domain-specific code generation. *ACM Transactions on Software Engineering and Methodology*, 34(3):1–22.

Gu, X., Zhang, H., Zhang, D., and Kim, S. (2017). Deepam: Migrate apis with multi-modal sequence to sequence learning. *arXiv preprint arXiv:1704.07734*.

Guennebaud, G., Jacob, B., et al. (2010). Eigen. https://libeigen.gitlab.io.

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

Gulwani, S., Polozov, O., and Singh, R. (2017). Program synthesis. *Foundations and trends in programming languages*, 4(1-2):1–119.

Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., van Kerkwijk, M. H., Brett, M., Haldane, A., del Río, J. F., Wiebe, M., Peterson, P., Gérard-Marchant, P., Sheppard, K., Reddy, T., Weckesser, W., Abbasi, H., Gohlke, C., and Oliphant, T. E. (2020). Array programming with NumPy. *Nature*, 585(7825):357–362.

Hasabnis, N. and Sekar, R. (2016). Lifting assembly to intermediate representation: A novel approach leveraging compilers. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 311–324.

Hennessy, J. L. and Patterson, D. A. (2019). A new golden age for computer architecture. *Commun. ACM*, 62(2):48–60.

Ho, T. K. (1995). Random decision forests. In *Proceedings of 3rd international conference on document analysis and recognition*, volume 1, pages 278–282. IEEE.

Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580.

Hong, H., Zhang, J., Zhang, Y., Wan, Y., and Sui, Y. (2021). Fix-filter-fix: Intuitively connect any models for effective bug fixing. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 3495–3504.

Hong, J. and Ryu, S. (2023). Concrat: An automatic c-to-rust lock api translator for concurrent programs. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 716–728. IEEE.

Hong, J. and Ryu, S. (2024). Don't write, but return: Replacing output parameters with algebraic data types in c-to-rust translation. *Proceedings of the ACM on Programming Languages*, 8(PLDI):716–740.

Hopcroft, J. E. and Ullman, J. D. (1979). *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, Massachusetts.

Hsu, O., Rucker, A., Zhao, T., Olukotun, K., and Kjolstad, F. (2022). Stardust: Compiling sparse tensor algebra to a reconfigurable dataflow architecture. *CoRR*. Available at `https://arxiv.org/abs/2211.03251`.

Huet, G., Kahn, G., and Paulin-Mohring, C. (1997). The coq proof assistant a tutorial. *Rapport Technique*, 178:113.

Huijsman, R., van Katwijk, J., Pronk, C., and Toetenel, W. (1987). Translating algol 60 programs into ada. *ACM SIGAda Ada Letters*, 7(5):42–50.

Imambi, S., Prakash, K. B., and Kanagachidambaresan, G. (2021). Pytorch. *Programming with TensorFlow: Solution for Edge Computing Applications*, pages 87–104.

Inc, G. and Immunant, I. (2020). c2rust. https://github.com/immunant/c2rust.

Instruments, T. (2023). Texas instrument digital signal processing (dsp) library for msp430 microcontrollers. `https://www.ti.com/tool/MSP-DSPLIB`.

Intel (2025). Intel® oneapi math kernel library. https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl.html.

James, M. B., Guo, Z., Wang, Z., Doshi, S., Peleg, H., Jhala, R., and Polikarpova, N. (2020). Digging for fold: synthesis-aided api discovery for haskell. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–27.

JAX (Accessed 2024). JAX: High performance array computing. Available at `https://jax.readthedocs.io/en/latest/index.html`.

Jha, S., Gulwani, S., Seshia, S. A., and Tiwari, A. (2010). Oracle-guided component-based program synthesis. In *ICSE (1)*, pages 215–224. ACM.

Jia, L., Luo, Z., Lu, L., and Liang, Y. (2021). TensorLib: A spatial accelerator generation framework for tensor algebra. *DAC*.

Jia, L., Luo, Z., Lu, L., and Liang, Y. (2023). Automatic generation of spatial accelerator for tensor algebra. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 42(6).

Kamil, S., Cheung, A., Itzhaky, S., and Solar-Lezama, A. (2016). Verified lifting of stencil computations. *ACM SIGPLAN Notices*, 51(6):711–726.

Kang, Y., Wang, Z., Zhang, H., Chen, J., and Hanmo, Y. (2021). APIRecX: Cross-library API recommendation via pre-trained language model. *EMNLP*.

Karaivanov, S., Raychev, V., and Vechev, M. (2014). Phrase-based statistical translation of programming languages. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, pages 173–184.

Katz, D. S., Ruchti, J., and Schulte, E. (2018). Using recurrent neural networks for decompilation. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 346–356.

Katz, O., Olshaker, Y., Goldberg, Y., and Yahav, E. (2019). Towards neural decompilation. *CoRR*.

Kim, Y. and Kim, H. (2019). Translating CUDA to OpenCL for hardware generation using neural machine translation. *CGO*.

Kimura, K., Sekiguchi, A., Choudhary, S., and Uehara, T. (2018). A javascript transpiler for escaping from complicated usage of cloud services and apis. In *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*, pages 69–78. IEEE.

Kingma, D. P. and Ba, J. (2017). Adam: A method for stochastic optimization.

Kjolstad, F., Kamil, S., Chou, S., Lugato, D., and Amarasinghe, S. (2017). The tensor algebra compiler. *OOPSLA*.

Klaus, J., Blacher, M., and Giesen, J. (2023). Compiling tensor expressions into einsum. In *International Conference on Computational Science*, pages 129–136. Springer.

Koehn, P., Hoang, H., Birch, A., Callison-Burch, C., Federico, M., Bertoldi, N., Cowan, B., Shen, W., Moran, C., Zens, R., et al. (2007). Moses: Open source toolkit for statistical machine translation. In *Proceedings of the 45th annual meeting of the association for computational linguistics companion volume proceedings of the demo and poster sessions*, pages 177–180.

Koehn, P., Och, F. J., and Marcu, D. (2003). Statistical phrase-based translation. In *Proceedings of the 2003 human language technology conference of the North American chapter of the Association for computational linguistics*, pages 127–133.

Kroening, D. and Tautschnig, M. (2014). CBMC - C bounded model checker - (competition contribution). In *TACAS*, volume 8413 of *Lecture Notes in Computer Science*, pages 389–391. Springer.

Kudo, T. and Richardson, J. (2018). Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. In *EMNLP (Demonstration)*, pages 66–71. Association for Computational Linguistics.

Lacomis, J., Yin, P., Schwartz, E., Allamanis, M., Le Goues, C., Neubig, G., and Vasilescu, B. (2019). Dire: A neural approach to decompiled identifier namings. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 628–639. IEEE.

Laird, A. (2024). Speq: Translation of sparse codes using equivalences. `https://zenodo.org/records/10906216`.

Laird, A., Liu, B., Bjørner, N., and Dehnavi, M. M. (2024). SpEQ: Translation of sparse codes using equivalences. *Proc. ACM Program. Lang.*, 8(PLDI).

Lattner, C. and Adve, V. (2004). Llvm: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004.*, pages 75–86. IEEE.

Lattner, C., Amini, M., Bondhugula, U., Cohen, A., Davis, A., Pienaar, J., Riddle, R., Shpeisman, T., Vasilache, N., and Zinenko, O. (2021). Mlir: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 2–14. IEEE.

Le, X.-B. D., Lo, D., and Le Goues, C. (2016). Empirical study on synthesis engines for semantics-based program repair. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 423–427. IEEE.

Lee, C., Mahmoud, A., Kurek, M., Campanoni, S., Brooks, D., Chong, S., Wei, G.-Y., and Rush, A. M. (2023). Guess & sketch: Language model guided transpilation. *arXiv preprint arXiv:2309.14396*.

Lee, S., Min, S.-J., and Eigenmann, R. (2009). Openmp to gpgpu: a compiler framework for automatic translation and optimization. *ACM Sigplan Notices*, 44(4):101–110.

Lee, W., Heo, K., Alur, R., and Naik, M. (2018). Accelerating search-based program synthesis using learned probabilistic models. In *PLDI*, pages 436–449. ACM.

Leino, K. R. M. (2010). Dafny: An automatic program verifier for functional correctness. In *International conference on logic for programming artificial intelligence and reasoning*, pages 348–370. Springer.

Levenshtein, V. I. (1966). Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady*, 10:707.

Lezama, A. S. (2008). *Program synthesis by sketching*. PhD thesis, PhD thesis, EECS Department, University of California, Berkeley.

Lhoták, O. (2003). Spark: A flexible points-to analysis framework for java.

Li, H. (2022). Language models: past, present, and future. *Communications of the ACM*, 65(7):56–63.

Li, J., Ma, Y., Wu, X., Li, A., and Barker, K. (2019). Pasta: a parallel sparse tensor algorithm benchmark suite. *CCF Transactions on High Performance Computing*, 1(2):111–130.

Li, X. S., Demmel, J., Gilbert, J., Grigori, L., and Shao, M. (2011). *SuperLU*, pages 1955–1962. Springer US, Boston, MA.

Li, Y., Magalhães, J. W. d. S., Brauckmann, A., O'Boyle, M. F., and Polgreen, E. (2025). Guided tensor lifting. *Proceedings of the ACM on Programming Languages*, 9(PLDI):1984–2006.

Li, Y., Parsert, J., and Polgreen, E. (2024). Guiding enumerative program synthesis with large language models. In *International Conference on Computer Aided Verification*, pages 280–301. Springer.

Ling, M., Yu, Y., Wu, H., Wang, Y., Cordy, J. R., and Hassan, A. E. (2022). In rust we trust: a transpiler from unsafe c to safer rust. In *Proceedings of the ACM/IEEE 44th international conference on software engineering: companion proceedings*, pages 354–355.

Löff, J., Griebler, D., Mencagli, G., Araujo, G., Torquati, M., Danelutto, M., and Fernandes, L. G. (2021). The nas parallel benchmarks for evaluating c++ parallel programming frameworks on shared-memory architectures. *Future Generation Computer Systems*, 125:743–757.

Machiry, A., Kastner, J., McCutchen, M., Eline, A., Headley, K., and Hicks, M. (2022). C to checked c by 3c. *Proceedings of the ACM on Programming Languages*, 6(OOPSLA1):1–29.

Magalhães, J. W. d. S., Woodruff, J., Polgreen, E., and O'Boyle, M. F. (2023). C2TACO: Lifting tensor code to TACO. *GPCE*.

Mandal, S., Chethan, A., Janfaza, V., Mahmud, S. M. F., Anderson, T. A., Turek, J., Tithi, J. J., and Muzahid, A. (2023). Large language models based automatic synthesis of software specifications. *CoRR*. Available at `https://arxiv.org/pdf/2304.09181.pdf`.

Mariano, B., Chen, Y., Feng, Y., Durrett, G., and Dillig, I. (2022). Automated transpilation of imperative to functional code using neural-guided program synthesis. *Proceedings of the ACM on Programming Languages*, 6(OOPSLA1):1–27.

Martínez, P. A., Woodruff, J., Armengol-Estapé, J., Bernabé, G., García, J. M., and O'Boyle, M. F. (2023). Matching linear algebra and tensor code to specialized hardware accelerators. In *Proceedings of the 32nd ACM SIGPLAN International Conference on Compiler Construction*, pages 85–97.

McKinney, W. et al. (2011). pandas: a foundational python library for data analysis and statistics. *Python for high performance and scientific computing*, 14(9):1–9.

Mendis, C., Bosboom, J., Wu, K., Kamil, S., Ragan-Kelley, J., Paris, S., Zhao, Q., and Amarasinghe, S. (2015). Helium: Lifting high-performance stencil kernels from stripped x86 binaries to halide dsl code. In *Proceedings of the*

36th ACM SIGPLAN Conference on Programming Language Design and Implementation, pages 391–402.

Mikolov, T., Sutskever, I., Chen, K., Corrado, G. S., and Dean, J. (2013). Distributed representations of words and phrases and their compositionality. *Advances in neural information processing systems*, 26.

Navarro, G. (2001). A guided tour to approximate string matching. *ACM Comput. Surv.*, 33(1):31–88.

Netlib (2025a). Clapack.

Netlib (2025b). Lapack.

Ni, A., Ramos, D., Yang, A., Lynce, I., Manquinho, V., Martins, R., and Le Goues, C. (2021). Soar: A synthesis approach for data science api refactoring. icse (2021).

Nishida, Y., Bhatia, S., Laddad, S., Genc, H., Shao, Y. S., and Cheung, A. (2023). Code transpilation for hardware accelerators. *CoRR*. Available at `https://arxiv.org/pdf/2308.06410.pdf`.

Nishino, R. and Loomis, S. H. C. (2017). Cupy: A numpy-compatible library for nvidia gpu calculations. *31st confernce on neural information processing systems*, 151(7).

NVIDIA (2025a). cublas: Basic linear algebra on nvidia gpus. https://developer.nvidia.com/cublas.

NVIDIA (2025b). cusparse [n. d.]. basic linear algebra for sparse matrices on nvidia gpus. https://developer.nvidia.com/cusparse.

NVIDIA (2025c). cutensor: Tensor linear algebra on nvidia gpus. https://developer.nvidia.com/cutensor.

NVIDIA (2025d). Sdk manager. https://developer.nvidia.com/sdk-manager.

O'Boyle, M. F. P. and Knijnenburg, P. M. W. (2002). Integrating loop and data transformations for global optimization. *J. Parallel Distributed Comput.*, 62(4):563–590.

Odena, A., Shi, K., Bieber, D., Singh, R., Sutton, C., and Dai, H. (2020). Bustle: Bottom-up program synthesis through learning-guided exploration. *arXiv preprint arXiv:2007.14381*.

OpenAI (2022). Chatgpt. `https://openai.com/chatgpt`.

Oracle (2025). Package java.util.stream.

Ott, M., Edunov, S., Baevski, A., Fan, A., Gross, S., Ng, N., Grangier, D., and Auli, M. (2019). fairseq: A fast, extensible toolkit for sequence modeling. *arXiv preprint arXiv:1904.01038*.

Pan, R., Ibrahimzada, A. R., Krishna, R., Sankar, D., Wassi, L. P., Merler, M., Sobolev, B., Pavuluri, R., Sinha, S., and Jabbarvand, R. (2024). Lost in translation: A study of bugs introduced by large language models while translating code. *ICSE*.

Phan, H. D., Nguyen, A. T., Nguyen, T. D., and Nguyen, T. N. (2017). Statistical migration of api usages. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 47–50. IEEE.

Phothilimthana, P. M., Thakur, A., Bodik, R., and Dhurjati, D. (2016). Scaling up superoptimization. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 297–310.

Pizzolotto, D. and Inoue, K. (2021). Identifying compiler and optimization level in binary code from multipler architectures. *IEEE Access*.

Pozo, R. and Miller, B. (2025). Scimark 2.0. `https://math.nist.gov/scimark2/`.

Priya, S., Su, Y., Bao, Y., Zhou, X., Vizel, Y., and Gurfinkel, A. (2022). Bounded model checking for llvm. In *# PLACEHOLDER_PARENT_METADATA_VALUE#*, pages 214–224. TU Wien Academic Press.

Qiu, J., Cai, C., Bhatia, S., Hasabnis, N., Seshia, S. A., and Cheung, A. (2024a). Tenspiler: A verified lifting-based compiler for tensor operations. In *38th European Conference on Object-Oriented Programming (ECOOP 2024)*.

Qiu, J., Cai, C., Bhatia, S., Hasabnis, N., Seshia, S. A., and Cheung, A. (2024b). Tenspiler: A Verified-Lifting-Based Compiler for Tensor Operations (Artifact). *Dagstuhl Artifacts Series*, 10(2).

QuantLib (2025). Quantlib: a free/open-source library for quantitative finance. `https://www.quantlib.org/`.

Radoi, C., Fink, S. J., Rabbah, R., and Sridharan, M. (2014). Translating imperative code to mapreduce. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, pages 909–927.

Ragan-Kelley, J., Barnes, C., Adams, A., Paris, S., Durand, F., and Amarasinghe, S. (2013). Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *Acm Sigplan Notices*, 48(6):519–530.

Raje, S., Xu, Y., Rountev, A., Valeev, E. F., and Sadayappan, S. (2024). CoNST: Code generator for sparse tensor networks. *CoRR*. Available at `https://arxiv.org/html/2401.04836v1`.

Recoules, F., Bardin, S., Bonichon, R., Mounier, L., and Potet, M.-L. (2019). Get rid of inline assembly through verification-oriented lifting. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 577–589. IEEE.

Reynolds, A., Deters, M., Kuncak, V., Tinelli, C., and Barrett, C. (2015). Counterexample-guided quantifier instantiation for synthesis in smt. In *International Conference on Computer Aided Verification*, pages 198–216. Springer.

Rocha, R. C., Sprokholt, D., Fink, M., Gouicem, R., Spink, T., Chakraborty, S., and Bhatotia, P. (2022). Lasagne: A static binary translator for weak memory model architectures. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 888–902.

Rodríguez, G., Andión, J. M., Kandemir, M. T., and Touriño, J. (2016). Trace-based affine reconstruction of codes. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, pages 139–149.

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

Roziere, B., Lachaux, M.-A., Chanussot, L., and Lample, G. (2020). Unsupervised translation of programming languages. *NeurIPs*.

Rozière, B., Zhang, J. M., Charton, F., Harman, M., Synnaeve, G., and Lample, G. (2022). Leveraging automated unit tests for unsupervised code translation.

Saghir, M. A. (1998). *Application-specific instruction-set architectures for embedded DSP applications*. Citeseer.

Santos, E. A., Campbell, J. C., Patel, D., Hindle, A., and Amaral, J. N. (2018). Syntax and sensibility: Using language models to detect and correct syntax errors. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 311–322. IEEE.

Schiavio, F., Rosà, A., and Binder, W. (2022). Sql to stream with s2s: An automatic benchmark generator for the java stream api. In *Proceedings of the 21st ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, pages 179–186.

Schryer, N. (1981). *A test of a computer's floating-point arithmetic unit*. Bell Laboratories. Computing Science.

Senanayake, R., Hong, C., Wang, S., Amalee, W., Chou, S., Kamil, S., Amarasinghe, S., and Kjolstad, F. (2020). A sparse iteration space transformation framework for sparse tensor algebra. *OOPSLA*.

Sennrich, R., Haddow, B., and Birch, A. (2015). Neural machine translation of rare words with subword units. *arXiv preprint arXiv:1508.07909*.

Shi, K., Bieber, D., and Singh, R. (2022). Tf-coder: Program synthesis for tensor manipulations. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 44(2):1–36.

Sidiropoulos, N. D., De Lathauwer, L., Fu, X., Huang, K., Papalexakis, E. E., and Faloutsos, C. (2017). Tensor decomposition for signal processing and machine learning. *IEEE Transactions on signal processing*, 65(13):3551–3582.

Singh, R., Singh, R., Xu, Z., Krosnick, R., and Solar-Lezama, A. (2014). Modular synthesis of sketches using models. In *Verification, Model Checking, and Abstract Interpretation: 15th International Conference, VMCAI 2014, San Diego, CA, USA, January 19-21, 2014, Proceedings 15*, pages 395–414. Springer.

Slape, J. K. and Wallis, P. J. (1983). Conversion of fortran to ada using an intermediate tree representation. *The Computer Journal*, 26(4):344–353.

Smith, G. H., Kushigian, B., Canumalla, V., Cheung, A., Lyubomirsky, S., Porncharoenwase, S., Just, R., Bernstein, G. L., and Tatlock, Z. (2024). Fpga technology mapping using sketch-guided program synthesis. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 416–432.

Smith, S. and Karypis, G. (2016). SPLATT: The Surprisingly ParalleL spArse Tensor Toolkit. `http://cs.umn.edu/~splatt/`.

So, S. and Oh, H. (2017). Synthesizing imperative programs from examples guided by static analysis. In *International Static Analysis Symposium*, pages 364–381. Springer.

Solar-Lezama, A., Tancau, L., Bodík, R., Seshia, S. A., and Saraswat, V. A. (2006). Combinatorial sketching for finite programs. In *ASPLOS*, pages 404–415. ACM.

Song, L., Chi, Y., Sohrabizadeh, A., kyu Choi, Y., Lau, J., and Cong, J. (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*, pages 65–77.

Stahlberg, F. (2020). Neural machine translation: A review. *Journal of Artificial Intelligence Research*, 69:343–418.

Stratton, J. A., Rodrigues, C., Sung, I.-J., Obeid, N., Chang, L.-W., Anssari, N., Liu, G. D., and Hwu, W.-m. W. (2012). Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing*, 127(7.2).

Stroustrup, B. (1996). A history of c++ 1979–1991. In *History of programming languages—II*, pages 699–769.

Sutskever, I., Vinyals, O., and Le, Q. V. (2014a). Sequence to sequence learning with neural networks. *Advances in neural information processing systems*, 27.

Sutskever, I., Vinyals, O., and Le, Q. V. (2014b). Sequence to sequence learning with neural networks. In *NIPS*, pages 3104–3112.

Szafraniec, M., Roziere, B., Leather, H., Charton, F., Labatut, P., and Synnaeve, G. (2023). Code translation with compiler representations. *ICLR*.

Tillet, P., Kung, H.-T., and Cox, D. (2019). Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pages 10–19.

Torlak, E. and Bodik, R. (2014). A lightweight symbolic virtual machine for solver-aided host languages. *ACM SIGPLAN Notices*, 49(6):530–541.

Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., et al. (2023a). Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*.

Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Babaei, Y., Bashlykov, N., Batra, S., Bhargava, P., Bhosale, S., et al. (2023b). Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*.

Udupa, A., Raghavan, A., Deshmukh, J. V., Mador-Haim, S., Martin, M. M. K., and Alur, R. (2013). TRANSIT: specifying protocols with concolic snippets. In *PLDI*, pages 287–296. ACM.

VanHattum, A., Nigam, R., Lee, V. T., Bornholt, J., and Sampson, A. (2021). Vectorization for digital signal processors via equality saturation. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 874–886.

Vapnik, V. N. (1997). The support vector method. In *International conference on artificial neural networks*, pages 261–271. Springer.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. (2017). Attention is all you need. In *NIPS*, pages 5998–6008.

Virtanen, P., Gommers, R., Oliphant, T. E., Haberland, M., Reddy, T., Cournapeau, D., Burovski, E., Peterson, P., Weckesser, W., Bright, J., van der Walt, S. J., Brett, M., Wilson, J., Millman, K. J., Mayorov, N., Nelson, A. R. J., Jones, E., Kern, R., Larson, E., Carey, C. J., Polat, İ., Feng, Y., Moore, E. W., VanderPlas, J., Laxalde, D., Perktold, J., Cimrman, R., Henriksen, I., Quintero, E. A., Harris, C. R., Archibald, A. M., Ribeiro, A. H., Pedregosa, F., van Mulbregt, P., and SciPy 1.0 Contributors (2020). SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272.

Wang, B., Kolluri, A., Nikolić, I., Baluta, T., and Saxena, P. (2023). User-customizable transpilation of scripting languages. *Proceedings of the ACM on Programming Languages*, 7(OOPSLA1):201–229.

Wang, B., Li, T., Li, R., Mathur, U., and Saxena, P. (2025a). Program skeletons for automated program translation. *Proceedings of the ACM on Programming Languages*, 9(PLDI):920–944.

Wang, C., Yu, T., Xie, C., Wang, J., Chen, D., Zhang, W., Shi, Y., Gu, X., and Shen, B. (2025b). Evoc2rust: A skeleton-guided framework for project-level c-to-rust translation. *arXiv preprint arXiv:2508.04295*.

Waters, R. C. (1988). Program translation via abstraction and reimplementation. *IEEE Transactions on Software Engineering*, 14(8):1207–1228.

Webber, A. B. (2010). *Modern Programming Languages: A Practical Introduction*. Franklin, Beedle & Associates, 2nd edition.

Wen, Y., Guo, Q., Fu, Q., Li, X., Xu, J., Tang, Y., Zhao, Y., Hu, X., Du, Z., Li, L., et al. (2022). Babeltower: Learning to auto-parallelized program translation. In *International Conference on Machine Learning*, pages 23685–23700. PMLR.

Weng, J., Jian, A., Wang, J., Wang, L., Wang, Y., and Nowatzki, T. (2021). UNIT: Unifying tensorized instruction compilation. *CGO*.

White, T. (2012). *Hadoop: The definitive guide*. " O'Reilly Media, Inc.".

Won, J., Mendis, C., Emer, J. S., and Amarasinghe, S. (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*, pages 920–934.

Woodruff, J., Armengol-Estapé, J., Ainsworth, S., and O'Boyle, M. F. P. (2022). Bind the gap: compiling real software to hardware FFT accelerators. In *PLDI*, pages 687–702. ACM.

Wu, X. and Demsky, B. (2025). GenC2Rust: Towards Generating Generic Rust Code from C . In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 90–102, Los Alamitos, CA, USA. IEEE Computer Society.

Xinyun, C., Chang, L., Dawn, S., et al. (2018). Tree-to-tree neural networks for program translation. *NeurIPS*.

Yadav, R., Aiken, A., and Kjolstad, F. (2022). DISTAL: The distributed tensor algebra compiler. *PLDI*.

Yang, Y., Prestwood, S., and Barnes, C. (2016). Vizgen: accelerating visual computing prototypes in dynamic languages. *ACM Transactions on Graphics (TOG)*, 35(6):1–13.

Yarberry, W. (2021). Dplyr. In *CRAN recipes: DPLYR, stringr, lubridate, and regex in R*, pages 1–58. Springer.

Zhan, Q., Hu, X., Xia, X., and Li, S. (2024). Verified lifting of deep learning operators. *arXiv preprint arXiv:2412.20992*.

Zhang, G., Mariano, B., Shen, X., and Dillig, I. (2023). Automated translation of functional big data queries to sql. *Proceedings of the ACM on Programming Languages*, 7(OOPSLA1):580–608.

Zhang, G., Xu, Y., Shen, X., and Dillig, I. (2021). Udf to sql translation through compositional lazy inductive synthesis. *Proceedings of the ACM on Programming Languages*, 5(OOPSLA):1–26.

Zhang, Q., Wang, J., Xu, G. H., and Kim, M. (2022). Heterogen: transpiling c to heterogeneous hls code with automated test generation and program repair. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1017–1029.

Zhao, T., Rucker, A., and Olukotun, K. (2023). Sigma: Compiling einstein summations to locatlity-aware dataflow. *ASPLOS*.

Zivojnovic, V. (1994). Dspstone: A dsp-oriented benchmarking methodology. *Proc. Signal Processing Applications & Technology, Dallas, TX, 1994*, pages 715–720.

Zohar, A. and Wolf, L. (2018). Automatic program synthesis of long programs with a learned garbage collector. *Advances in neural information processing systems*, 31.