Can LLMs Perform Verified Lifting of Code?

Sahil Bhatia
Jie Qiu
Sanjit A. Seshia
Alvin Cheung

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2024-11
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2024/EECS-2024-11.html

March 14, 2024

Can LLMs Perform Verified Lifting of Code?

SAHIL BHATIA[†], JIE QIU[§], SANJIT A. SESHIA[†], and ALVIN CHEUNG[†], [†] UC Berkeley, [§] Duolingo, USA

Domain-specific languages (DSLs) have become integral to various software workflows, offering domain-specific optimizations and abstractions that improve code readability and maintainability. These languages have found applications across diverse domains such as image processing, deep learning, and distributed computing. However, the adoption of these languages often necessitates developers to rewrite existing code using the specific DSLs. Manual rewriting is error-prone and infeasible, leading to the development of numerous automated code translation tools. One notably successful approach for addressing this challenge is verified lifting, which relies on program synthesis (search-based) to find programs in the target language that are functionally equivalent to the source language program. While several tools based on verified lifting have been developed for various application domains, they are often specialized for specific tasks or require significant manual effort in terms of domain knowledge or heuristics to scale the search. In this paper, leveraging recent advances in large language models (LLMs) for code, we propose an LLM-based approach to building verified lifting tools. We use the LLM's capabilities to reason about programs by leveraging contextual information to translate a given program into its corresponding equivalent in the target language. This contextual information includes expressing the semantics of the target language using Python as the intermediate language. Additionally, we utilize the LLMs to generate proofs for functional equivalence. We develop lifting-based compilers for **three** DSLs targeting different application domains. Our approach not only outperforms previous symbolic-based tools but also requires significantly less effort to build.

1 INTRODUCTION

In recent years, domain-specific languages (DSLs) have increasingly become part of software workflows. DSLs offer optimizations and abstractions that enhance code readability and improve performance in specific domains. Examples of recent DSLs include Spark for distributed computing, NumPy for array processing, TACO for tensor processing, and Domino for network packet processing. With new DSLs emerging for diverse application domains and programming languages, developers often face the task of manually rewriting existing code to incorporate these languages into their existing workflows. This manual rewriting process, being repetitive, can be tedious and may introduce bugs into the code. We term this code translation problem as *lifting* since it usually involves translating code in a somewhat lower-level, general-purpose language to equivalent code in a DSL.

To address this challenge, significant effort has been dedicated to developing tools aimed at automating the task of lifting. Rule-based approaches rely on traditional pattern-matching techniques [14] to construct DSL compilers. However, describing these rules can be complex, leading to interest in search-based techniques for DSL compiler construction. These techniques seek to leverage the advances in *program synthesis* (e.g., see [6, 7, 18]) over the last two decades. Contemporary program synthesis approaches can be broadly classified into two categories: *symbolic* and *neural*. The use of program synthesis for lifting, termed *verified lifting*, involves searching for a program in the DSL and subsequently formally verifying its semantic equivalence to the source program. Verified lifting has been successfully applied in building compilers for DSLs like Spark, SQL, Halide, and TACO [8]. Traditionally, symbolic techniques such as enumerative, deductive, and constraint-based searches have been employed for implementation. More recently, neural networks [12] have been leveraged to develop compilers that translate sequential programs into their corresponding functional equivalents.

Despite their successes, both symbolic and neural approaches exhibit common drawbacks: (1) The synthesizer is customized for each DSL, making them challenging to adapt for new DSLs, and

(2) Significant effort is required to design the synthesizer, including domain-specific heuristics for symbolic approaches and the generation of paired data for ML-based approaches, to enable generalization and scalability for the target DSL. To address these challenges, recently the Metalift [3] system was proposed, enabling developers to construct verified lifting-based compilers for their own DSLs. Metalift operates as a semi-automated tool, abstracting the search and verification phases from developers. However, developers still need to describe the space of potential solutions for the search engine to ensure tractability. Additionally, developers must devise heuristics to scale the search process, with many of these heuristics relying on domain-specific knowledge for successful application.

Large language models (LLMs) have made significant progress in various programming-related tasks, including code generation, repair, testing, summarization, and even formal methods tasks like aiding solver proofs for correctness and formalizing specifications from natural language. The success of LLMs in these tasks can be attributed to several factors:

(1) **Extensive Training Data**: LLMs are trained on vast and diverse datasets containing code-related information such as documentation, code repositories, and forums. This comprehensive training data seems to enable LLMs to learn the syntax, semantics, and patterns of widely-used programming languages (PLs).
(2) **Generalization Capability**: LLMs excel at reasoning about new tasks without the need for additional training, leveraging the context provided in the prompt to adapt to various scenarios. For instance, [4] demonstrated LLMs are few-shot reasoners.
(3) **Inherent Domain Knowledge**: LLMs seems to possess vast domain knowledge acquired from diverse textual sources spanning multiple domains. Unlike humans, who may struggle with retaining and retrieving information across diverse domains, LLMs excel at rapid information retrieval and synthesis.
(4) **Multimodal Integration**: The flexible interface of LLMs allows them to seamlessly integrate multi-modal information, a capability that symbolic solvers often struggle with due to their reliance on predefined, formalizable input formats.

Related Work: Despite their significant promise in various tasks, generating reliable code with formal guarantees on correctness remains a challenging task for LLMs. The general idea of using learning-based synthesis to generate code as well as proof artifacts for formal verification is not new; see, for example, [16]. Most work on LLMs for code generation does not explicitly integrate LLMs with verification oracles, and prior approaches have typically focused on either generating code or proofs independently. For instance, [5, 13] illustrate how to leverage LLMs to generate loop invariants and several other works have focused on using LLMs to generate code from diverse forms of specifications [9, 10, 15]. To leverage LLMs for building verified lifting compilers, two key constraints need to be addressed: the ability to generalize to new DSLs (i.e., generate code for languages unseen in the training data) and providing guarantees on the correctness of the generated code (i.e., generating a proof of validity).

In this work, we propose an approach to address these challenges and leverage LLMs to build Verified Lifting (VL)-based compilers. Our approach is inspired by the core technique of verified lifting, which involves translating the source program to a higher-level intermediate representation (IR). This IR describes the semantics of the operators in the DSLs, and once the synthesized code is verified, it is translated to the concrete syntax of the DSL using rewrite rules. Instead of prompting the models directly to generate code in the DSL, which may be new and unfamiliar, we leverage the reasoning capabilities of LLMs to infer code from context. We instruct the model via a prompt to generate code using the operators of the DSL, with Python serving as the IR to encode the semantics of these operators. Python's significant representation in the training datasets of LLMs makes it a suitable choice for this purpose. In addition to generating the DSL program, we also

```java
public class ConditionalSum {
  public static int sumList(List<Integer> data) {
    int sum = 0;
    for(int i=0; i<data.size(); i++) {
      int var = data.get(i);
      if(var < 100){
        sum += var;}}
    return sum;}}
```

(a) Source Code (S)

```python
def map(data,f):
  if len(data) == 0: return []
  else:
     return [f(data[0])] + map(data[1:],f)
def reduce(data,f):
  if len(data) == 0: return 0
  else:
    return f(data[0],reduce(data[1:],f))

def ite(a, b, cond):
    if cond: return a
    else: return b
```

(b) Target Language (T_{lang})

prompt the model to generate a proof of correctness for the program. We then translate both the generated program and the proof to the syntax of an automated theorem prover to verify if the program is functionally equivalent to the given source program for all program states.

In summary, this paper makes the following contributions

(1) We propose an approach to leverage LLMs for building VL-based compilers, simplifying the compiler-building process significantly compared to prior symbolic approaches. Our method drastically reduces the human effort required in traditional approaches for building such code translators.
(2) We demonstrate how our approach enables LLMs to generalize and generate code for new DSLs not present in their training dataset. Importantly, this is achieved with minimal prompting and without additional fine-tuning of the models.
(3) We show the effectiveness of our approach by constructing compilers for three DSLs spanning various application domains. In terms of accuracy, our LLM-based compilers achieve comparable performance to existing tools and, in some domains, outperforms the prior approaches.

2 BACKGROUND

In this section, we provide an overview and present an end-to-end example of verified lifting. Traditional compilers have relied on pattern-matching rules for translating programs from one language to another. These rules are typically manually defined by developers and are both tedious to write and prone to errors. To address these challenges, verified lifting (VL) uses a search-based approach for translation. Given a program (S) in the source language (S_{lang}), VL uses a search procedure to find a program (T) in the target language (T_{lang}) that is functionally equivalent to the given source program. VL has proven effective in building compilers for various application domains, such as translating Java to Spark (distributed computing), C++ to Halide (image processing), and Java to SQL (database), among others. The key idea behind VL involves using an intermediate representation (IR) of the operators in the target language. This representation captures only the functional description of the operators and ignoring the low-level implementation details. VL comprises three key phases: (1) search, (2) verification, and (3) code generation. In the search phase, the source program is **lifted** to transform it into a sequence of operators in the IR. This sequence serves as the program summary (PS) which summarizes the source program using the operators in the IR. Subsequently, the program summary is **verified** using a theorem prover to check for semantic equivalence with source for all program inputs. If the verification succeeds, PS is then translated into the concrete syntax of the target language using simple pattern-matching rules. These rules are notably simpler to write since the PS is already expressed using the operators in the target language.

Next, we demonstrate an example of translating a sequential Java program to Spark (distributed computing DSL) using VL. Fig. 1a displays a sequential source program (S). The given Java program takes a list of integers as input and calculates the sum of all integers in the list that are less than 100. In Fig. 1b, we illustrate the semantics of the `map` and `reduce` operators from the target language as the IR. Note that these functions abstract the low-level implementation details of the operators while capturing only the high-level semantics of the operator. Now, the objective is to find a sequence of `map` and `reduce` such that it is functionally equivalent to (S). Traditional approaches to solving this search problem in VL (Casper, Metalift) involve framing it as SyGuS [2] problem. A SyGuS problem involves defining a search space that syntactically restricts the space of possible solutions for the synthesizer, making the search problem tractable. Formally, this objective can be stated as $\exists\ T\ \in\ T_{lang}.\forall\ \sigma.\ S(\sigma) == T(\sigma)$, where T is a program in the target language. For the given problem, the synthesis phase would return:

```
reduce(map(data, lambda i : ite(i > 100, 1, 0)), lambda a, b: a + b)
```

The expression initially maps each element in data to either 1 or 0 based on whether the element 'i' is greater than 100 or not. Next it reduces the list obtained from the previous step by summing up all the elements to return the count of elements less than 100. Since S includes a loop, proving equivalence with the generated program requires another predicate called the "loop invariant". The synthesis, in addition to generating the program summary, also generates the loop invariant. The final step involves translating the generated program summary to the concrete syntax of the DSL (Spark) using simple pattern-matching rules, resulting in the following expression:

```
map(lambda i: 1 if i > 100 else 0).reduce(lambda a, b: a + b)
```

Solving this SyGuS problem requires substantial domain knowledge and heuristics from developers, as naively encoding all possible solutions in the grammar renders the search intractable. For instance, one standard technique involves incrementally increasing the depth of expressions in the search space, as it grows exponentially with the number of operators in the DSL. Other approaches include type-based filtering of expressions, eliminating choices based on static analysis, and multi-phase synthesis for generating different parts of the solutions. In the next section, we provide details on how this synthesis problem can be simplified using our LLM-based approach.

3 LLM-BASED VERIFIED LIFTING APPROACH

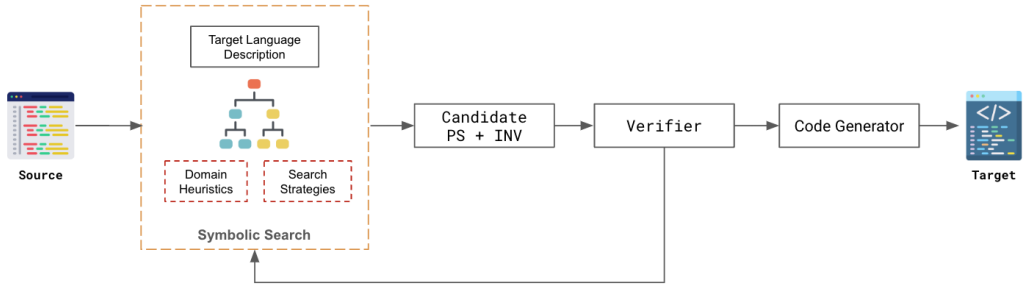
In this section, we describe our LLM-based approach for verified lifting. We begin by contrasting this approach with the traditional verified lifting approach as implemented in the MetaLift framework [3]. Then we give details of how we use LLMs to improve on certain aspects of the MetaLift approach.

3.1 Traditional vs. LLM-Based Approach for Verified Lifting

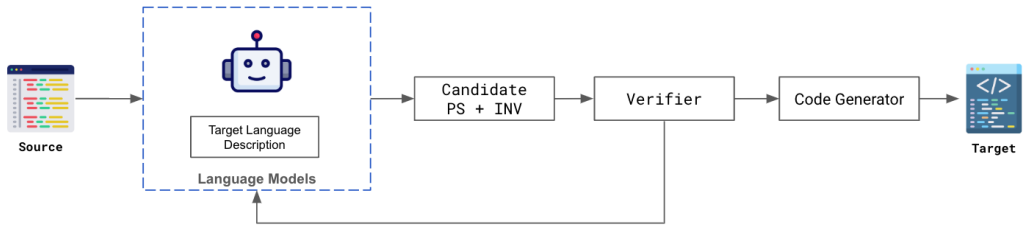
In Fig. 2, we illustrate the contrast between the traditional approach (orange) and our LLM-based approach (blue). Conventionally, compilers based on verified lifting have relied on symbolic search to solve the translation problem. When building a compiler using VL for a given source language (S_{lang}) and target language (T_{lang}), the search problem in VL is characterized by three components:

(1) **Specification** (ϕ): This defines the property that the target program (T) should satisfy. In the context of VL, this corresponds to T being semantically equivalent to the source program (S) for all program states.

(2) **Program Space** (G): This outlines the space of potential solutions, typically expressed as a context-free grammar. In VL, solutions space consist of various combinations of operators from T_{lang}. The design of G is crucial for the performance of search algorithms in VL. An overly restrictive grammar may limit expressiveness, failing to map some source programs. Conversely,



(a) Traditional symbolic verified lifting approach.



(b) LLM-based verified lifting approach.

Fig. 2. A comparison of the symbolic and LLM-based lifting approach.

an overly expressive grammar may become a bottleneck for the synthesizer, slowing down the synthesis process.

(3) **Search Algorithm** (A): This refers to the algorithm used to solve the synthesis problem. Traditional symbolic solvers utilize enumerative search, deductive search, and constraint-based approaches. Recently, neural approaches have also been proposed for this task.

Formally, we can describe the search problem in VL as:

$$\exists\ PS,\ \ Inv_1, Inv_2, \ldots\ \in G.\forall\sigma\ .\ \phi(PS, Inv_1, Inv_2 \ldots, \sigma) \tag{1}$$

This statement states that the goal is to find a program summary and invariants from the defined search space such that the given specification holds for all possible program states.

Traditional approaches rely on manually tuned search strategies, requiring developers to painstakingly design and fine-tune these strategies to effectively guide the search space. In contrast, our LLM-based approach relies on a simple prompt to perform the search, leveraging the inherent reasoning capabilities of Large Language Models (LLMs). Unlike traditional methods, which often struggle to adapt existing tools for new DSLs, our approach accommodates novel DSLs without the need for extensive modifications or domain-specific heuristics by leveraging Python as the IR.

Naively encoding all potential solutions in the grammar makes the search problem intractable. For instance, a straightforward search algorithm exploring every possible solution would need to search through at least n^m expressions for a depth m expression in a T_{lang} with n operators. To mitigate this exponential growth in the search space and make the search feasible, significant human input in the form of domain expertise and developer heuristics is necessary. For example, Casper [1], in constructing the Java to Spark compiler, adopts a standard technique of gradually increasing the depth of expressions in the search space. Other strategies include: (1) Type-based filtration of expressions, where operators incompatible with the source's output type are excluded from the grammar. (2) Eliminating choices through static analysis, as seen in C2TACO [11], which

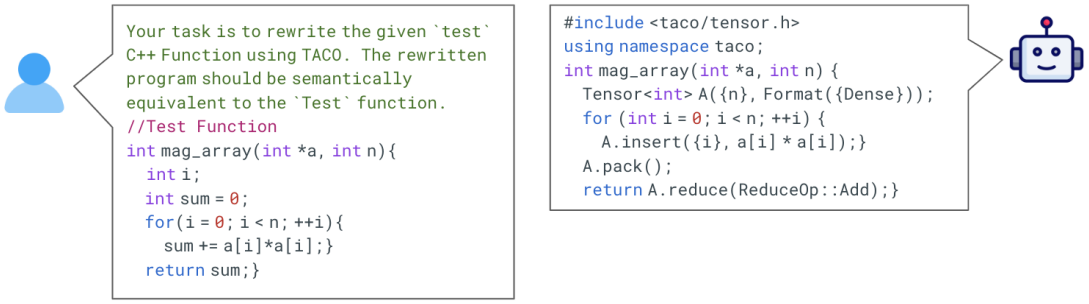


Fig. 3. End-to-End Lifting Example

incorporates only operators present in the source program. (3) Multi-phase synthesis, which involves generating different parts of the solutions sequentially. For instance, the high-level sketch may be synthesized initially, followed by the synthesis of its arguments in subsequent phases. It is important to recognize that these heuristics are often tailored to specific characteristics of the S_{lang} and T_{lang} , making them less universally applicable. Furthermore, these heuristics often need to be combined and adjusted based on the specific characteristics and requirements of the translation task. Successfully applying these techniques often demands a deep understanding of the domains involved and extensive experimentation.

While traditional approaches to building VL-based compilers rely on symbolic search and manual effort, we present an alternative approach for buildi leveraging LLMs to address these challenges. One approach to building a VL-based compiler using LLMs involves providing instructions via prompts to translate programs from S_{lang} to T_{lang}. However, using LLMs in an end-to-end manner has limitations due to several reasons:

(1) In code translation, the primary objective is to find a program in T_{lang} that is functionally equivalent to the program in S_{lang}. VL-based compilers require that the candidates generated during the search phase be tested for functional correctness using a verification oracle. This poses a challenge for LLMs because they lack an explicit link to any verifier, making model outputs uncertain in terms of correctness.

(2) While it's theoretically possible to develop a VL-based compiler for any new DSL and recent tools like MetaLift [3] assist developers in this process by abstracting the search and verification tasks. However, when prompted, LLMs struggle to generate code in low-resource languages that they have not encountered in training data. While LLMs can be fine-tuned with pairs from S_{lang} and T_{lang}, creating such pairs for an entirely new DSL presents a significant challenge.

In Fig. 3, we show an example of instructing the LLMs (GPT-3.5) to translate code in an end-to-end manner. We instruct the model to translate a C++ function to TACO (tensor processing DSL), and the model outputs a completely incorrect solution by hallucinating a TACO library. Currently, all the LLMs we have experimented with have a knowledge cut-off date, which is the point in time up to which the data, information, and events used to train a language model are considered. Even though GPT-3.5 has a knowledge cut-off date of 2021 and TACO was introduced much before 2021, the model still cannot generate the correct code. This problem will be more evident for completely new DSLs which the model might have never seen in the training dataset.

To address these challenges, we adopt the fundamental principle of VL, which involves generating code in an intermediate representation. In Fig. 2, we illustrate our LLM-based VL approach where we use LLMs to generate the program summaries and invariants in an IR. These are validated for

```
#Prompt Preamble
Your task is to rewrite the given `test` Java
Function. You need to use only the set of provided
functions and constants to achieve this. The
rewritten program should be semantically equivalent
to the `test` function.

#Semantics of Operators
def map(data,f):
  if len(data) == 0: return []
  else:
      return [f(data[0])] + map(data[1:],f)

def reduce(data,f):
  if len(data) == 0: return 0
  else:
    return f(data[0],reduce(data[1:],f))

#Test Function
public static int test(List<Integer> data) {
    int sum = 0;
    for(int i=0; i<data.size(); i++) {
      int var = data.get(i);
      if(var < 100){
        sum += var;}}
    return sum;}
```

Fig. 4. PS guessing prompt

correctness using a verification oracle. Following verification, we convert the program summaries into the concrete syntax of T_{lang} through simple pattern-matching rules. Our approach involves using LLMs within a few-shot learning framework, which we describe next.

3.2 Few-shot Learning Approach

LLMs have demonstrated few-shot reasoning capabilities [4]. Few-shot reasoning allows LLMs to generalize their understanding to new tasks by leveraging a small set of similar examples. This allows them to extend their reasoning capabilities to tasks without requiring explicit training or fine-tuning for those specific tasks. Formally, few-shot learning, often denoted as K-shot learning, involves presenting the model with K instances of a task description T_i and its corresponding solution Sol_i for $i = 1, 2, ..., K$ in the prompt. Here, T represents the task description, and Sol illustrates how to perform the task. Typically, K ranges from 0 to 10, indicating the number of examples provided to the model.

LLMs have been trained on vast amounts of code-related data, enabling them to understand and generate code across various programming languages and domains. LLMs can capture and utilize contextual information effectively. They can consider the entire context provided in a prompt or code snippet to generate syntactically and semantically meaningful code within the given context. In the context of VL, we leverage the few-shot reasoning capability by providing the models with the semantics of operators from the target language (T_{lang}). By exposing the LLMs to these semantics, we enable them to use their reasoning capabilities over code to generate the PS and invariants in T_{lang}. This helps the models to generalize their understanding to new DSLs without the need for explicit training or fine-tuning on those specific tasks

```
#Prompt Preamble
Your task is to prove that `assertion` is true in the `Test` function. The assertion can be
proved by finding a loop invariant using the defined functions. Write the loop invariant as a
python boolean formula. Example 1 shows a simple example of a loop invariant. Write the loop
invariant for Example 2.

#Example1:
#Semantics of Operators
def map(data,f):
  if len(data) == 0: return []
  else:
      return [f(data[0])] + map(data[1:],f)

#Test Function
public static int countList(List<Integer> data) {
  int count = 0;
  for(int i=0; i<data.size(); i++) {
    count++;}
    assert count == reduce(map(data, lambda x: 1), add);}

#Invariant
def invariant(data,count, i):
 return i >= 0 and i <= len(data) and count = reduce(map(data[:i], lambda x: 1),lambda
x,y:x+ y)

#Example2:
#Semantics of Operators
def map(data,f):
  if len(data) == 0: return []
  else:
      return [f(data[0])] + map(data[1:],f)

#Test Function
public static int test(List<Integer> data) {
  int sum = 0;
  for(int i=0; i<data.size(); i++) {
    int var = data.get(i);
    if(var < 100){
      sum += var;}}
  assert sum == reduce(map(data, lambda x: ite(x < 100, 1,0)), lambda x,y: x + y);}
```

Fig. 5. Invariant guessing prompt

As described in Sec. 2, the fundamental principle of VL involves generating candidates in an IR that abstracts away low-level implementation details of operators in T_{lang}. The objective, as shown in Eq. (1), is to find PS and Inv expressed using operators from T_{lang} such that ϕ holds. We split the generation of PS and Inv into a two-phase process by first guessing the PS and then inferring invariants corresponding to it. In Fig. 4, we show the prompt for generating the PS for our running example in Fig. 1a. We generate the PS in zero-shot setting using a prompt which consists

```
(define-fun-rec inv0 ((i Int) (data (List Int))) Bool
(and (and (>= i 0) (<= i (list_length arg))) (= i1 (reduce_lr (map_lm (list_take arg i2))))))
```

Fig. 6. SMT-LIB invariants

of task instruction (I), the semantics of DSL operators, and the specification (ϕ). While symbolic techniques often rely on approaches like test cases, bounded model checking, and VC (Hoare triple) for defining specifications, the natural language interface of LLMs offers flexibility in using various specifications and combining different forms. Given that LLMs are primarily trained on raw source code and may not have encountered other forms of specification during training, we directly use the source program (S) as the specification in our prompt. This simplifies the prompt and ensures our approach is independent of S_{lang} being used for translation. Additionally, we include the semantics of all operators in our T_{lang} as Python implementations in our prompt. Python is chosen as our IR due to (1) its widespread use across domains, (2) its concise and expressive nature, making the representation readable and straightforward and, (3) its significant representation in code datasets used for training LLMs [9]. Finally, we include a simple instruction to translate the given S using only the operators defined in the prompt. When prompted, the model generates the following solution:

```
reduce(map(data, lambda i : ite(i > 100, 1, 0)), lambda a, b: a + b)
```

Furthermore, to ensure that the generated candidates follow the DSL operators defined in the prompt, we use a parser to reject candidates which do not satisfy this constraint.

Given that S contains loops, establishing the functional equivalence of the generated PS for all program states with S necessitates the generation of loop invariants. In VL, loop invariants typically follow a templated structure:

$$Inv \triangleq f(i) \wedge e(T_{lang}) \tag{2}$$

where $f(i)$ denotes an expression over loop indexes and $e(T_{lang})$ represents an expression recursively constructed using operators from T_{lang}. This structured nature simplifies the invariant generation process compared to solving general loop invariant synthesis problems. To facilitate the generation of loop invariants, we use 1-shot learning to familiarize the model with the concept and structure of invariants in the VL context. In Fig. 5, we illustrate the prompt used to generate the invariants. The prompt for invariant generation closely resembles that used for generating program summaries, including S with an additional assertion stating the equality of the return variable with the previously generated PS. This instruction guides the model to produce an invariant corresponding to the generated PS. Additionally, the prompt includes the semantics of operators in our T_{lang} and instructs the model to generate a proof supporting the assertion encoded in S. Notably, the invariants are formulated as Boolean formulas in Python rather than SMT-LIB, as we found that LLMs encounter difficulties in generating SMT-LIB code due to its low-resource nature and limited representation in training datasets. When prompted, model generates the following solution:

```
def invariant(data, i):
  return i>=0 and i<=len(data) and sum = reduce(map(data[:i], lambda i : ite(i > 100, 1, 0)), lambda a, b: a + b)
```

The loop invariant states that the loop index i remains within the bounds of the data array ($0 \leq i \leq len(data)$). Additionally, the invariant expresses sum as the MapReduce expression over the first i elements of the data array which helps prove that the invariant holds in each iteration of the loop.

Both the program summaries (PS) and invariants are expressed in Python. We use simple pattern-matching rewrite rules to translate the expressions to the syntax compatible with the verification oracle used to check for functional equivalence. Once verified, the PS is similarly translated to the concrete syntax of T_{lang} using straightforward rewrite rules, leveraging the syntactic nature of Python. The translation process is simplified due to Python's highly structured syntax. In Fig. 6, we present the invariant expressed in the SMT-LIB format, where subexpressions are represented in a fully parenthesized, prefix style, contrasting with the in-order style notation of Python.

4 EXPERIMENTS

To evaluate the effectiveness of using LLMs for constructing VL-based compilers, we evaluate our approach across three distinct DSLs, each targeting diverse application domains:

(1) **Distributed Computing**: We convert sequential Java programs into MapReduce implementations within Apache Spark [20]. Spark, an open-source distributed computing framework, provides an interface for programming multiple clusters which for data parallelism which helps in large-scale data processing.
(2) **Network Packet Processing**: We map sequential network processing algorithms in C to the operators of programmable switch devices [17]. This translation enables the exploration of novel algorithms, such as congestion control and load balancing, on programmable switch devices.
(3) **Tensor Processing**: We convert sequential C++ programs into the operators of TACO [8] (tensor processing compiler), capable of generating highly optimized GPU code for performing tensor computations.

Model: In all experiments, we use GPT-4 via their APIs to generate candidates. The default temperature setting is applied for the model, and we generate ten completions for each experiment. For PS and invariant generation across all domains, we use the same prompt (excluding the DSL description) as depicted in Fig. 4 and Fig. 5, respectively.

4.1 Distributed Computing

MapReduce, a programming model for parallel processing of large datasets across distributed clusters, simplifies parallel computation by abstracting away distributed system complexities. It comprises two phases: (1) Map: Input data is partitioned into smaller chunks, each processed by a mapper function to generate key-value pairs. (2) Reduce: Intermediate key-value pairs are shuffled, sorted based on keys, and then processed by reducer functions to aggregate associated values.

LLM implementation. We compare the performance of our LLM-based implementation against MetaLift [3][1]. MetaLift uses a symbolic solver (Rosette [19]) to perform the search. We evaluate on the same 45 benchmarks as MetaLift. All the benchmarks have loops and require loop invariants to prove the functional equivalence of the source and the generated program. MetaLift solves 40 out of 45 with a timeout of 1 hour. Our LLM-based implementation is able to solve **44** i.e. generate the correct translation as well as the required invariants to prove the correctness. LLM approach solves 4 additional benchmarks on which MetaLift times out. In addition to solving more benchmarks, the amount of effort required to build the LLM compiler is significantly less than MetaLift as it does not require the developers to provide any search-space description for PS and invariants. LLMs perform really well on benchmarks which take >1min to solve as each call to the model takes 1 min (this is dependent on the size of the prompt).

[1]Casper [1] is not functional and Mold [14] is not open-sourced

Tool	Artificial	BLAS	DSP	DSPStone	makespeare	mathfu	simpl_array	UTDSP	darknet
C2TACO	100%	100%	100%	100%	100%	90%	100%	80%	92%
LLM-TACO	100%	100%	100%	60%	100%	**100%**	100%	**100**%	**100%**

Table 1. Accuracy on various benchmarks for tensor processing domain.

4.2 Network Packet Processing

Network packet processing hardware, such as routers and switches, lacks flexibility post-development, preventing experimentation with new data-plane algorithms. Recently, a verified lifting approach [17] was introduced to simplify this process. This compiler offers the developers with two constructs: (1) a packet transaction language (subset of C language) to express the semantics of these data-plane algorithms (2) a compiler that translates the packet processing algorithms to the instruction set of programmable switch devices. Atoms are introduced as an instruction set of the hardware to represent the atomic operations supported by the hardware. Compiler translates the packet transaction algorithm to a sequence of atoms resulting in a different programmable switch configuration.

LLM implementation. We implement the Domino compiler using LLMs by defining the semantics of the atoms in the prompt. We compare the performance of our implementation against MetaLift's implementation of the same. Note that all benchmarks in Domino are imperative C programs without any loop constructs and no loop invariants are required for these benchmarks. However, the generated PS are verified using a SMT solver. MetaLift is able to solve all the 10 benchmarks. Our LLM-based implementation is also able to map all the **10**. Similar to the Spark case study, we do not require developers to specify the search-space for PS. Our LLM-based implementation shows similar performance to the existing compiler but can be built using much less effort.

4.3 Tensor Processing

Tensors form the key construct in machine learning and tensor compilers play an important role in optimizing these operations. TACO [8] is one such compiler whic'h can automatically generate highly optimized code tailored to CPUs and GPUs. TACO's language represents the operations in a concise Einsum like notation. Recently, C2TACO [11] was proposed to automate the translation of C++ code to taco's representation to leverage the optimizations provided by the TACO compiler for legacy code.

LLM implementation. We implement the C2TACO compiler using LLMs by including the description of the TACO's notation in the prompt. We compare the performance of our implementation against the C2TACO. C2TACO evaluates kernels from several domains such as deep learning, linear algebra, array processing, signal processing and a few artificially generated ones. In Tab. 1, we show the accuracy of C2TACO and LLM-based implementation for all the domains. C2TACO achieves a mean accuracy of 95% while our LLM-based implementation also achieves the same accuracy across a total of 69 benchmarks. C2TACO uses an enumerative solver to perform the search and uses several heuristics including static code analysis, template enumeration to scale the search. On the other hand, our LLM-based approach just relies on a simple prompt and achieves the same performance. Also, our approach solves **2** additional benchmarks on which C2TACO times out after 90 minutes.

Discussion. Our experiments demonstrate the potential of using LLMs as reliable code translation tools. While verified lifting has been employed for years to develop compilers for various domains using symbolic search strategies, scaling symbolic search often requires the development of numerous search strategies with domain-specific heuristics. Our observations suggest that

LLMs can significantly reduce the effort required for this task. Due to their exposure to diverse training data, LLMs seems to possess numerous domain heuristics and can effectively reason about programs statically. This becomes particularly beneficial in benchmarks where the target program's depth is large, and symbolic engines must enumerate all candidates up to that depth. Moreover, our experiments reveal that LLMs perform well when presented with code written in high-level languages. As a result, we opted to utilize Python as the IR for expressing the semantics of the target language. Python's rich expressiveness and familiarity in the LLMs' training data make it a suitable choice. In addition, we found it advantageous to obtain proofs in Python and subsequently develop rule-based parsers to convert them into the concrete syntax of the solvers. This approach significantly reduces the complexity of prompt engineering required to instruct the model to generate solutions in low-level languages.

REFERENCES

[1] Maaz Bin Safeer Ahmad and Alvin Cheung. 2018. Automatically Leveraging MapReduce Frameworks for Data-Intensive Applications. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein (Eds.). ACM, 1205–1220.

[2] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-guided synthesis. In *2013 Formal Methods in Computer-Aided Design*. 1–8. https://doi.org/10.1109/FMCAD.2013.6679385

[3] Sahil Bhatia, Sumer Kohli, Sanjit A. Seshia, and Alvin Cheung. 2023. Building Code Transpilers for Domain-Specific Languages Using Program Synthesis. In *37th European Conference on Object-Oriented Programming (ECOOP 2023) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 263)*, Karim Ali and Guido Salvaneschi (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 38:1–38:30. https://doi.org/10.4230/LIPIcs.ECOOP.2023.38

[4] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners. In *Proceedings of the 34th International Conference on Neural Information Processing Systems* (Vancouver, BC, Canada) *(NIPS'20)*. Curran Associates Inc., Red Hook, NY, USA, Article 159, 25 pages.

[5] Saikat Chakraborty, Shuvendu K Lahiri, Sarah Fakhoury, Madanlal Musuvathi, Akash Lal, Aseem Rastogi, Aditya Senthilnathan, Rahul Sharma, and Nikhil Swamy. 2023. Ranking llm-generated loop invariants for program verification. *arXiv preprint arXiv:2310.09342* (2023).

[6] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. 2017. Program Synthesis. *Found. Trends Program. Lang.* 4, 1-2 (2017), 1–119.

[7] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. 2010. Oracle-Guided Component-Based Program Synthesis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE)*. 215–224.

[8] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The Tensor Algebra Compiler. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 77 (Oct. 2017), 29 pages. https://doi.org/10.1145/3133901

[9] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2023. StarCoder: may the source be with you! arXiv:2305.06161 [cs.CL]

[10] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme

Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022. Competition-level code generation with AlphaCode. *Science* 378, 6624 (Dec. 2022), 1092–1097. https://doi.org/10.1126/science.abq1158

[11] José Wesley de Souza Magalhães, Jackson Woodruff, Elizabeth Polgreen, and Michael F. P. O'Boyle. 2023. C2TACO: Lifting Tensor Code to TACO. In *Proceedings of the 22nd ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences* (Cascais, Portugal) *(GPCE 2023)*. Association for Computing Machinery, New York, NY, USA, 42–56. https://doi.org/10.1145/3624007.3624053

[12] Benjamin Mariano, Yanju Chen, Yu Feng, Greg Durrett, and Işil Dillig. 2022. Automated Transpilation of Imperative to Functional Code Using Neural-Guided Program Synthesis. *Proc. ACM Program. Lang.* 6, OOPSLA1, Article 71 (April 2022), 27 pages. https://doi.org/10.1145/3527315

[13] Kexin Pei, David Bieber, Kensen Shi, Charles Sutton, and Pengcheng Yin. 2023. Can Large Language Models Reason about Program Invariants?. In *Proceedings of the 40th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 202)*, Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett (Eds.). PMLR, 27496–27520. https://proceedings.mlr.press/v202/pei23a.html

[14] Cosmin Radoi, Stephen J. Fink, Rodric Rabbah, and Manu Sridharan. 2014. Translating Imperative Code to MapReduce. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications* (Portland, Oregon, USA) *(OOPSLA '14)*. ACM, New York, NY, USA, 909–927. https://doi.org/10.1145/2660193.2660228

[15] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2024. Code Llama: Open Foundation Models for Code. arXiv:2308.12950 [cs.CL]

[16] Sanjit A. Seshia. 2015. Combining Induction, Deduction, and Structure for Verification and Synthesis. *Proc. IEEE* 103, 11 (2015), 2036–2051.

[17] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. 2016. Packet Transactions: High-Level Programming for Line-Rate Switches. In *Proceedings of the ACM SIGCOMM 2016 Conference, Florianopolis, Brazil, August 22-26, 2016*. 15–28.

[18] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. 2006. Combinatorial sketching for finite programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM Press, 404–415.

[19] Emina Torlak and Rastislav Bodik. 2013. Growing Solver-aided Languages with Rosette. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software* (Indianapolis, Indiana, USA) *(Onward! 2013)*. ACM, New York, NY, USA, 135–152. https://doi.org/10.1145/2509578.2509586

[20] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI'12)*.