



# VEGA: Automatically Generating Compiler Backends using a Pre-trained Transformer Model

Ming Zhong
zhongming21s@ict.ac.cn
SKLP, ICT, CAS; UCAS
Beijing, China

Fang Lv*
flv@ict.ac.cn
SKLP, ICT, CAS
Beijing, China

Lulin Wang
wanglulin@ict.ac.cn
SKLP, ICT, CAS
Beijing, China

Lei Qiu
qiulei21b@ict.ac.cn
SKLP, ICT, CAS; UCAS
Beijing, China

Yingying Wang
wangyingying@bosc.ac.cn
SKLP, ICT, CAS
Beijing, China

Ying Liu
liuying2007@ict.ac.cn
SKLP, ICT, CAS
Beijing, China

Huimin Cui*
cuihm@ict.ac.cn
SKLP, ICT, CAS; UCAS
Beijing, China

Xiaobing Feng
fxb@ict.ac.cn
SKLP, ICT, CAS; UCAS
Beijing, China

Jingling Xue
j.xue@unsw.edu.au
UNSW
Sydney, Austrilia

## Abstract

We introduce VEGA, an AI-driven system aimed at easing the development of compiler backends for new targets. Our approach involves categorizing functions from existing backends into function groups, each comprising various target-specific implementations of a standard compiler interface function, abstracted as a single function template. Therefore, generating a new backend involves customizing these function templates to specific target requirements. To capitalize on AI's capabilities in code generation, VEGA maps statements in a target-specific version of a function template into feature vectors, distinguishing between target-independent and target-specific properties. Leveraging a pre-trained model, VEGA can *efficiently auto-generate* a version of each function template tailored to a specific target, thereby enabling the construction of a complete compiler backend for a new target based solely on its target description files.

We evaluated VEGA on three distinct targets: a CPU processor (RISC-V), a customized processor with instruction extensions (RI5CY), and an IoT processor (xCORE). VEGA demonstrated high efficiency, generating compiler backends under an hour, which can substantially enhance developer productivity. Across the three targets, VEGA achieved accuracy rates of 71.5%, 73.2%, and 62.2% for all generated functions, significantly outperforming the traditional fork-flow method, which yielded less than 8% accuracy. Moreover, VEGA provides explicit confidence scores for generated functions and statements, allowing developers to easily identify areas requiring minimal manual intervention. This research has the potential to improve the effectiveness of traditional compiler backend development.

***CCS Concepts:*** • **Software and its engineering** → *Retargetable compilers*; *Source code generation.*

***Keywords:*** AI-Generated Compilers, Compiler Backends

**ACM Reference Format:**
Ming Zhong, Fang Lv, Lulin Wang, Lei Qiu, Yingying Wang, Ying Liu, Huimin Cui, Xiaobing Feng, and Jingling Xue. 2025. VEGA: Automatically Generating Compiler Backends using a Pre-trained Transformer Model. In *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization (CGO '25), March 01–05, 2025, Las Vegas, NV, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3696443.3708931

## 1 Introduction

Over the last few years, the machine learning community has achieved remarkable progress in AI-generated content, spanning various application domains such as AI painting [49, 50], AI chatbots [52], and AI programming [27, 39, 65]. A notable example is DeepMind's AlphaCode [39], which demonstrated the ability to produce computer programs of competitive quality. Moreover, recent research has even explored the application of large language models (LLMs) in hardware logic design [5, 9, 10, 70], inspiring our investigation into AI-generated compiler backends.

To enhance retargetability, the compiler community has achieved success in creating sophisticated and robust compiler infrastructures such as GCC [23] and LLVM [43]. These

*Corresponding Author.


*CGO '25, March 01–05, 2025, Las Vegas, NV, USA*

ACM ISBN 979-8-4007-1275-3/25/03
https://doi.org/10.1145/3696443.3708931

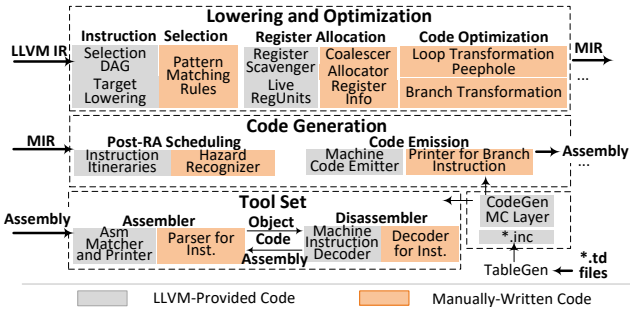



**Figure 1.** The compiler backend infrastructure in LLVM.

infrastructures divide a compiler into a frontend, a middle-end, and a backend, empowering compiler developers to focus on writing a compiler backend for a new target rather than building an end-to-end compiler from scratch.

A compiler backend translates a source language into object code (e.g., assembly or binary code). Fig. 1 depicts a typical LLVM compiler backend, divided into three stages with seven function modules, lowering an intermediate representation to assembly for a specific target processor.

At each stage, two types of code are present:

- The gray boxes contain LLVM-provided code requiring no manual effort. This code details target properties like registers and instructions, and defines reusable components via interfaces like classes and virtual functions. Target properties are captured by a target-independent code generator, as defined by LLVM, and the TableGen tool [43]. The code generator manages abstract target properties, while the TableGen tool records target-specific properties in `*.inc` files derived from developer-provided `*.td` files.
- The orange boxes indicate code provided by developers, who are tasked with manually creating target-specific code generation algorithms using LLVM code. This process often involves the inclusion of TableGen-produced `*.inc` files. For example, in the Code Emission module, developers are required to develop the `ARMELFObjectWriter` class, subclassing the `MCELFObjectTargetWriter` interface. A critical task is implementing the virtual function `getRelocType` for the ARM architecture, which necessitates the use of `ARMGenInstrInfo.inc`—derived from developer-provided `.td` files—and ARM-specific values from the `ARM::Fixups` enum in `ARMFixupKinds.h`.

Our analysis of 101 LLVM backends, selected from 379 GitHub repositories containing the keywords "`LLVM`" and "`Backend`", revealed that the amount of manually written code is significantly larger (37,780,305 lines) compared to the LLVM-provided code (16,727,635 lines). Consequently, creating concrete implementations for many LLVM-provided interface functions still demands substantial manual effort.

The primary challenge in developing AI-generated compiler backends is the overwhelming complexity involved. For example, the ARM backend in LLVM 12.0 comprises 3,628 functions and 83,704 lines of code. This vastness poses significant difficulties for existing AI models to fully grasp and accurately implement target-specific features.

In our pursuit of AI-driven compiler backend generation, we observe that all compiler backends share semantic equivalence, except for their target-specific implementations. For example, in LLVM, different implementations of an interface (e.g., the virtual function `getRelocType`) perform the same functionality across different targets but use target-specific values (e.g., from the `ARM::Fixups` enum on ARM) within a substantial amount of target-independent common code.

We introduce VEGA, a novel AI-driven system that streamlines the creation of compiler backends for new targets. Our approach organizes target-specific versions of standard compiler interface functions, such as `getRelocType`, into function groups. Each function group is then abstracted into a function template. Creating a new backend involves customizing these templates for each new target. VEGA leverages AI for code generation by transforming statements in a target-specific version of a function template into feature vectors, capturing both target-independent properties for common code and target-dependent properties for variant code that includes target-specific values. Utilizing a pre-trained transformer model, VEGA can *efficiently auto-generate* the target-specific version of each function template, thereby enabling the construction of an entire backend for the new target based solely on its target description files.

VEGA is aimed at easing compiler backend development by automating the creation of code that was previously written manually (highlighted in orange in Fig. 1). In addition, VEGA offers explicit and accurate confidence scores for generated functions and statements, identifying areas that need manual intervention. This marks a significant step towards the era of AI-generated compiler backends.

This paper presents the following main contributions:

- **Automatic Feature Selection** for backends by identifying target-dependent and target-independent properties, enabling their AI-driven automatic generation.
- **AI-Driven Code Generation** utilizing a transformer-based model to auto-generate backends for new targets, providing confidence scores for all functions and statements to enhance developer productivity.
- **Comprehensive Evaluation** using VEGA to develop backends for three targets in LLVM: RISC-V [61], RI5CY [22], and xCORE [76]. VEGA achieved accuracy of 71.5% for RISC-V, 73.2% for RI5CY, and 62.2% for xCORE, based on regression tests, measured by pass@1. This significantly surpasses the traditional fork-flow approach [60], which has an accuracy of less than 8.0% for these targets.

```
S1  unsigned Kind = Fixup.getTargetKind();
S2  MCSymbolRefExpr::VariantKind Modifier=Target.getAccessVariant();
S3  if (IsPCRel) {
S4    switch (Kind) {
S5      case ARM::fixup_arm_movt_hi16:
      ...
```

(a) ARM

```
S1  unsigned Kind = Fixup.getTargetKind();
S2
S3  if (IsPCRel) {
S4    switch (Kind) {
S5      case Mips::fixup_MIPS_HI16:
      ...
```

(b) MIPS

**Figure 2.** Two implementations of `getRelocType`.

## 2 Motivation

In this section, we present an end-to-end example that illustrates our machine-learning-driven approach. We detail the design and effectiveness of VEGA, highlighting its essential role in addressing the identified challenges. Additionally, we provide a clear rationale for employing machine learning over traditional statistical methods.

VEGA automatically generates all functions for a new compiler backend. We demonstrate this through a comprehensive example, where VEGA automatically generates a RISC-V-specific version of the `getRelocType` interface function in LLVM, leveraging existing ARM and MIPS implementations. Fig. 2 gives these implementations, highlighting five critical statements, $S_1$ to $S_5$, aligned using GumTree [18]. This function is essential for determining the appropriate relocation type for symbols or reference addresses [41].

VEGA operates in three stages. In the "Code-Feature Mapping" stage, VEGA identifies target-independent and target-dependent properties in the ARM and MIPS implementations, using these features to characterize each statement in a target-specific `getRelocType` implementation. During the "Model Creation" stage, VEGA fine-tunes a transformer model with feature vectors that represent statements from the ARM and MIPS implementations of `getRelocType`, thereby learning backend knowledge. Finally, in the "Target-Specific Code Generation" stage, VEGA synthesizes a RISC-V-specific implementation of `getRelocType` using the feature vectors derived from RISC-V's target description files.

Fig. 3 illustrates the Code-Feature Mapping and Model Creation stages using the ARM and MIPS implementations of `getRelocType`. Fig. 4 demonstrates the Target-Specific Code Generation stage for generating the RISC-V implementation for this interface function. To facilitate comprehension, Table 1 provides a list of symbols used in Fig. 2 – Fig. 4.

All LLVM-provided code is located under `LLVMDIRs` = {"llvm/CodeGen", "llvm/MC", "llvm/BinaryFormat", "llvm/-Target"}. The target description files in LLVM consist of TableGen files (.td), related .h files, and .def files, which are found in `TGTDIRs`={"lib/Target", "llvm/BinaryFormat/ELFRelocs"}.

**Table 1.** Description of symbols used in Fig. 2 – Fig. 4.

| Symbol | Description |
|---|---|
| $S_k$ | The $k$-th statement in a target-specific implementation of `getRelocType` |
| $T_k$ | The $k$-th statement in the function template capturing all `getRelocType`'s implementations |
| $V_k$ | The values recorded for $S_k$'s properties |
| $FV_k$ | The $k$-th feature vector $FV_k$ = <$T_k$, $V_k$> of $T_k$ |

### 2.1 Stage 1: Code-Feature Mapping

There are three steps, Templatization, Feature Selection, and Feature Representation, as shown in Fig. 3(a) – Fig. 3(e).

**2.1.1 Templatization.** Based on the ARM and MIPS implementations of `getRelocType` shown in Fig. 2, VEGA synthesizes a function template, comprising five *statement templates* $T_1$ – $T_5$, as depicted in Fig. 3(a). Each statement template includes variant code with a placeholder $SV_5$ for capturing target-specific values (if any), alongside the common code. VEGA templatizes each $T_k$ by matching corresponding statements $S_k$ from ARM and MIPS using GumTree [18].

It is noteworthy that $SV_5$ has 66 variants for ARM and 102 for MIPS, tailored to manage relocation fixups for specific instructions like `MOVT` on ARM and `LUI` on MIPS, targeting their upper 16 bits. Therefore, accurately identifying a target-specific value from these variants poses a significant challenge in automatic backend generation.

**2.1.2 Feature Selection.** Based on the `getRelocType` function template, VEGA identifies Boolean target-independent properties in the common code (Fig. 3(b)) and string target-dependent properties for specific values in the variant code (Fig. 3(c)) from its ARM and MIPS implementations. These properties serve as features characterizing the statements in each target-specific `getRelocType` implementation.

To locate a property, whether target-independent or target-dependent, we identify two key locations: (1) the *identified site* where the property is declared in `LLVMDIRs`, and (2) the *update site* in either `LLVMDIRs` or `TGTDIRs`. During this search, we exclude existing target-specific implementations highlighted in orange in Fig. 1, which is crucial for automatically generating a compiler backend for a new target using only its target description files.

Let us begin by analyzing the three target-independent properties illustrated in Fig. 3(b). Each property, represented as a Boolean, indicates the presence or absence of a type (such as a class or enum) or a global variable in a specific target implementation of `getRelocType`. To identify these properties, VEGA parses each statement in the `getRelocType` function template into individual tokens. In a target-specific implementation, each token associated with a target-independent property appears in the LLVM code under `LLVMDIRs` and may be potentially updated in the target description files under `TGTDIRs`. The target-independent properties identified in both the ARM and MIPS implementations are shared by the single function template for `getRelocType`.

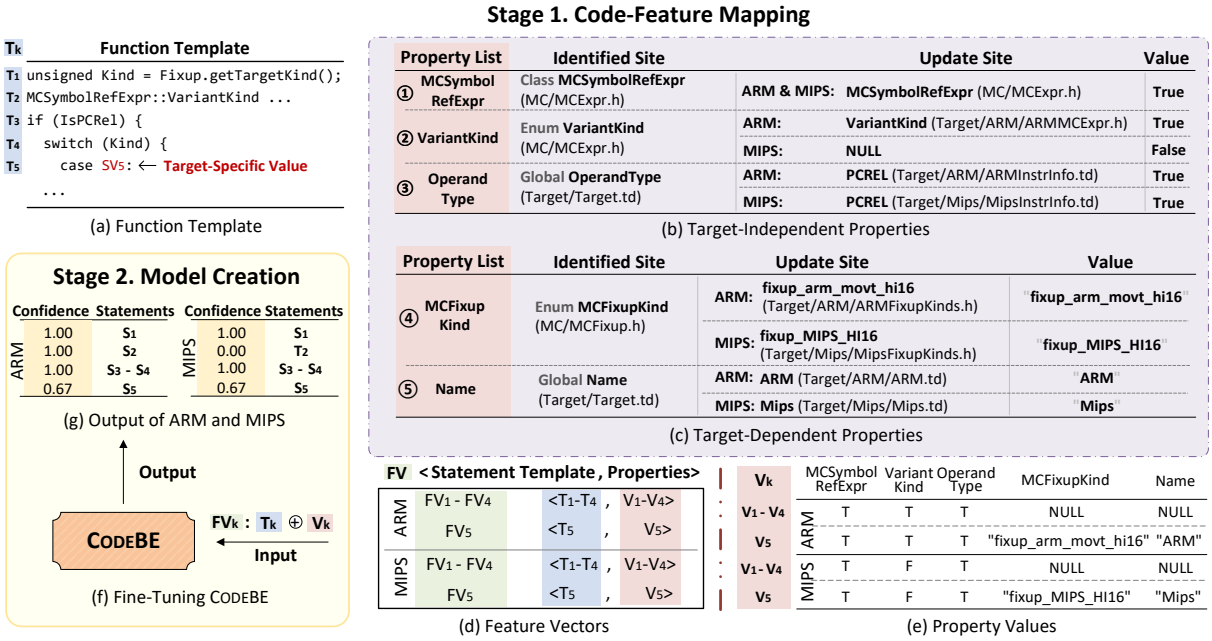



**Figure 3.** *VEGA*'s Code-Feature Mapping and Model Creation stages, illustrated using the implementations of the interface function `getRelocType` that are already available for two existing targets, ARM and MIPS, as shown in Fig. 2.

For example, $T_2$, which lacks placeholders, is broken down into several tokens, including `MCSymbolRefExpr` and `VariantKind`. VEGA identifies the class definition of MCSymbolRefExpr in `MCExpr.h` under `LLVMDIRs`, which acts as both its identified and update site, indicating it is defined for both targets. VariantKind varies between the two targets; for ARM, this enum is initially defined in `MCSymbolRefExpr::VariantKind` under `LLVMDIRs`, with ARM-specific members added in `ARMMCSymbolRefExpr::VariantKind` under `TGTDIRs`. For MIPS, this property is absent as $S_2$ is missing in Fig. 2(b), making its update site *null* and its property value false. Despite variations, VariantKind remains a target-independent property because its name is consistent across targets in the common code (free of placeholders). Finally, OperandType, a global variable defined in `Target.td` under `LLVMDIRs`, is linked to `IsPCRel` in $T_3$. This link is reinforced by a partial string match from `"IsPCRel"` to `"PCREL"` in the assignment `"OperandType = "OPERAND_PCREL""` found in both `ARMInstrInfo.td` and `MipsInstrInfo.td` under `TGTDIRs`, highlighting the importance of locating such target-independent properties for feature selection.

Let us move now to examining the two target-dependent properties in $T_5$ shown in Fig. 3(c), each a string, focusing on their discovery in ARM's $S_5$ (Fig. 2(a)). VEGA starts with `fixup_arm_movt_hi16` from `ARMFixupKinds.h` (update site), part of the `Fixups` enum under `TGTDIRs`. This enum correlates with the LLVM MCFixupKind in `MCFixup.h` (identified site) under `LLVMDIRs`, leading to the discovery of MCFixupKind for ARM with the value `fixup_arm_movt_hi16`. Additionally, the string `"ARM::fixup_arm_movt_hi16"` partially matches "ARM" in `"Name = "ARM""` from `ARM.td` (update site) under `TGTDIRs`, uncovering another target-dependent property, Name, with the value `"ARM"`, initially defined in `Target.td` (identified site) under `LLVMDIRs`. This process is similarly applied for MIPS, as also shown in Fig. 3(c).

Indeed, some statement templates, such as $T_4$, exemplified by `"switch (Kind) {"`, where `Kind` is a local variable defined in $T_1$, do not introduce any new properties.

### 2.1.3 Feature Representation.

VEGA utilizes the five properties identified in the `getRelocType` function template to characterize the statements $T_1$ – $T_5$ across all implementations. For target-specific versions on ARM or MIPS, target-independent properties maintain consistent values across statements, while target-dependent properties deliver specific values where applicable, or NULL otherwise.

For each statement $S_k$ in a target-specific `getRelocType` implementation, VEGA maps it to a feature vector $FV_k =< T_k, V_k >$ as shown in Fig. 3(d) and Fig. 3(e). Here, $T_k$ represents the tokenized statement template, and $V_k$ captures the property values of $S_k$. It is important to note that the values of $V_k$ for ARM and MIPS, which are detailed in Fig. 3(e), may differ between the two architectures.

Consider an example feature vector: on ARM, the feature vector $FV_5$ of $S_5$ is formed by concatenating $T_5$ and $V_5$ in that order, where $T_5$ = (`"case"`, `"SV5"`, `":"`) and $V_5$ = (T, T, T, `"fixup_arm_movt_hi16"`, `"ARM"`). For MIPS, $T_5$ remains the same as on ARM, while $V_5$ = (T, F, T, `"fixup_MIPS_HI16"`, `"Mips"`). Across both architectures, $V1$ through $V4$ are consistently (T, T, T, NULL, NULL) for ARM and (T, F, T, NULL, NULL) for MIPS.

### 2.2 Stage 2: Model Creation

In the second stage, VEGA fine-tunes the CodeBE component, an AI model based on UniXcoder [27], as shown in Fig. 3(f). It processes input sequences (feature vectors) from Fig. 3(d) and (e), and output sequences from Fig. 3(g) to automate compiler backend generation. CodeBE is vital for high-accuracy code generation, assimilating knowledge from feature vectors that encapsulate both target-independent and target-dependent properties. This enables it to generate new compiler backends, assigning a confidence score to each function and statement to indicate its likely correctness.

**2.2.1 Input Representation.** The input set is derived from the feature vectors for each target-specific version of `getRelocType`. For every ARM- or MIPS-specific statement $S_k$ (Fig. 2), with its feature vector $FV_k = T_k \oplus V_k$ already established (Fig. 3(d)), an input $I_k$ is created. This involves merging the tokens from its statement template $T_k$ (Fig. 3(a)) with its property values in $V_k$ (Fig. 3(e)).

**2.2.2 Output Representation.** In response to $I_k$ for a ARM- or MIPS-specific statement $S_k$, the output $O_k$, shown in Fig. 3(g), consists of $S_k$'s tokens preceded by a special confidence score, ranging from [0, 1.00], indicating the correctness certainty. Each statement, regardless of how many properties it has, is assigned a single confidence score. For $S_1$ – $S_4$ in both ARM and MIPS, except $S_2$ in MIPS, a score of 1.0 signifies high certainty. For MIPS, $S_2$ is *null* (indicating its absence), resulting in a score of 0.00, and $T_2$ is used instead in its place, suggesting lower confidence and a higher risk of incorrect code. For $S_5$, which includes a target-specific value, the confidence score, influenced by the potential value choices in $S_5$, is detailed in Sec. 3.2.2.

**2.2.3 Fine-Tuning.** CodeBE is fine-tuned using input and output sequences to understand details about target-specific implementations of each function template across existing compiler backends. Specifically, for our example, it focuses on the ARM and MIPS implementations of `getRelocType`.

### 2.3 Stage 3: Target-Specific Code Generation

Given a new target, VEGA utilizes all the function templates obtained during the learning stage and generates automatically a target-specific implementation for each function template. The confidence score of each generated function is given by the confidence score of its first line, which corresponds to its function definition statement.

Fig. 4(a) – Fig. 4(d) show how VEGA generates the RISC-V implementation of `getRelocType`. As depicted in Fig. 4(a), VEGA utilizes RISC-V target description files, such as `RISCVInstrInfo.td` and `RISCVFixupKinds.h` from `TGTDIRs`. In Fig. 4(c), it constructs five feature vectors, $FV_k =< T_k, V_k >$, with $T_k$ derived from the `getRelocType` function template (Fig. 3(a)) and $V_k$ sourced from the RISC-V files as detailed in Fig. 4(b). The identified and update sites for these values in RISC-V mirror those used for ARM and MIPS.

For the three target-independent properties shown in Fig. 4(b), identical to those in Fig. 3(b), VEGA locates them in either the LLVM-provided code under `LLVMDIRs` or in RISC-V's target description files under `TGTDIRs`, which follow the same naming conventions as those used for ARM and MIPS. In LLVM, files across different compiler backends adhere to a standardized naming convention, allowing VEGA to find corresponding files for new targets. For example, the update site of OperandType for ARM is lib/Target/{ARM}/{ARM}InstrInfo.td. Similarly, for RISC-V, VEGA locates `"OperandType = "OPERAND_PCREL""` in `lib/Target/{RISCV}/{RISCV}InstrInfo.td`, indicating that OperandType is also utilized for RISC-V as well.

For the two target-dependent properties in Fig. 4(b), identical to those in Fig. 3(c), VEGA checks the RISC-V update-site files for definitions. For MCFixupKind, VEGA reviews RISC-V's `RISCVFixupKinds.h` under `TGTDIRs`, analogous to `ARMFixupKinds.h` and `MipsFixupKinds.h` for ARM and MIPS, respectively, and identifies `"fixup_riscv_pcrel_hi20"` in the MCFixupKind enum. Therefore, this value is assigned to MCFixupKind in $V_5$. For Name, "Name = "RISCV"" is located in `RISCV.td` under `TGTDIRs`.

Once $V_1$ – $V_5$ are identified, VEGA constructs five feature vectors, $FV_1$ – $FV_5$, for RISC-V by combining $T_1$ – $T_5$ with $V_1$ – $V_5$, respectively. Then, using these feature vectors as input, VEGA uses CodeBE to generate the `getRelocType` function for RISC-V (Fig. 4(d)), with its statements annotated automatically with confidence scores. Notably, the confidence score for $S_2$ is 0.23, which is below the 0.5 accuracy threshold. After removing this statement, the generated implementation functionally matches the manual LLVM (12.0) version.

### 2.4 Discussion

To automate the construction of compiler backends, we have chosen machine learning over statistical methods. We find that AI models excel in inference tasks, particularly for determining values of target-dependent properties. A target-dependent property can have numerous possible values for a given target processor, indicating that the mapping from properties to values is not necessarily one-to-one. If a statement contains $N$ placeholders, $SV_1$, ..., $SV_N$, with each $SV_k$ having multiple possible values in its corresponding feature vector, selecting the correct combination of values for each $SV_k$ is a significant challenge that heavily depends on the statement's context. To the best of our knowledge, no well-established statistical methods exist for generating compiler backends automatically.

Furthermore, our VEGA approach assigns a confidence score to each statement in every function generated for a compiler backend. Relying solely on frequency statistics makes it challenging to accurately predict its likelihood

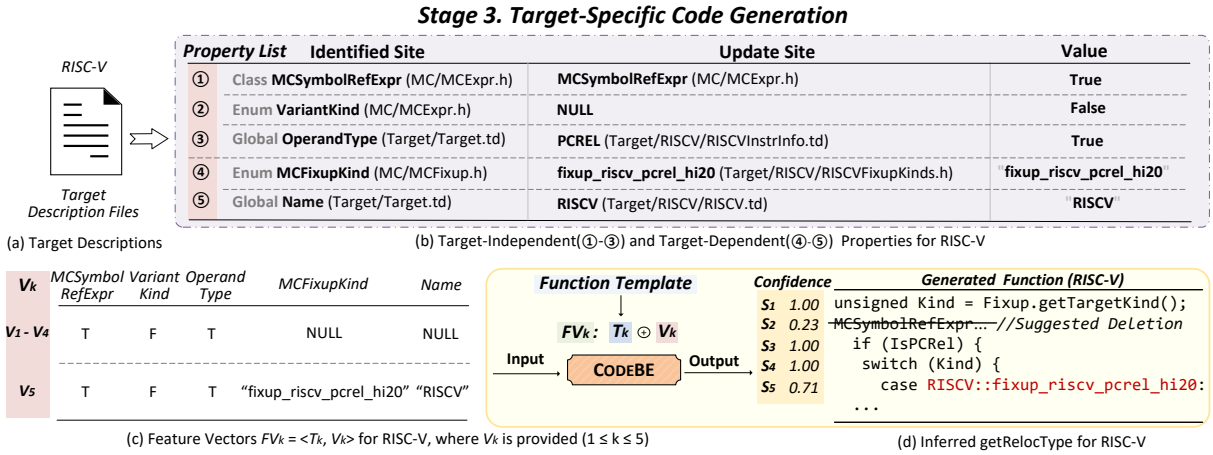



**Figure 4.** VEGA's Target-Specific Code Generation stage for emitting a RISC-V-specific `getRelocType` implementation.

of correctness. The presence of each statement in an auto-generated function depends on a combination of property values influenced by the statement's context. For example, $T_2$ in Fig. 3(a) appears for ARM due to a definition of VariantKind within ARM's `TGTDIRs` but is absent for MIPS. A machine learning approach more effectively captures the relationships between statement occurrences and their associated property values in the feature vector, thereby enhancing the precision of confidence score assessments.

## 3 VEGA: A Transformer-Based System

Expanding on our motivating example, we explore VEGA's architecture, shown in Fig. 5. We begin with preprocessing in existing compiler backends (Sec. 3.1) and discuss VEGA's three stages, spanning Sec. 3.2 to Sec. 3.4.

### 3.1 Pre-Processing

In developing VEGA, we used a dataset of 103 compiler backends from GitHub, identified with keywords "LLVM" and "Backend". After excluding two incomplete backends and removing non-functional elements like excess comments and unnecessary characters, the refined dataset includes a variety of target processors such as CPUs, GPUs, and DSPs.

The set of compiler backends, denoted as $\mathcal{B}$, undergoes the following preprocessing steps. Each function group $FG_M$ contains all target-specific implementations of a common LLVM-provided interface function $M$ (e.g., `getRelocType`) in $\mathcal{B}$. To improve syntactic resemblance, for each non-recursive function $f \in FG_M$, its callee functions are recursively inlined, maintaining calls to target-specific functions (e.g., inlining `GetRelocTypeInner` into `getRelocType`, as shown in Fig. 2(a)). We also align statements within each function group using GumTree [18], defining a *statement* as a line ending with any of { ";", ":", "{", "}" }. Additionally, we normalize equivalent selection statements like `if elif` into `switch`.

### 3.2 Stage 1: Code-Feature Mapping

We explain the automation of the three steps outlined in Fig. 5 for handling a function template. This process is detailed in Algorithm 1 and previously exemplified in Sec. 2.1.

**3.2.1 Templatization.** For the function group $FG_M$, linked to a common compiler-provided interface function $M$, VEGA creates a function template $FT_M$. This template blends common code shared across different target-specific implementations in $FG_M$ and variant code with placeholders for target-specific values (lines 1–2 of Algorithm 1). Each statement within a function of $FG_M$ is matched with corresponding statements in other functions of $FG_M$ using GumTree [18]. The Longest Common Subsequence analysis of the ASTs of these matching statements distinguishes the common code ($FT_M^{\text{com}}$) from the variant code ($FT_M^{\text{var}}$) in $FT_M$, with the latter containing placeholders for target-specific values. In the `getRelocType` function template (Fig. 3(a)), $FT_M^{\text{var}} = \{SV_5\}$, while $FT_M^{\text{com}}$ includes the rest of the common code.

**3.2.2 Feature Selection.** In the context of VEGA, identifying the features for a statement in a function template $FT_M$ entails searching through the LLVM-provided code in `LLVMDIRs` and the target description files for the compiler backends in $\mathcal{B}$ from `TGTDIRs`. This process aims to identify Boolean target-independent properties for $FT_M^{\text{com}}$ and string target-dependent properties for $FT_M^{\text{var}}$ as detailed in lines 3–40 of Algorithm 1. Thus, two types of directories are considered: `LLVMDIRs` (line 4) and `TGTDIRs` (line 7). `LLVMDIRs` contains all the fundamental definitions for compiler backends, such as types (i.e., classes and enums) and global variables. `TGTDIRs` contains a target-specific implementation for these definitions in terms of target description files (*.td, *.def, and *.h). In line 5, *PropList* contains all possible properties obtained from `LLVMDIRs`, represented by identifiers for class names, enum names, or global variables. Discovering properties for $FT_M$ involves string comparisons using Tokenizer [42] on token sequences of the files (lines 8 and 25).

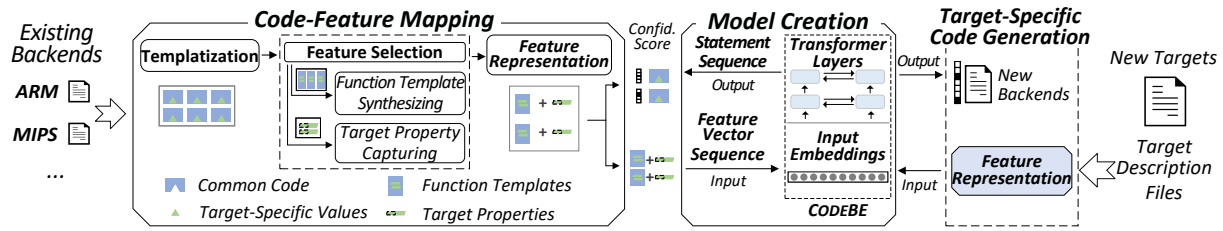



**Figure 5.** The architecture of VEGA.

**Algorithm 1:** Code-Feature Mapping

**Input:** $FG_M$: a function group of an interface function $M$
**Output:** $FV$: feature vectors for all functions in $FG_M$

1 // **1. Templatization**
2 $FT_M^{\text{com}} \leftarrow$ common code, $FT_M^{\text{var}} \leftarrow$ set of placeholders;
3 // **2. Feature Selection**
4 LLVMDIRs = {llvm/CodeGen, llvm/MC, llvm/BinaryFormat, llvm/Target};
5 $PropList$ = PropCandidateSet(LLVMDIRs);
6 **for** *TargetProc over the set of all targets* **do**
7 TGTDIRs={lib/Target/$TargetProc$, llvm/BinaryFormat/ELFRelocs};
8 **for** $tok \in Tokenizer(FT_M^{\text{com}})$ **do**
9 $found \leftarrow false$;
10 **if** *tok appears under* TGTDIRs **then**
11 **if** $tok \in PropList$ **then**
12 $found \leftarrow true$;
13 $FileLoc(tok) \leftarrow$ the file where $tok$ is found;
14 **if** $\neg found$ *and tok is a global s.t. tok is a substring of str or vice versa for an assignment "tok′ = str" under* TGTDIRs **then**
15 **if** $tok' \in PropList$ **then**
16 $found \leftarrow true, tok \leftarrow tok'$;
17 $FileLoc(tok) \leftarrow$ the file where $tok$ is found;
18 **if** $\neg found$ *and* $tok \in PropList$ **then**
19 $found \leftarrow true$;
20 $FileLoc(tok) \leftarrow$ the file where $tok$ is found;
21 **if** $found$ **then**
22 Record $tok$'s identified site in LLVMDIRs for $tok$;
23 Record $tok$'s update site in $FileLoc(tok)$;
24 Record $tok$'s value as true;
25 **for** $tok \in Tokenizer(FT_M^{\text{var}})$ **do**
26 $TgtValSet(tok) \leftarrow$ set of $TargetProc$-specific values of $tok$;
27 **for** $val \in TgtValSet(tok)$ **do**
28 $found \leftarrow false$;
29 **if** *tok appears as a member of an enum tok′ or an assignment "tok′ = tok" under* TGTDIRs **then**
30 **if** $tok' \in PropList$ **then**
31 $found \leftarrow true, tok \leftarrow tok'$;
32 $FileLoc(tok) \leftarrow$ the file where $tok$ is found;
33 **if** $\neg found$ *and tok is a global s.t. tok is a substring of str or vice versa for an assignment "tok′ = str" under* TGTDIRs **then**
34 **if** $tok' \in PropList$ **then**
35 $found \leftarrow true, tok \leftarrow tok'$;
36 $FileLoc(tok) \leftarrow$ the file where $tok$ is found;
37 **if** $found$ **then**
38 Record $tok$'s identified site in LLVMDIRs for $tok$;
39 Record $tok$'s update site in $FileLoc(tok)$;
40 Record $tok$'s value in $FileLoc(tok)$
41 // **3. Feature Representation**
42 **for** *function* $f \in FG_M$ **do**
43 **for** *statement* $S$ *in* $f$ **do**
44 $T \leftarrow$ the corresponding statement template in $FT_M$;
45 Map $S$ to its feature vector $FV < T, V >$, where $T$ is tokenized and $V$ is the set of its property values found (lines 8–40);
46 $FV \leftarrow$ set of feature vectors thus created;
47 **return** $FV$;

In line 6, we go over all available targets, one at a time.

To locate a property $p$, we identify two crucial locations: the *identified site*, where $p$ is declared in LLVMDIRs, and the *update site*, where $p$ is defined or modified. This process also determines the property's value, linking each property to three attributes: its identified site, its update site, and its value. Importantly, the *update site* for $p$ can be in LLVMDIRs (where it is both declared and defined) or TGTDIRs if $p$ is target-independent, or exclusively in TGTDIRs if $p$ is target-dependent. We previously discussed the identified and update sites, along with the values for the five properties in our motivating example, in Fig. 3(b), Fig. 3(c), and Fig. 4(b).

In lines 8–24, we seek a target-independent property for a token $tok$ in $FT_M^{\text{com}}$ (the common code in $FT_M$) for a specific target (lines 6–7). The search is conducted first in TGTDIRs and then in LLVMDIRs, as $tok$ defined in LLVMDIRs might be updated (with a global variable defined, an enum specialized, or a class potential overridden). The process involves three cases. If $tok$ is found in a file under TGTDIRs, it is identified as a target-independent property when $tok \in PropList$ (lines 10–13). In the second case, if $tok$ partially matches the RHS $str$ of an assignment "$tok'$ = str" in a file under TGTDIRs, $tok'$ is recognized as a target-independent property for $tok$ when $tok' \in PropList$ (lines 14–17). This partial matching process is effective for two reasons. First, tokens in LLVM backends, like $tok$, are typically named descriptively, such as OperandType and IsPCRel, which clearly indicates their function, rather than generic identifiers like i and tmp. Second, in the assignment $tok' = str$, $tok'$ represents a class name, enum name, or global variable (all part of $PropList$), while $str$ denotes a target-specific value. Thus, through partial matching, $tok'$ helps capture such target-independent properties that significantly influence code inference. For example, this applies to discovering the property OperandType from IsPCRel present in Fig. 3(b), as explained in Sec. 2.1.2. In the third case, if $tok$ appears in LLVMDIRs (when $tok \in PropList$), it is recognized as a target-independent property (lines 18–20). Finally, we record the three attributes for the discovered target-independent property (lines 21–24).

In lines 25–40, we identify a target-dependent property for a token $tok$ in $FT_M^{\text{var}}$ (the variant code of $FT_M$). This process mirrors that for target-independent properties but excludes the third case for $FT_M^{\text{com}}$ as target-dependent properties are consistently redefined with target-specific values

in TGTDIRs, regardless of prior definitions in LLVMDIRs. It should be pointed out that, in line 27, $TgtValSet(tok)$ requires just one target-specific value from an enum, as any member identifies the target-dependent property.

Every property must have a unique identified site in LLVMDIRs. However, if a property $p$ is target-specific, it might not exist for another target (e.g., VariantKind for ARM is absent in MIPS, as discussed in Sec. 2.1.2). In such cases, the update site and updated values for $p$ become undefined. For target-independent $p$, the update site and updated value default to NULL and false, respectively, as $p$ is a Boolean. For target-dependent $p$, both are set to NULL, as $p$ is a string.

After the completion of Algorithm 1, all the statements in a function group $FG_M$ have been linked to their target-independent and target-dependent properties.

**3.2.3 Feature Representation.** In lines 41–47, we map each statement from every target-specific function in $FG_M$ to a feature vector. This mapping utilizes the tokenized statement template from $FT_M$ and integrates both target-independent and target-dependent properties identified earlier, as illustrated in Fig. 3(b) and Fig. 3(c).

### 3.3 Stage 2: Model Creation

In VEGA, as shown in Fig. 5 and demonstrated in Fig. 3(f) and Fig. 3(g), the AI model component, CodeBE, is fine-tuned using UniXcoder [27], which excels at generating code fragments, including C++. This is ideal for our objective of generating LLVM compiler backends primarily in C++.

For each interface function $M$, $FG_M$ denotes the function group with all target-specific implementations, and $FT_M$ the corresponding function template. Each statement $S_k$ correlates with its statement template $T_k$ in $FT_M$. Let $f \in FG_M$ be a specific target implementation of $M$. CodeBE learns backend knowledge by deriving the set of input sequences $I_f$ and the set of output sequences $O_f$ for $f$. Specifically, $I_f$ includes feature vectors for each statement $S_k$ in $f$, represented by $I_k$ from $FV_k < T_k, V_k >$ (lines 42–45 of Algorithm 1). These feature vectors combine tokens from $T_k$ with property values $V_k$, the latter capturing both target-independent and target-dependent properties, distinguished by a [SEP] token. CodeBE processes these vectors into input embeddings using token and position embeddings, as detailed in [27].

Additionally, CodeBE generates an output sequence $O_k \in O_f$ for each input $I_k \in I_f$, including a confidence score from [0, 1.00] and the associated statement $S_k$. For non-existent statements like $S_2$ for MIPS (Fig. 2(b)), CodeBE assigns a 0.00 score and substitutes $T_2$ for $S_2$ (illustrated in Fig. 3(g)), indicating the lowest confidence. This approach not only enhances the accuracy of confidence scoring but also supports effective code generation while addressing potential inaccuracies. Subsequently, each output sequence $O_k$ is tokenized.

In CodeBE, both $I_k$ and $O_k$ are preceded by [CLS] and [E2D] tokens to activate the encoder-decoder mode [27].

The confidence score for a statement $S_k$ in $f \in FG_M$, where $f$ is implemented for target $P_f$, denoted by $CS(S_k)$, is estimated by examining the presence of potential target-specific values in $T_k$ for target $P_f$. Let $|T_k|$ represent the total number of tokens in $T_k$. Additionally, let $|T_k{}^{\text{com}}|$ denote the number of tokens in the common code, and $|T_k{}^{\text{var}}|$ indicate the number of placeholders in $T_k$. For each placeholder $SV$ in $T_k{}^{\text{var}}$, let $N(SV)$ represent the total number of possible target-specific values for target $P_f$. Consequently, $CS(S_k)$ can be estimated as follows:

$$CS(S_k) = \left( \frac{|T_k{}^{\text{com}}|}{|T_k|} + \sum_{SV \in T_k{}^{\text{var}}} \frac{1}{|T_k| \times N(SV)} \right) \times has(S_k) \quad (1)$$

As an empty sum equals zero, $CS(S_k)$ simplifies to $(|T_k{}^{\text{com}}|/|T_k|) \times has(S_k)$, reducing to $has(S_k)$ when $T_k{}^{\text{var}} = \emptyset$. Here, $has(S_k)$ is 1 if $S_k$, derived from its function template $FT_M$, exists in the target-specific implementation $f$ in target $P$, and 0 if absent. For instance, $T_2$ in Fig. 3(a) has no corresponding $S_2$ on MIPS as shown in Fig. 2(b), resulting in $has(S_2) = 0$. Hence, if $T_k{}^{\text{var}} = \emptyset$, $CS(S_k)$ simplifies to 1 if $S_k$ is present and 0 otherwise. The confidence score for a function's definition is similarly set to 1 if it exists in target $P_f$ and 0 otherwise, using the same logic.

Generally, a statement is considered potentially correct if it has a confidence score of 0.5 or higher, and deemed incorrect if the score is below 0.5. For example, $S_2$ in Fig. 4(c) is labeled incorrect as its confidence score is 0.23. Removing ensures that the generated getRelocType for RISC-V aligns with the manually crafted LLVM (12.0) version.

### 3.4 Stage 3: Target-Specific Code Generation

VEGA generates a compiler backend for a new platform using all available function templates, as demonstrated in Fig. 5 and Fig. 4(a)-(d). It creates target-specific functions for each template and assigns confidence scores to their statements. The confidence score of the first statement indicates the overall confidence in the entire function, simplifying the correction of erroneous code and boosting productivity.

For each function template $FT_M$ of an interface function $M$, containing $N$ statements $T_k$, VEGA generates $N$ feature vectors for a new target based on (1) the target's description files and (2) existing feature vectors from other implementations of $M$ trained earlier. Each feature vector, $FV_k < T_k, V_k >$, aligns with a statement template $T_k$ in $FT_M$ and is tokenized as during the training phase. To populate $V_k$ with property values for the new target, VEGA examines feature vectors from previous targets to pinpoint their data sources (i.e., identified and update sites) and scans TGTDIRs for matching files to update values, as illustrated earlier in Sec. 2.3.

Utilizing the feature vectors developed for each function template specific to a target, CodeBE will generate a version of the template customized to that target. As a result, this enables CodeBE to create a complete backend for a new target, relying exclusively on its target description files.

# 4 Evaluation

We demonstrate that VEGA, an AI-driven framework, can automatically generate new compiler backends, significantly streamlining the process compared to traditional methods. A new compiler backend comprises all functions automatically generated from all the synthesized function templates.

## 4.1 Methodology

We evaluate VEGA in comparison to the state of the art, focusing on two critical areas: its efficiency in accurately inferring and generating new backends, and its capability to automatically obtain high-performing backends, especially after rectifying any inaccuracies in functions.

**4.1.1 Target Processors.** VEGA is tailored for scenarios with rapidly evolving processor architectures. We evaluate VEGA on three types of processors (Fig. 6(a)): (1) RISC-V [61], an open-source general-purpose processor (GPP) with 9 types of instructions [24], (2) RI5CY [56], a customized processor with an ISA extension for ultra-low power (ULP) [56], making it ideal for testing VEGA's backend generation with custom ISA extensions, and (3) xCORE [76], representing IoT chips with domain-specific instructions, chosen to test VEGA's backend creation for non-RISC-V instruction sets. As illustrated in Fig. 6(b), these processors have unique ISAs for different domains, highlighting their adaptability to various customized architectures.

**4.1.2 Training.** Our CodeBE model employs UniXcoder (the unixcoder-base-nine version) [27], a 12-layer Transformer with 125M parameters, known for its proficiency in generating code across multiple programming languages, including C++ for LLVM backends. Although detailed evaluation results are omitted here, the UniXcoder-based VEGA significantly surpasses both RNN-based VEGA [32] and vanilla-BERT-based VEGA [15] in function generation accuracy, with improvements of 35.3% – 77.7% and 32.1% – 67.0% respectively (measured by pass@1 [36]). Recent studies also confirm UniXcoder's superiority over code-specific BERT variants like CodeBert [20] and GraphCodeBert [28] [27], highlighting the critical role of a high-quality pre-trained model in VEGA's success.

In CodeBE, hyperparameters are set with a learning rate of 6e-5, a batch size of 256, and a maximum sequence length of 512 for both inputs and outputs. Consequently, the length of each feature vector $FV_k < T_k, V_k >$ is established at 512. Specifically, $V_k$ is designated to represent 345 distinct properties, while $T_k$ is maximized at 167. If $T_k$ contains fewer than 167 tokens, UniXcoder automatically fills the remaining positions with a special [PAD] token.

In the experimental evaluations with RISC-V, RI5CY, and xCORE, no input or output sequence exceeded 512 tokens, allowing VEGA to avoid token limit issues and ensuring consistent performance across all three targets. Should future needs require handling $FV_k < T_k, V_k >$ longer than 512 tokens, we could switch from UniXcoder to larger models like Code-LLaMA [63], which supports up to 16,384 tokens.

The CodeBE model undergoes fine-tuning for 50 epochs using the Adam optimizer and cross-entropy loss, with end-to-end back-propagation targeting a sequence-to-sequence objective.

From the 101 backends in $\mathcal{B}$, we exclude RISC-V, RI5CY, and xCORE, retaining 98 backends. These are split into training and verification sets. We randomly allocate 75% of functions from each function group to the training set, formed automatically by the function names implemented across different targets. The training set includes 107,718 statements across 7,902 functions in 825 function groups. The remaining 25% form the verification set, which contains 52,615 statements in 3,338 functions from 825 groups. CodeBE's inference on this verification set achieves a 99.03% Exact Match score [27], demonstrating its effectiveness in adapting backends for new targets.

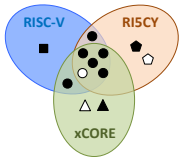
**4.1.3 Workloads.** In line with established procedures, we evaluate the accuracy of all functions generated for a new backend by conducting regression tests against its existing counterpart in LLVM. We select the LLVM regression tests with 16,005 cases for RISC-V, 16,040 for RI5CY, and 5,466 for xCORE [43] as the regression test suite. Additionally, we assess the robustness and performance of the three VEGA-generated backends with 28 C/C++ benchmarks in SPEC CPU2017 [13] for RISC-V (excluding FORTRAN benchmarks due to inadequate tool-chains), 69 test cases from PULP regression suite [57] for RI5CY, and 22 Embench benchmarks [66] for xCORE. To manage high memory requirements and limitations in target-specific simulators, we include smaller test cases from Embench.

**4.1.4 Evaluation Metrics.** VEGA leverages LLVM as the foundational framework, acting as the *base compilers* for developing new compiler backends for the targets evaluated in this paper. Specifically, we use LLVM 12.0 for RISC-V and RI5CY, while opting for LLVM 3.0 for xCORE due to a specific requirement related to thread stack allocation [47]. Additionally, it is worth noting that the LLVM 3.0 version lacks integration of the disassembler module for xCORE.

To assess VEGA's inference ability, we evaluate the accuracy of its generated functions and the confidence scores. We use the pass@1 methodology [36], where each newly generated function (without manual modification) substitutes its corresponding function inside the base compiler. We then evaluate the compiler's correctness through regression tests. Functions passing this evaluation are labeled as *accurate functions* and included in the successful outcomes pool.

| | Category | | RISC-V (GPP) | RI5CY (ULP) | xCORE (IoT) |
|---|---|---|---|---|---|
| ISA | Standard | | ○ I,M,F,C,D,A,B,FH,V | ○ I,M,F,C | ○ Data Access, Expression, Evaluation, etc* |
| | Customized | | — | ✿ Hardware loop, SIMD, etc | △ Thread Scheduler, Synchronization, etc |
| Function Module | Instruction Selection | SEL | ● Converts IRs into selection nodes | ● Conversion rules for customized ISA | ● Conversion rules for real-time instructions |
| | Register Allocation | REG | ● Transforms virtual regs into physical regs | — | ● Transforms conventions for specialized regs |
| | Code Optimization | OPT | ■ Improves machine-dependent code | ⬟ Hardware loop and SIMD optimization | ▲ Real-time optimization |
| | Instruction Scheduling | SCH | ● Reorders instructions | ● ISA-related info and scheduling | ● Reorders instructions |
| | Code Emission | EMI | ● Emits specific object code formats | ● New format of instructions | ● Emits new format of specialized real-time ISA |
| | Assembler | ASS | ● Parses assembly to instructions | ● New format of assembly code | ● Parses new format of specialized real-time ISA |
| | Disassembler | DIS | ● Converts bytes into assembly code | ● New instructions and register classes | — |

**(a)** ISAs and Function Modules




**(b)** Differences in ISAs

**Figure 6.** Three target processors, where '*' for xCORE symbolizes standard instructions uniquely named in xCORE.

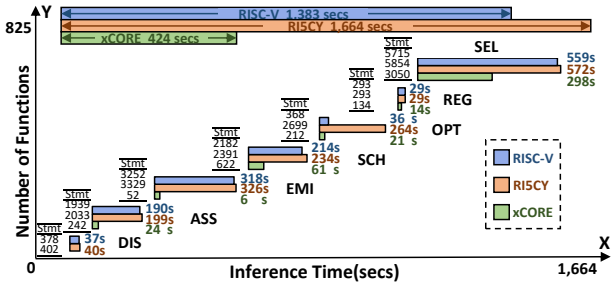



**Figure 7.** Inference times for the three targets in VEGA.

To evaluate VEGA's compilation capability, we focus on measuring its robustness. This involves replacing all inaccurate functions in the three VEGA-generated compiler backends with accurate ones from their respective base compilers. We then assess the correctness and performance of these amended compilers using our designated workloads.

**4.1.5 Computing Platform.** VEGA is deployed on a server with 64-core, 2.60GHz Intel Xeon Gold 6132 CPU, 512 GB of memory, and 8 NVIDIA Tesla V100 GPUs with 16GB memory each. All evaluations are conducted under this environment. Additionally, three simulators are utilized to run the resulting binaries: RISC-V QEMU Simulator [58] for RISC-V, PULP RTL simulation platform [56] for RI5CY, and XSIM [76], a hardware simulator for xCORE.

**4.1.6 Pre-trained UniXcoder.** To ensure that UniXcoder [27] was not pre-trained with compiler backends for RISC-V, RI5CY, and xCORE, we tested the original model (without fine-tuning) on our test data for these targets. The original UniXcoder failed to generate any correct functions, confirming that our test data was not included in its training dataset.

## 4.2 VEGA's Code Inference Ability

We demonstrate that VEGA lessens the backend development burden for compiler developers, enabling them to create new compiler backends more efficiently.

**Training and Inference Times.** The Code-Feature Mapping stage of VEGA is particularly efficient, completing in about 1,200 seconds automatically without manual intervention. The Model Creation stage takes approximately 72 hours. In practical applications, the Target-Specific Code Generation stage is swift for new targets. As shown in Fig. 7, VEGA completes each of the seven function modules in just a few hundred seconds, facilitating the generation of backends with hundreds of functions within an hour. Specifically, VEGA efficiently generates backends for RISC-V, RI5CY, and xCORE in 1,383 seconds, 1,664 seconds, and 424 seconds, respectively—all under one hour.

**Accuracy.** We assessed VEGA's accuracy using pass@1 (function-level evaluation) on the seven function modules shown in Fig. 1, with the results provided in Fig. 8. In testing an LLVM backend with pass@1, each VEGA-generated function substitutes its original counterpart. If a function fails any test case, it is subjected to a detailed manual review guided by its confidence score, with the duration depending on the developer's expertise. For each function module corresponding to a target, the purple bar indicates the percentage of functions accurately synthesized from the statements of various existing targets. Note that DIS is absent for xCORE, as explained in Sec. 4.1.4.

VEGA demonstrates remarkable effectiveness, achieving accuracy rates of up to 83.3% (OPT), 84.2% (SCH), and 85.7% (EMI) across the seven function modules (as shown in Fig. 1) for RISC-V, RI5CY, and xCORE, respectively. It is worth noting that VEGA shows relatively high accuracy in OPT, SCH, and EMI for all three targets. On average, this results in accuracy rates of 72.3%, 71.5%, and 67.2% for all generated functions across these three targets.

In VEGA, many functions are generated accurately despite having confidence scores below 1.00, suggesting that no manual adjustments are actually necessary. This cautious approach is evident in the REG module for RISC-V, where about 71.4% of functions are generated accurately. Among these, approximately 53.6% achieve confidence scores of 1.00, while the remaining 46.4% score below 1.00.

The purple bars highlight that a significant portion of accurate code, averaging 14.8%, 13.5%, and 4.0% across the seven function modules for RISC-V, RI5CY, and xCORE, respectively, is derived from multiple existing targets. For instance, the `applyFixup` function created for RISC-V incorporates elements from PowerPC and AMDGPU, reflecting similarities in byte order and instruction format. This highlights VEGA's proficiency in efficiently managing complex customization.

While VEGA is effective, it still necessitates manual intervention due to a non-negligible portion of inaccurate code. As shown by the gray areas in Fig. 8, VEGA generates code with errors, with some functions having incorrect statements

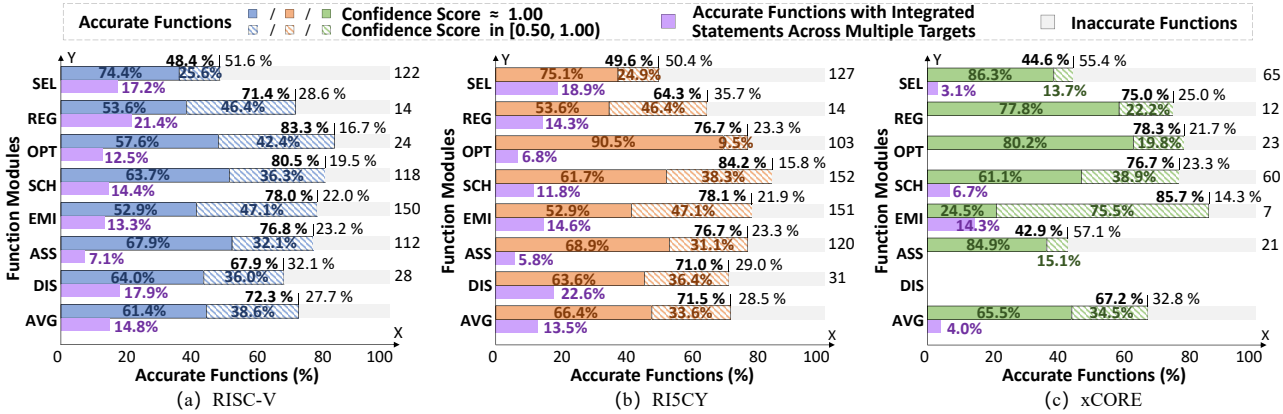



**Figure 8.** Accuracy of VEGA in generating functions across seven modules (measured by pass@1). Note: In the legend, "Confidence Score ≈ 1.00" indicates a confidence score greater than 0.99.

**Table 2.** Three sources of inaccurate statements.

| Error Type | RISC-V | RI5CY | xCORE | Statements | |
|---|---|---|---|---|---|
| 1. Err-V | 3.9% | 3.0% | 1.1% | Opcode==RISCVISD::VLSRFB_MASK | ✗ |
| | | | | Opcode==RISCVISD::VLSEGFF_MASK | ✓ |
| 2. Err-CS | 11.6% | 10.6% | 10.1% | ~~return MCDisassembler::Success;~~ 0.12 | ✗ |
| | | | | return MCDisassembler::Success; | ✓ |
| 3. Err-Def | 23.9% | 22.9% | 37.2% | + OutMI.setOpcode(RVV->BaseInstr); | ✓ |

that fail verification using pass@1. On average, inaccurate code comprises 27.7% for RISC-V, 28.5% for RI5CY, and 32.8% for xCORE across the seven function modules, corresponding to 28.5%, 26.8%, and 37.8% of all functions, respectively.

As shown in Table 2, VEGA generates three types of inaccuracies: (1) statements with incorrect target-specific values (**Err-V**), (2) statements with contradicting confidence scores (correct when scores are below 0.5 but incorrect when scores are 0.5 or higher) (**Err-CS**), and (3) deficient statements due to missing necessary elements (**Err-Def**). The percentages represent the proportion of functions with each error type relative to all generated functions. Totals may exceed 100% as a single function can show multiple errors simultaneously.

The majority of errors (**Err-Def**) stem from missing necessary statements. **Err-V** contributes the least, ranging from 1.1% to 3.9% across the three targets, due to the UniXcoder model used. For the first example, in RISC-V, the incorrect value "RFB" was generated instead of "EGFF" based on probability estimation of sequential characters. Implementing domain-specific heuristics that prioritize relevant terms in backends might mitigate this issue. Additionally, **Err-CS** and **Err-Def** errors are linked to the code coverage of VEGA's function templates. In another example, a low confidence score (0.12) might lead developers to manually review a correct statement. The final example illustrates a failure in generation by VEGA, requiring manual intervention.

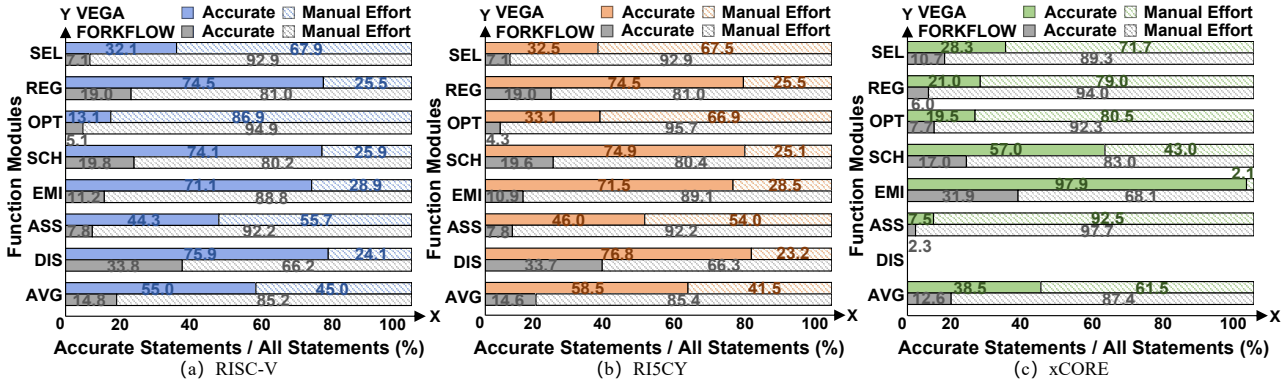
In addition to the function-group-based method for creating the training dataset (Sec. 4.1.2), we also tested a backend-based alternative. This approach allocates 75% of compiler backends to the training set and the remaining 25% to verification, risking coverage of all 825 function templates. Consequently, this split could omit some templates from either set, affecting VEGA's accuracy. This alternative method led to an average accuracy reductions of 26.2% for RISC-V, 25.2% for RI5CY, and 11.1% for xCORE, underscoring its limitations for VEGA. Thus, we have adopted the function-group-based scheme during our training stage.

**Comparing with ForkFlow.** We compared VEGA with the traditional fork-flow approach, denoted ForkFlow [44, 60], which involves forking a function from an existing backend and modifying it for a new backend. Specifically, we manually developed backends for RISC-V, RI5CY, and xCORE by forking from the most similar architecture, MIPS. ForkFlow produced only 45, 47, and 4 accurate functions for these targets, with average accuracy rates of 6.6%, 5.6%, and 2.7% across the seven function modules. In contrast, VEGA achieved significantly higher average accuracy rates of 72.3%, 71.5%, and 67.2% as shown in Fig. 8. Including all functions generated, ForkFlow's accuracy rates were 7.9%, 6.7%, and 2.1%, compared to VEGA's 71.5%, 73.2%, and 62.2%.

In Fig. 9, we assess VEGA and ForkFlow by counting the number of accurate statements. For functions deemed accurate, all statements are counted as accurate. For each inaccurate function, we repeatedly modify its statements until it passes accuracy checks by pass@1. Statements that remain unmodified during this process are considered accurate.

In Fig. 9, the differences in the "Accurate" bars between VEGA and ForkFlow highlight the significant productivity gains achieved by using VEGA over ForkFlow. ForkFlow demands substantial manual efforts, as indicated by its "Manual Effort" bars, where over 85% of statements require manual modification or supplementation. This results in average rates of 85.2%, 85.4%, and 87.4% (at the statement level) for RISC-V, RI5CY, and xCORE, respectively. For unique modules closely tied to customized hardware characteristics, such as OPT, the manual effort required exceeds 90%, making it almost equivalent to completing an entire module by hand.

VEGA achieves superior statement-level average accuracy rates: 55.0% for RISC-V, 58.5% for RI5CY, and 38.5% for




(a) RISC-V (b) RI5CY (c) xCORE

**Figure 9.** Comparing VEGA and ForkFlow for productivity gains (in terms of accuracy at the statement level).

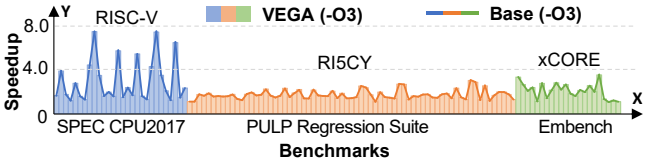



**Figure 10.** Backend performance.

xCORE across seven function modules. It notably surpasses ForkFlow in four specific modules — REG, SCH, EMI, and DIS, where accuracy rates for RISC-V and RI5CY exceed 70%, greatly reducing manual effort. Even in the highly customized OPT module, VEGA outperforms ForkFlow with 33.1% accuracy for RI5CY compared to ForkFlow's 4.3%. Analysis indicates VEGA effectively utilizes existing backends, like Hexagon's, which is optimized for RI5CY, highlighting VEGA's ability to streamline development and optimization for rapidly evolving processors within same ISA.

**Manual Effort Required for VEGA.** Let us evaluate the manual effort needed to correct VEGA-generated compiler backends. After examining inaccuracies in functions and statements for RISC-V, RI5CY, and xCORE, we detail the specific corrections made to the RISC-V backend.

Based on the data from Fig. 9, Table 3 details the number of statements accurately generated ("Accurate") and those needing manual corrections ("Manual Effort") across seven function modules for RISC-V, RI5CY, and xCORE. VEGA generated 5,524 statements for RISC-V, 6,996 for RI5CY, and 1,071 for xCORE, significantly outperforming ForkFlow, which produced 1,195, 1,342, and 370 accurate statements for these targets, respectively (Fig. 9). This underscores VEGA's enhanced productivity over traditional methods. Although some code requires manual adjustments, these are typically isolated to specific functions, as evidenced by the high proportion of accurate functions (Fig. 8). VEGA's average accuracy rates of 72.3% for RISC-V, 71.5% for RI5CY, and 67.2% for xCORE far surpass ForkFlow's 6.6%, 5.6%, and 2.7%, notably easing the overall development effort.

To assess manual corrections for VEGA-generated compiler backends, our analysis focused on the RISC-V backend, initially comprising 10,013 lines of C++ code. Table 4 details efforts by two LLVM developers, who had no prior involvement in this research project. Developer A, a third-year PhD candidate specializing in LLVM compiler mid-ends, spent 42.54 hours obtaining a RISC-V backend with 12,696 lines of C++ code. Developer B, a compiler engineer with two years in RISC-V-specific performance testing and analysis, spent 48.12 hours achieving a similar result with 13,675 lines of C++ code. They estimated 120–176 hours would be required using the ForkFlow approach [44, 60], based on VEGA's and ForkFlow's accuracy rates of 71.5% and 7.9%, respectively.

Both developers reported positively, noting that about 71.5% of VEGA-generated functions required no manual modifications. Most corrections focused on the SEL and OPT modules, which exhibited significant deficiencies (**Err-Def**) and required extensive correction time (Fig. 9). Despite the time investment, the localized errors allowed for targeted corrections, leveraging expertise in architectures like MIPS and ARM to expedite the process. VEGA-generated confidence scores facilitated quick identification and modification or removal of inaccurate statements, substantially reducing error detection time. In the SEL and OPT modules, low confidence scores enabled developers to undertake extensive or complete rewrites promptly, avoiding detailed debugging. Moreover, VEGA's synthesized templates correctly specify names, parameters, and types for target-specific functions, even when some statements are not accurately generated. This streamlines corrections and reduces development time compared to building a backend from scratch.

### 4.3 VEGA's Compilation Ability

For the three targets considered, we refer to the new compilers with the newly composed backends as $\texttt{VEGA}^{RISC-V}$, $\texttt{VEGA}^{RI5CY}$, and $\texttt{VEGA}^{xCORE}$, respectively.

**Robustness.** After manually correcting the inaccurate functions, $\texttt{VEGA}^{RISC-V}$, $\texttt{VEGA}^{RI5CY}$, and $\texttt{VEGA}^{xCORE}$ were subjected to the same regression tests as before (Sec. 4.1.3), executed under the `-O3` optimization level. Each of these

**Table 3.** Statements accurately generated ("Accurate") and requiring manual correction ("Manual Effort") by VEGA across seven function modules for RISC-V, RI5CY, and xCORE.

| | RISC-V | | RI5CY | | xCORE | |
|---|---|---|---|---|---|---|
| Function Modules | Accurate | Manual Effort | Accurate | Manual Effort | Accurate | Manual Effort |
| SEL | 1,770 | 3,747 | 1,811 | 3,765 | 515 | 1,306 |
| REG | 102 | 35 | 102 | 35 | 59 | 222 |
| OPT | 181 | 1,204 | 1,329 | 2,690 | 76 | 314 |
| SCH | 803 | 281 | 929 | 311 | 255 | 192 |
| EMI | 1,447 | 589 | 1,509 | 602 | 46 | 1 |
| ASS | 1,041 | 1,310 | 1,127 | 1,323 | 120 | 1,481 |
| DIS | 180 | 57 | 189 | 57 | – | – |
| **ALL** | **5,524** | **7,223** | **6,996** | **8,783** | **1,071** | **3,516** |

**Table 4.** Manual corrections to the VEGA-generated RISC-V backend, performed by two LLVM developers: Developer A, a third-year PhD candidate specializing in compiler mid-ends, and Developer B, a compiler engineer with two years in RISC-V-specific performance testing and analysis.

| Function Modules | Developer A (Hours) | Developer B (Hours) |
|---|---|---|
| SEL | 21.83 | 17.47 |
| REG | 0.41 | 0.39 |
| OPT | 7.23 | 10.87 |
| SCH | 3.17 | 3.04 |
| EMI | 4.15 | 7.47 |
| ASS | 5.17 | 7.90 |
| DIS | 0.58 | 0.98 |
| **ALL** | **42.54** | **48.12** |

compilers produced results in line with their respective base compilers when run on the appropriate simulators.

**Performance.** VEGA$^{RISC-V}$, VEGA$^{RI5CY}$, and VEGA$^{xCORE}$ were evaluated using 28 SPEC CPU2017 C/C++ benchmarks, 69 PULP regression tests, and 22 Embench cases, as depicted in Fig. 10. The performance of each compiler (shown by speedup bars under "-O3", normalized against "-O0") closely matches that of its respective base compiler (represented by speedup curves under "-O3", also normalized to "-O0").

## 5 Related Work

**AI Programming.** The rise of "AI-generated" applications has significantly advanced code generation capabilities, as evidenced by AI models such as those from Amazon CodeWhisperer, OpenAI Codex, and others [53, 64, 67]. Particularly, pre-trained models like Code Llama [63] and UniXcoder [27] have shown exceptional proficiency. Research into LLMs for code generation has demonstrated substantial promise in automating these tasks [5, 10, 16, 17, 38, 68, 70, 77]. While advanced conversational models like Copilot [65] and ChatGPT [52] may be explored to generate backends, they require manual, carefully-engineered prompts, unlike VEGA, which fully automates the workflow.

**AI in Compilation.** The rapid progress of AI has accelerated the adoption of AI-based compilation. AI techniques have been employed for various tasks, including designing cost models [48, 59, 62, 79, 80], determining transformation order [11, 21, 40, 54, 71], and optimizing parallel programs [33, 34, 37, 72–75]. In particular, recent projects utilizing transformer models for decompilation[3, 4] and code optimization [14, 25, 26] have underscored the potential of AI for compilers.

**Retargetable Compilers.** Modern compiler development has been influenced by target-independent code generation and processor design languages (PDLs), such as generators commonly used in compilers [1, 12, 35, 51, 81], and those applied to hardware accelerators [6, 30, 45, 69, 78]. In addition, automatic generation has also had an impact on the peripheral area around compilers, specifically in the domain of ASIPs using PDLs [7, 8, 19, 29, 31, 46, 55].

## 6 Conclusion

In this paper, we introduce VEGA, an advancement in compiler technology aimed at streamlining backend development for new targets. VEGA utilizes an AI-driven approach to automate the synthesis of features from existing backends, facilitating the creation of new backends using only target description files. Our evaluations across three distinct target processors illustrate that VEGA can reduce manual effort, potentially enhancing the speed and efficiency of compiler development and code generation.

However, VEGA is not without limitations. As an AI-driven tool, it may generate incorrect code, akin to errors in developer-produced code due to the limited variety of compiler backends it has been exposed to. To counteract this, we propose implementing a software update mechanism to enhance its inferential accuracy by learning from newly synthesized function templates. Additionally, we are exploring program synthesis techniques to broaden its functional scope. To accommodate specialized hardware features, such as RVV in RISC-V [2], we plan to integrate reinforcement learning techniques and progressively enrich the dataset to fully support the capabilities of such custom hardware. Rigorous testing is crucial to uncover inaccuracies, which will inform the refinement of VEGA in subsequent iterations.

## 7 Data-Availability Statement

The artifact is publicly available at [82].

## Acknowledgement

We would like to thank all anonymous reviewers for their insightful feedback. This work was supported by National Key R&D Program of China, Grant No. 2022ZD0116316. It was also supported by the Strategic Priority Research Program of the Chinese Academy of Sciences, Grant No. XDB0660102, the National Natural Science Foundation of China, Grant No. U23B2020, No. 62090024, No. 62302479 and the Innovation Funding of ICT, CAS under Grant No. E361010.

## References


[1] Maaz Bin Safeer Ahmad, Alexander J. Root, Andrew Adams, Shoaib Kamil, and Alvin Cheung. 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* (Lausanne, Switzerland) *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 1004–1016. https://doi.org/10.1145/3503222.3507714

[2] RISC-V The Open-Standard Instruction Set Architecture. 2024. RISCV-V-SPEC. https://github.com/riscv/riscv-v-spec/blob/master/v-spec.adoc.

[3] Jordi Armengol-Estapé and Michael O'Boyle. 2021. Learning C to x86 Translation: An Experiment in Neural Compilation. In *Advances in Programming Languages and Neurosymbolic Systems Workshop*. https://openreview.net/forum?id=444ug_EYXet

[4] Jordi Armengol-Estapé, Jackson Woodruff, Chris Cummins, and Michael F.P. O'Boyle. 2024. SLaDe: A Portable Small Language Model Decompiler for Optimized Assembly. In *2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, Edinburgh, United Kingdom, 67–80. https://doi.org/10.1109/CGO57630.2024.10444788

[5] Jason Blocklove, Siddharth Garg, Ramesh Karri, and Hammond Pearce. 2023. Chip-Chat: Challenges and Opportunities in Conversational Hardware Design. https://arxiv.org/abs/2305.13243. arXiv:2305.13243 [cs.LG]

[6] Thomas Bourgeat, Clément Pit-Claudel, Adam Chlipala, and Arvind. 2020. The Essence of Bluespec: A Core Language for Rule-Based Hardware Design. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 243–257. https://doi.org/10.1145/3385412.3385965

[7] Florian Brandner, Viktor Pavlu, and Andreas Krall. 2013. Automatic generation of compiler backends. *Software: Practice and Experience* 43, 2 (2013), 207–240. https://doi.org/10.1002/spe.2106

[8] Gunnar Braun, Achim Nohl, Weihua Sheng, Jianjiang Ceng, Manuel Hohenauer, Hanno Scharwächter, Rainer Leupers, and Heinrich Meyr. 2004. A Novel Approach for Flexible and Consistent ADL-Driven ASIP Design. In *Proceedings of the 41st Annual Design Automation Conference* (San Diego, CA, USA) *(DAC '04)*. Association for Computing Machinery, New York, NY, USA, 717–722. https://doi.org/10.1145/996566.996763

[9] Kaiyan Chang, Kun Wang, Nan Yang, Ying Wang, Dantong Jin, Wenlong Zhu, Zhirong Chen, Cangyuan Li, Hao Yan, Yunhao Zhou, Zhuoliang Zhao, Yuan Cheng, Yudong Pan, Yiqi Liu, Mengdi Wang, Shengwen Liang, yinhe han, Huawei Li, and Xiaowei Li. 2024. Data is all you need: Finetuning LLMs for Chip Design via an Automated design-data augmentation framework. In *2024 61th ACM/IEEE Design Automation Conference (DAC)*. IEEE, SAN FRANCISCO, CA, USA.

[10] Kaiyan Chang, Ying Wang, Haimeng Ren, Mengdi Wang, Shengwen Liang, Yinhe Han, Huawei Li, and Xiaowei Li. 2023. ChipGPT: How far are we from natural language hardware design. https://arxiv.org/abs/2305.14019. arXiv:2305.14019 [cs.AI]

[11] Junjie Chen, Ningxin Xu, Peiqi Chen, and Hongyu Zhang. 2021. Efficient Compiler Autotuning via Bayesian Optimization. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE Computer Society, Madrid, ES, 1198–1209. https://doi.org/10.1109/ICSE43902.2021.00110

[12] Yishen Chen, Charith Mendis, Michael Carbin, and Saman Amarasinghe. 2021. VeGen: A Vectorizer Generator for SIMD and Beyond. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS 2021)*. Association for Computing Machinery, New York, NY, USA, 902–914. https://doi.org/10.1145/3445814.3446692

[13] Standard Performance Evaluation Corporation. 2024. SPEC CPU 2017. https://www.spec.org/cpu2017/.

[14] Chris Cummins, Volker Seeker, Dejan Grubisic, Mostafa Elhoushi, Youwei Liang, Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Kim Hazelwood, Gabriel Synnaeve, and Hugh Leather. 2023. Large Language Models for Compiler Optimization. https://arxiv.org/abs/2309.07062. arXiv:2309.07062 [cs.PL]

[15] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, Jill Burstein, Christy Doran, and Thamar Solorio (Eds.). Association for Computational Linguistics, Minneapolis, Minnesota, 4171–4186. https://doi.org/10.18653/v1/N19-1423

[16] Elizabeth Dinella, Todd Mytkowicz, Alexey Svyatkovskiy, Christian Bird, Mayur Naik, and Shuvendu Lahiri. 2023. DeepMerge: Learning to Merge Programs. *IEEE Transactions on Software Engineering* 49, 4 (2023), 1599–1614. https://doi.org/10.1109/TSE.2022.3183955

[17] Hantian Ding, Varun Kumar, Yuchen Tian, Zijian Wang, Rob Kwiatkowski, Xiaopeng Li, Murali Krishna Ramanathan, Baishakhi Ray, Parminder Bhatia, and Sudipta Sengupta. 2023. A Static Evaluation of Code Completion by Large Language Models. In *Proceedings of the The 61st Annual Meeting of the Association for Computational Linguistics: Industry Track, ACL 2023, Toronto, Canada, July 9-14, 2023*, Sunayana Sitaram, Beata Beigman Klebanov, and Jason D. Williams (Eds.). Association for Computational Linguistics, Toronto, Canada, 347–360. https://doi.org/10.18653/V1/2023.ACL-INDUSTRY.34

[18] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-Grained and Accurate Source Code Differencing. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering* (Vasteras, Sweden) *(ASE '14)*. Association for Computing Machinery, New York, NY, USA, 313–324. https://doi.org/10.1145/2642937.2642982

[19] Stefan Farfeleder, Andreas Krall, Edwin Steiner, and Florian Brandner. 2006. Effective Compiler Generation by Architecture Description. In *Proceedings of the 2006 ACM SIGPLAN/SIGBED Conference on Language, Compilers, and Tool Support for Embedded Systems* (Ottawa, Ontario, Canada) *(LCTES '06)*. Association for Computing Machinery, New York, NY, USA, 145–152. https://doi.org/10.1145/1134650.1134671

[20] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*. Association for Computational Linguistics, Online, 1536–1547. https://doi.org/10.18653/v1/2020.findings-emnlp.139

[21] Grigori Fursin and Olivier Temam. 2011. Collective Optimization: A Practical Collaborative Approach. *ACM Trans. Archit. Code Optim.* 7, 4, Article 20 (dec 2011), 29 pages. https://doi.org/10.1145/1880043.1880047

[22] Michael Gautschi, Pasquale Davide Schiavone, Andreas Traber, Igor Loi, Antonio Pullini, Davide Rossi, Eric Flamand, Frank K. Gürkaynak, and Luca Benini. 2017. Near-Threshold RISC-V Core With DSP Extensions for Scalable IoT Endpoint Devices. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 25, 10 (2017), 2700–2713. https://doi.org/10.1109/TVLSI.2017.2654506

[23] GCC. 2023. GNU Compiler Collection. https://gcc.gnu.or.

[24] Hong-Na Geng, Fang Lv, Ming Zhong, Hui-Min Cui, Jingling Xue, and Xiao-Bing Feng. 2023. Automatic Target Description File Generation. *Journal of Computer Science and Technology* 6, 38 (2023), 1339–1355. https://doi.org/10.1007/s11390-022-1919-x

[25] Dejan Grubisic, Chris Cummins, Volker Seeker, and Hugh Leather. 2024. Compiler generated feedback for Large Language Models. arXiv:2403.14714 [cs.PL]

[26] Dejan Grubisic, Volker Seeker, Gabriel Synnaeve, Hugh Leather, John Mellor-Crummey, and Chris Cummins. 2024. Priority Sampling of Large Language Models for Compilers. In *Proceedings of the 4th Workshop on Machine Learning and Systems* (, Athens, Greece,) *(EuroMLSys '24)*. Association for Computing Machinery, New York, NY, USA, 91–97. https://doi.org/10.1145/3642970.3655831

[27] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. UniXcoder: Unified Cross-Modal Pre-training for Code Representation. https://aclanthology.org/2022.acl-long.499. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, Dublin, Ireland, 7212–7225. https://doi.org/10.18653/v1/2022.acl-long.499

[28] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin B. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. GraphCodeBERT: Pre-training Code Representations with Data Flow. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, Virtual Event, Austria, 18 pages. https://openreview.net/forum?id=jLoC4ez43PZ

[29] Ashok Halambi, Peter Grun, Vijay Ganesh, Asheesh Khare, Nikil Dutt, and Alex Nicolau. 1999. EXPRESSION: A Language for Architecture Exploration through Compiler/Simulator Retargetability. In *Proceedings of the Conference on Design, Automation and Test in Europe* (Munich, Germany) *(DATE '99)*. Association for Computing Machinery, New York, NY, USA, 100–es. https://doi.org/10.1145/307418.307549

[30] Tae Jun Ham, Lisa Wu, Narayanan Sundaram, Nadathur Satish, and Margaret Martonosi. 2016. Graphicionado: A high-performance and energy-efficient accelerator for graph analytics. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society, Taipei, Taiwan, 1–13. https://doi.org/10.1109/MICRO.2016.7783759

[31] Manuel Hohenauer, Felix Engel, Rainer Leupers, Gerd Ascheid, and Heinrich Meyr. 2009. A SIMD Optimization Framework for Retargetable Compilers. *ACM Trans. Archit. Code Optim.* 6, 1, Article 2 (apr 2009), 27 pages. https://doi.org/10.1145/1509864.1509866

[32] M. I. Jordan. 1986. Serial order: A parallel distributed processing approach. *ICS-Report 8604 Institute for Cognitive Science University of California* 121 (1986), 64.

[33] Wookeun Jung, Thanh Tuan Dao, and Jaejin Lee. 2021. DeepCuts: A Deep Learning Optimization Framework for Versatile GPU Workloads. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 190–205. https://doi.org/10.1145/3453483.3454038

[34] Yiping Kang, Johann Hauswald, Cao Gao, Austin Rovinski, Trevor Mudge, Jason Mars, and Lingjia Tang. 2017. Neurosurgeon: Collaborative Intelligence Between the Cloud and Mobile Edge. https://doi.org/10.1145/3037697.3037698. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (Xi'an, China) *(ASPLOS '17)*. Association for Computing Machinery, New York, NY, USA, 615–629. https://doi.org/10.1145/3037697.3037698

[35] Theodoros Kasampalis, Daejun Park, Zhengyao Lin, Vikram S. Adve, and Grigore Roşu. 2021. Language-Parametric Compiler Validation with Application to LLVM. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS 2021)*. Association for Computing Machinery, New York, NY, USA, 1004–1019. https://doi.org/10.1145/3445814.3446751

[36] Sumith Kulal, Panupong Pasupat, Kartik Chandra, Mina Lee, Oded Padon, Alex Aiken, and Percy S Liang. 2019. SPoC: Search-based Pseudocode to Code. In *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.), Vol. 32. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2019/file/7298332f04ac004a0ca44cc69ecf6f6b-Paper.pdf

[37] Benjamin C. Lee and David M. Brooks. 2006. Accurate and Efficient Regression Modeling for Microarchitectural Performance and Power Prediction. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, California, USA) *(ASPLOS XII)*. Association for Computing Machinery, New York, NY, USA, 185–194. https://doi.org/10.1145/1168857.1168881

[38] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K. Lahiri, and Siddhartha Sen. 2023. CodaMosa: Escaping Coverage Plateaus in Test Generation with Pre-trained Large Language Models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, Melbourne, Australia, 919–931. https://doi.org/10.1109/ICSE48619.2023.00085

[39] 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 (2022), 1092–1097. https://doi.org/10.1126/science.abq1158

[40] Hongzhi Liu, Jie Luo, Ying Li, and Zhonghai Wu. 2021. Iterative Compilation Optimization Based on Metric Learning and Collaborative Filtering. *ACM Trans. Archit. Code Optim.* 19, 1, Article 2 (dec 2021), 25 pages. https://doi.org/10.1145/3480250

[41] LLVM. 2021. getRelocType Function Reference(ARM). https://llvm.org/doxygen/ARMELFObjectWriter_8cpp_source.html.

[42] LLVM. 2023. Clang Lexer Reference. https://clang.llvm.org/doxygen/classclang_1_1Lexer.html.

[43] LLVM. 2023. LLVM Documentations. "http://llvm.org/docs".

[44] LLVM. 2023. Writing an LLVM Backend. https://llvm.org/docs/WritingAnLLVMBackend.html.

[45] Divya Mahajan, Jongse Park, Emmanuel Amaro, Hardik Sharma, Amir Yazdanbakhsh, Joon Kyung Kim, and Hadi Esmaeilzadeh. 2016. TABLA: A unified template-based framework for accelerating statistical machine learning. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE Computer Society, Barcelona, Spain, 14–26. https://doi.org/10.1109/HPCA.2016.7446050

[46] P. Marwedel. 1984. The MIMOLA Design System: Tools for the Design of Digital Processors. In *21st Design Automation Conference Proceedings*. IEEE Computer Society, Albuquerque, NM, USA, 587–593. https://doi.org/10.1109/DAC.1984.1585857

[47] JACK MCCREA. 2020. GETTING STACK SIZE JUST RIGHT ON XCORE. https://llvm.org/devmtg/2020-09/slides/McCrea-Getting_Stack_Size_Just_Right_on_XCore.pdf.

[48] Charith Mendis, Alex Renda, Saman Amarasinghe, and Michael Carbin. 2019. IThemal: Accurate, Portable and Fast Basic Block Throughput Estimation using Deep Neural Networks. In *36th International Conference on Machine Learning, ICML 2019 (36th International Conference on Machine Learning, ICML 2019)*. International Machine Learning Society (IMLS), LA, USA, 7908–7918.

[49] Haoran Mo, Edgar Simo-Serra, Chengying Gao, Changqing Zou, and Ruomei Wang. 2021. General Virtual Sketching Framework for Vector Line Art. *ACM Trans. Graph.* 40, 4, Article 51 (jul 2021), 14 pages. https://doi.org/10.1145/3450626.3459833

[50] Reiichiro Nakano. 2019. Neural Painters: A learned differentiable constraint for generating brushstroke paintings. http://arxiv.org/abs/1904.08410.

[51] Rachit Nigam, Samuel Thomas, Zhijing Li, and Adrian Sampson. 2021. A Compiler Infrastructure for Accelerator Generators. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS 2021)*. Association for Computing Machinery, New York, NY, USA, 804–817. https://doi.org/10.1145/3445814.3446712

[52] OpenAI. 2022. ChatGPT: Optimizing Language Models for Dialogue. https://openai.com/blog/chatgpt/.

[53] OpenAI. 2023. OpenAI Codex. https://openai.com/blog/openai-codex

[54] Sunghyun Park, Salar Latifi, Yongjun Park, Armand Behroozi, Byungsoo Jeon, and Scott Mahlke. 2022. SRTuner: Effective Compiler Optimization Customization by Exposing Synergistic Relations. In *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE Computer Society, Seoul, Korea, 118–130. https://doi.org/10.1109/CGO53902.2022.9741263

[55] Stefan Pees, Andreas Hoffmann, Vojin Zivojnovic, and Heinrich Meyr. 1999. LISA—Machine Description Language for Cycle-Accurate Models of Programmable DSP Architectures. In *Proceedings of the 36th Annual ACM/IEEE Design Automation Conference* (New Orleans, Louisiana, USA) *(DAC '99)*. Association for Computing Machinery, New York, NY, USA, 933–938. https://doi.org/10.1145/309847.310101

[56] PULP. 2021. PULP Project. https://github.com/pulp-platform.

[57] PULP. 2021. PULP Regression Test Cases. https://github.com/pulp-platform/regression_tests.

[58] QEMU. 2023. QEMU RISC-V Simulator. https://wiki.qemu.org/Documentation/Platforms/RISCV.

[59] Martin Rapp, Anuj Pathania, Tulika Mitra, and Jörg Henkel. 2021. Neural Network-Based Performance Prediction for Task Migration on S-NUCA Many-Cores. *IEEE Trans. Comput.* 70, 10 (2021), 1691–1704. https://doi.org/10.1109/TC.2020.3023022

[60] Ayushi Rastogi and Nachiappan Nagappan. 2016. Forking and the Sustainability of the Developer Community Participation – An Empirical Investigation on Outcomes and Reasons. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. IEEE Computer Society, Osaka, Japan, 102–111. https://doi.org/10.1109/SANER.2016.27

[61] RISCV. 2023. RISCV Official Website. https://riscv.org/.

[62] Fabian Ritter and Sebastian Hack. 2020. PMEvo: Portable Inference of Port Mappings for out-of-Order Processors by Evolutionary Optimization. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 608–622. https://doi.org/10.1145/3385412.3385995

[63] 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]

[64] Amazon Web Services. 2023. Amazon CodeWhisperer. %https://aws.amazon.com/codewhisperer

[65] Dominik Sobania, Martin Briesch, and Franz Rothlauf. 2022. Choose your programming copilot: a comparison of the program synthesis performance of github copilot and genetic programming. In *Proceedings of the Genetic and Evolutionary Computation Conference* (Boston, Massachusetts) *(GECCO '22)*. Association for Computing Machinery, New York, NY, USA, 1019–1027. https://doi.org/10.1145/3512290.3528700

[66] Embench standards group. 2024. Embench: A Modern Embedded Benchmark Suite. https://www.embench.org/.

[67] Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. 2020. IntelliCode compose: code generation using transformer. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 1433–1443. https://doi.org/10.1145/3368089.3417058

[68] Alexey Svyatkovskiy, Sarah Fakhoury, Negar Ghorbani, Todd Mytkowicz, Elizabeth Dinella, Christian Bird, Jinu Jang, Neel Sundaresan, and Shuvendu K. Lahiri. 2022. Program Merge Conflict Resolution via Neural Transformers. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Singapore, Singapore) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 822–833. https://doi.org/10.1145/3540250.3549163

[69] Xifan Tang, Edouard Giacomin, Baudouin Chauviere, Aurélien Alacchi, and Pierre-Emmanuel Gaillardon. 2020. OpenFPGA: An Open-Source Framework for Agile Prototyping Customizable FPGAs. *IEEE Micro* 40, 4 (2020), 41–48.

[70] Shailja Thakur, Baleegh Ahmad, Zhenxing Fan, Hammond Pearce, Benjamin Tan, Ramesh Karri, Brendan Dolan-Gavitt, and Siddharth Garg. 2023. Benchmarking Large Language Models for Automated Verilog RTL Code Generation. In *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 1–6.

[71] Jack Turner, Elliot J. Crowley, and Michael F. P. O'Boyle. 2021. Neural Architecture Search as Program Transformation Exploration. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 915–927. https://doi.org/10.1145/3445814.3446753

[72] S. VenkataKeerthy, Siddharth Jain, Umesh Kalvakuntla, Pranav Sai Gorantla, Rajiv Shailesh Chitale, Eugene Brevdo, Albert Cohen, Mircea Trofin, and Ramakrishna Upadrasta. 2024. The Next 700 ML-Enabled Compiler Optimizations. In *Proceedings of the 33rd ACM SIGPLAN International Conference on Compiler Construction (CC 2024)*. Association for Computing Machinery, New York, NY, USA, 238–249. https://doi.org/10.1145/3640537.3641580

[73] S. VenkataKeerthy, Siddharth Jain, Anilava Kundu, Rohit Aggarwal, Albert Cohen, and Ramakrishna Upadrasta. 2023. RL4ReAl: Reinforcement Learning for Register Allocation. In *Proceedings of the 32nd ACM SIGPLAN International Conference on Compiler Construction (CC 2023)*. Association for Computing Machinery, New York, NY, USA, 133–144. https://doi.org/10.1145/3578360.3580273

[74] Zheng Wang, Dominik Grewe, and Michael F. P. O'boyle. 2014. Automatic and Portable Mapping of Data Parallel Programs to OpenCL for GPU-Based Heterogeneous Systems. *ACM Trans. Archit. Code Optim.* 11, 4, Article 42 (dec 2014), 26 pages. https://doi.org/10.1145/2677036

[75] Jaeyeon Won, Charith Mendis, Joel S. Emer, and Saman P. Amarasinghe. 2023. WACO: Learning Workload-Aware Co-optimization of the Format and Schedule of a Sparse Tensor Program. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*, Tor M. Aamodt, Natalie D. Enright Jerger, and Michael M. Swift (Eds.). ACM, New York, NY, USA, 920–934. https://doi.org/10.1145/3575693.3575742

[76] XMOS. 2021. xCORE Processor. https://www.xmos.ai/xcore-200/.

[77] Prateek Yadav, Qing Sun, Hantian Ding, Xiaopeng Li, Dejiao Zhang, Ming Tan, Parminder Bhatia, Xiaofei Ma, Ramesh Nallapati, Murali Krishna Ramanathan, Mohit Bansal, and Bing Xiang. 2023. Exploring Continual Learning for Code Generation Models. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki (Eds.). Association for Computational Linguistics, Toronto, Canada, 782–792. https://doi.org/10.18653/v1/2023.acl-short.68

[78] Drew Zagieboylo, Charles Sherk, Gookwon Edward Suh, and Andrew C. Myers. 2022. PDL: A High-Level Hardware Design Language for Pipelined Processors. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 719–732. https://doi.org/10.1145/3519939.3523455

[79] Yi Zhai, Yu Zhang, Shuo Liu, Xiaomeng Chu, Jie Peng, Jianmin Ji, and Yanyong Zhang. 2023. TLP: A Deep Learning-Based Cost Model for Tensor Program Tuning. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 833–845. https://doi.org/10.1145/3575693.3575737

[80] Jiepeng Zhang, Jingwei Sun, Wenju Zhou, and Guangzhong Sun. 2020. An Active Learning Method for Empirical Modeling in Performance Tuning. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE Computer Society, New Orleans, LA, USA, 244–253. https://doi.org/10.1109/IPDPS47924.2020.00034

[81] Tuowen Zhao, Tobi Popoola, Mary Hall, Catherine Olschanowsky, and Michelle Strout. 2022. Polyhedral Specification and Code Generation of Sparse Tensor Contraction with Co-Iteration. *ACM Trans. Archit. Code Optim.* 20, 1, Article 16 (dec 2022), 26 pages. https://doi.org/10.1145/3566054

[82] Ming Zhong. 2024. VEGA. https://doi.org/10.5281/zenodo.14064392

Received 2024-09-10; accepted 2024-11-04