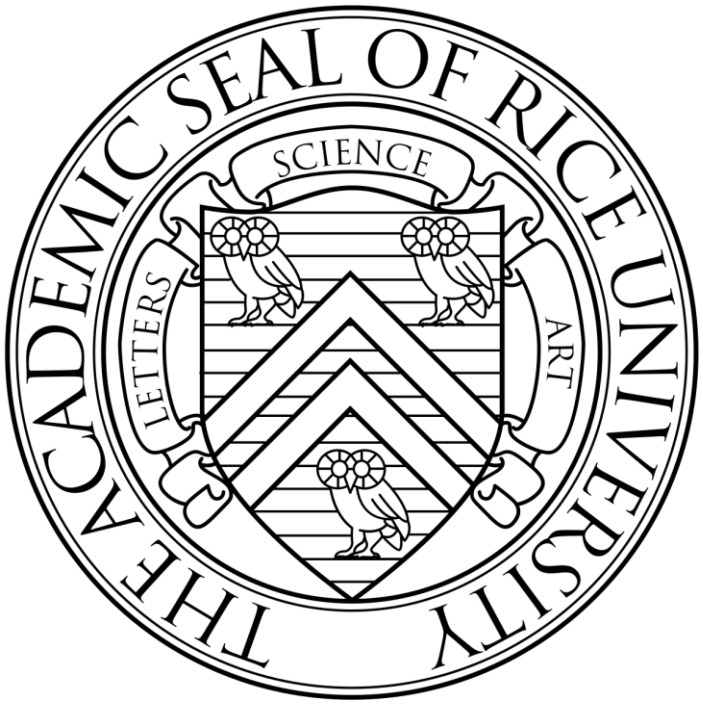
DOCTORAL DISERTATION

Optimizing Compiler Heuristics with Machine Learning



Dejan Grubišić

April, 2024

RICE UNIVERSITY

Optimizing Compiler Heuristics with Machine Learning

By

Dejan Grubisic

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE

Doctor of Philosophy

APPROVED, THESIS COMMITTEE

John Mellor-Crummey
John Mellor-Crummey (Apr 16, 2024 15:59 CDT)

John Mellor-Crummey (Chair)

Professor of Computer Science and of
Electrical and Computer Engineering

Christopher Jermaine
Christopher Jermaine (Apr 16, 2024 15:03 CDT)

Christopher M. Jermaine

Professor of Computer Science

Ray Simar
Ray Simar (Apr 16, 2024 15:02 CDT)

Ray Simar

Professor in the Practice of Electrical and
Computer Engineering

HOUSTON, TEXAS

April 2024

ABSTRACT

Optimizing Compiler Heuristics with Machine Learning

by

Dejan Grubisic

Compiler technology is crucial for enhancing the performance and efficiency of modern software. The complexity of novel computer architectures, the ever-evolving software landscape, and the ever-growing scale of computation have made manual optimization techniques increasingly difficult and time-consuming. To address this, machine learning (ML) can recognize intricate patterns and automatically tailor code generation and optimization strategies for specific hardware configurations, significantly enhancing program performance. This thesis demonstrates these ideas.

First, we showcase the use of reinforcement learning in optimizing tensor computations with LoopTune. LoopTune optimizes tensor traversal order while using the ultra-fast lightweight code generator LoopNest to perform hardware-specific optimizations. With a novel graph-based representation and action space, LoopTune speeds up LoopNest by 3.2x, generating an order of magnitude faster code than TVM, 2.8x faster than MetaSchedule, and 1.08x faster than AutoTVM, consistently performing at the level of the hand-tuned library Numpy.

Second, we pioneer the use of large language models (LLMs) in compiler optimization. Our model generates optimization in seconds, achieving a 3.0% improvement in reducing instruction counts over the compiler, outperforming two state-of-the-art baselines that require thousands of compilations. Even more, the model shows sur-

prisingly strong code reasoning abilities, generating compilable code 91% of the time and perfectly emulating the output of the compiler 70% of the time.

Third, we evaluate feedback-directed LLMs that use compiler feedback collected in inference time to improve generated code. We evaluate three feedback formats with various degrees of information, which all outperform the original model by 0.11%, 0.4%, and 0.53%. We further combine this approach with temperature-based sampling and iterative compilation. Sampling techniques show superior performance, reaching 98% of autotuner's performance over the compiler given the budget of 100 samples.

Fourth, we present Priority Sampling, a simple deterministic LLM sampling technique that produces unique samples ordered by the model's confidence. Priority Sampling outperforms Nucleus Sampling for any number of samples, reducing the code size further than the original model and achieving a 5% reduction over -Oz instead of 2.87%. Moreover, it outperforms the autotuner used for the generation of labels for the training of the original model in just 30 samples.

Acknowledgments

I want to thank all the people who were consistently helping me during my PhD journey, teaching me how to do research, collaborate, and grow as a person and scientist. Without these amazing people, I couldn't publish papers, present my work, and finally write this thesis.

I want to thank my advisor Professor John Mellor-Crummey who mentored me through the whole PhD and spent numerous hours discussing and debugging complex problems in the development of profiling tools. John introduced me to the world of research and showed me what an organized research group looks like, and how to collaborate, listen, and focus common efforts toward reaching the common goal. During 5 years working with John, I had the opportunity to learn the intricacies of parallel and distributed computing, optimize various problems for diverse hardware architectures, and get familiar with open questions and cutting-edge research in high-performance computing. Besides technical work, John taught me how to present my work and write scientific papers, and provided valuable feedback for writing this thesis.

I would like to thank the following people from the group whom I had the pleasure of working with:

- Laksono Adhianto, who helped me integrate GPU Idleness Analysis to HPCViewer.
- Jonathon Anderson, who developed a second-generation, highly multithreaded HPCToolkit's post-mortem data analysis and made it scalable for large-scale applications.
- Aaron Cherian, who collaborated with me on the development of a GPU tracing

architecture.

- Vladimir Indjic, who was the main contributor with me on the Compiler2 project.
- Mark Krentel, who helped refactor the loading of the HPCToolkit measurement subsystem.
- Yumeng Liu, who implemented HPCToolkit's sparse metrics representation.
- Xiaozhu Meng, who has collaborated with me on adding support for profiling AMD GPUs.
- Ryuichi Sai, who developed optimizations for high-order stencils on GPUs.
- Keren Zhou, who developed GPU Binary Analysis, GPU Advisor, and helped me numerous times to understand GPU tracing infrastructure.

Besides my incredible group, I had the opportunity to do an internship at Berkeley Lab and two internships at Meta AI, collaborating with the following amazing people:

- Brian Austin (Berkeley Lab), who mentored me in developing support for GPU power analysis in HPCToolkit.
- Aleksandar Zlateski and Bram Wasti (Meta AI), who mentored me for the LoopTune project and implemented LoopTool and LoopNest.
- Chris Cummins, Volker Seeker, and Hugh Leather (Meta AI), who were mentoring and collaborating with me on the "Large Language Models in Compilers" project.

In addition to research and engineering collaboration, I would like to thank my family and friends, without whom, I wouldn't have been able to go through the PhD study. To my parents, Predrag and Snežana, for being a constant source of love and

support, brother Dušan who was always there for me, my grandma Venetka, and foremost my wife Emily.

Last, I would like to thank Professor Ray Simar and Christopher Jermaine for being members of the Thesis Committee together with my advisor John Mellor-Crummey. Without their great support and guidance, I wouldn't be able to write my thesis successfully and finish my PhD.

Contents

Illustrations

Tables

Chapter 1

Introduction

1.1 Optimizing Compiler Heuristics

Computer systems play an integral role in the contemporary world. They enable communication over the internet, transportation using a global positioning system (GPS), control manufacturing processes, monitoring medical conditions, and many more. At the heart of these systems lie compilers, essential tools that translate high-level programming languages into machine code understandable by computers. Compilers are crucial in optimizing code for efficiency and performance, ensuring that software runs smoothly and effectively on diverse hardware platforms. Their ability to bridge the gap between human-readable code and machine-executable instructions is fundamental to the functionality and success of modern computing environments.

Designing and optimizing compilers is a complex problem. Compilers need to know how to parse high-level programming languages, optimize their representations while maintaining the semantics, and interact with the low-level intricacies of underlying hardware. Modern compilers, such as LLVM [12], implement thousands of rules and algorithms in over 1M lines of C++ code to achieve this. Managing and maintaining such a code base requires significant effort. Additionally, the pace of developing new hardware increased the need for flexible compiler designs that could be easily optimized for any new architecture.

Traditional compiler design methodology [13] doesn't provide a satisfactory solu-

tion. According to this methodology, compiler engineers design optimization passes that transform the compiler's intermediate representation (IR) into more efficient implementation. These optimizations include eliminating redundant computation, increasing memory access to nearby elements, or reordering instructions to utilize hardware resources effectively. Some optimizations, such as *loop-unroll* and *inline*, include configuration parameters that must be tuned for different applications to obtain optimal performance.

Optimizing many of these problems is NP-complete, which means finding the optimal solution in polynomial time is impossible. This is why compiler engineers design hand-crafted heuristics that provide performant implementations most of the time. Writing compiler heuristics requires expertise in hardware design and iterative empirical evaluation. Once a set of heuristics works well for most applications, they are hard-coded into the compiler code base and shipped to production.

The problem with this approach is that heuristics don't optimize efficiently all programs. For example, programs that contain a lot of calls for small functions could benefit from more aggressive inlining, while programs that contain loops could benefit from unrolling and vectorizing on a specific type of hardware. Additionally, the ordering of optimization passes is heuristic, resulting in different programs' performances. Finally, many heuristics are hard-coded in the evergrowing compiler code base and even become forgotten [14].

Instead of optimizing heuristics manually, compiler engineers found that iteratively compiling programs with various heuristic configurations could be about 2.3x more effective [15]. Section 2.1 describes an extensive collection of prior work that explores genetic algorithms, greedy search, and other design-space exploration techniques for compiler optimization. The problem with these approaches is that they

require running a search from scratch for each new program. This introduces significant resource constraints and impedes applications that must be optimized in real time.

Machine learning (ML) presents an attractive approach to overcoming the problem of repeated search. Instead of searching for the heuristic parameters for each program, the ML provides a mechanism to map the input program to the most performant heuristic configuration given enough data. This practically replaces a time-consuming search with the single feed-forward inference of the machine learning model. The compute-intensive part is moved to the training phase, which must be executed only once.

Machine learning, however, comes with a set of challenges. First, the input program must be represented as a vector of constant size. This vector must capture the program's unique characteristics relevant to the predicting task. Next, most machine learning models, such as neural networks, require a lot of training data and predictive labels to achieve satisfactory prediction accuracy. Although a large amount of code is available online on platforms such as Github, generating labels such as execution time would require building projects automatically and creating appropriate input data, which is extremely difficult. Finally, using ML to write compiler optimizations requires that the produced code doesn't change the semantics of the input program. This is hard to achieve because machine learning models are inherently probabilistic.

This thesis aims to demonstrate the practical use of machine learning in compiler optimization. It proposes novel ML architectures that outperform human experts and current state-of-the-art methods, evaluates their capabilities, and discusses their limitations.

1.2 Thesis Statement

Machine learning is a viable methodology for code optimization that could be used to predict performant optimization strategies that don't change program semantics. To prove this thesis, we provide the evidence for the following three claims:

1. Reinforcement Learning can help optimize the execution time of tensor programs by predicting tensor traversal order (loop schedule) and using a backend compiler to execute it to the given hardware.

2. Large Language Models can efficiently optimize code size by predicting the sequence of LLVM's optimizations by directly using the input program's text representation of LLVM IR, for programs that fit in the model's context size.

3. The performance of Large Language Models in predicting LLVM's optimization sequence can exceed the performance of optimization sequences used for training.

1.3 Contributions

This thesis includes the following contributions:

- It describes the design, implementation, and evaluation of LoopTune, a deep reinforcement learning framework that optimizes tensor computations in deep learning models for the CPU. LoopTune optimizes tensor traversal order while using the ultra-fast lightweight code generator LoopNest to perform hardware-specific optimizations. With a novel graph-based representation and action space, LoopTune speeds up LoopNest by 3.2x, generating an order of magnitude faster code than TVM, 2.8x faster than MetaSchedule, and 1.08x faster than AutoTVM, consistently performing at the level of the hand-tuned library Numpy. Moreover, LoopTune tunes code in order of seconds.

- It is the first study to evaluate the use of large language models for predicting LLVM optimization sequence. The model takes as input unoptimized assembly and predicts a list of compiler options to optimize the code size of the program. Crucially, during training, we ask the model to predict the instruction counts before and after optimization and the optimized code itself. This approach achieves a 3.0% improvement in reducing instruction counts over the compiler, outperforming two state-of-the-art baselines requiring thousands of compilations. Furthermore, the model shows surprisingly strong code reasoning abilities, generating compilable code 91% of the time and perfectly emulating the output of the compiler 70% of the time.

- It proposes and evaluates feedback-directed LLMs that use the compiler to evaluate the response by the LLM and provide feedback. LLM then uses the feedback and tries again. We consider this method for three different feedback formats and assess how this method works with sampling techniques. Our approach adds an extra 0.53% of the improvement over -Oz to the original model. Nevertheless, it doesn't significantly exceed the original model's performance when combined with the sampling.

- It introduces Priority Sampling, a simple deterministic sampling technique for LLMs that produces unique samples ordered by the model's confidence. Each new sample expands the unexpanded token with the highest probability in the augmented search tree. Priority Sampling outperforms Nucleus Sampling for any number of samples, boosting the performance of the original model from 2.87% to 5% improvement over -Oz and outperforming the autotuner used for the generation of labels for the training of the original model in just 30 samples.

1.4 Thesis Structure

Chapter 2 describes the context for this work. It discusses the history of compiler development, explains the design of modern compilers, and describes previous work and limitations in machine learning focused on optimizing compilers. Chapter 3 presents a LoopTool, a reinforcement learning based framework that optimizes loop schedules. Chapter 4 presents the use of large language models for predicting LLVM optimization sequence. Chapter 5 introduces the idea of a feedback-directed method that evaluates the response of the LLM and provides the feedback. Chapter 6 presents Priority Sampling, a novel LLM sampling method that guarantees unique samples ordered by the model's confidence. Chapter 7 summarizes our findings, provides critical discussion, and outlines potential research directions.

Chapter 2

Background

This chapter serves two purposes. First, it provides a historical overview of compiler development, interaction with hardware and programming languages, and runtime systems. It presents the design of the modern compiler toolchain LLVM [12] and automatic optimization techniques that motivated machine learning based autotuning. Second, it describes the development of machine learning techniques in compiler design, which serve as the foundation for the research described in this dissertation.

2.1 A Brief History of Computer Hardware, Programming Languages, and Compilers

The development of the first microprocessors marked a significant breakthrough in computer architecture. In the early 1970s, companies such as Intel and AMD introduced the first mass-produced chips, consolidating the functions of multiple discrete components into a single integrated circuit. This innovation paved the way for creating more compact and powerful computers that are pervasive components of today's society.

Advancements in semiconductor technology, storage devices, and networking components have collectively contributed to the exponential growth in computing power. The constant pursuit of miniaturization increased processing speed and improved en-

ergy efficiency by roughly doubling the number of transistors every two years. This pace of development, known as Moore's Law, continued for more than 50 years, reducing the size of transistors to about 5nm in 2020, reaching the limit of Silicon technology [16].

Transistor scaling allowed for a higher density of transistors on a chip, leading to more sophisticated and intricate logic circuits. Directly manipulating these circuits became impractical and error-prone for human programmers [17]. This gave rise to the need for a higher-level abstraction that could express computational tasks more clearly and structured.

In response to this need, the first programming languages emerged. These languages provided a bridge between human understanding and machine execution. Languages such as Fortran [18] and COBOL [19] enabled programmers to express computational tasks in a syntax closer to natural language, making it easier to conceptualize and write code. This abstraction facilitated the translation of human-designed algorithms into machine-executable instructions.

The relationship between transistor scaling and programming languages was symbiotic. The increasing complexity of logic circuits made it imperative to have higher-level programming languages to manage and control these intricate circuits efficiently. At the same time, transistor scaling enabled increases in clock frequency that accelerated the program execution.

To bridge the gap between programming languages and complex hardware circuitry, compiler technology became necessary. The purpose of compilers was to translate and optimize a high-level program to binary code that the machine can execute. Compilers need to perform a series of complex tasks to do this job successfully. First, they need to ensure that a program adheres to the rules of the programming lan-

guage's grammar or provide feedback if this is not the case. Second, they need to tailor high-level program abstractions to various hardware backends, utilizing diverse hardware components while preserving the program's original intent.

Compiler development was always closely connected to hardware innovations and applications of interest. In the early days, compilers focused on basic code optimization techniques such as peephole optimization, constant folding, and algebraic simplifications [20, 21]. These techniques aimed to improve the generated machine code by simplifying and optimizing the intermediate representation.

In the 1970s, computers could run numeric simulations, which are highly important for a wide range of scientific disciplines [22–24]. Since the main component of these applications was implemented with loops, compilers started to analyze and optimize loops for better performance. Some techniques developed during this period include loop unrolling, loop fusion, and loop-invariant code motion [25–27]. For the first time, vector instructions, popularized by Cray [28], enabled executing multiple operations simultaneously.

The 1980s were marked by the rise of Object-Oriented Programming (OOP), which profoundly impacted the development of compilers and influenced both language design and the compilation process. Languages such as Smalltalk [29], Simula [30], and later, C++ [31] and Java [32], introduced features such as classes, objects, encapsulation, inheritance, and polymorphism, which made the compiler design process more challenging. Nevertheless, developing these compilers allowed programmers to write more complex programs and reusable code, improving productivity and making it easier to maintain large software systems.

To address the requirements of the OOP compiler, intermediate representations were redesigned to capture the structure and behavior of classes and objects. Dynamic

binding and late binding became prevalent to optimize the loading of the classes. Techniques such as vtable-based method dispatch [33] were introduced to manage polymorphic behavior, while sophisticated type-checking passes were implemented to ensure the correctness of object-oriented programs.

When it comes to compiler optimizations, the 1980s saw the proliferation of IR-based manipulations including data-flow and control-flow analysis [34], common subexpression elimination [35], inlining [36], and even software pipelining [37] which exploits instruction-level parallelism within loops. Additionally, considerable effort was focused on optimizing register allocation [38] and utilizing efficiently fast registers. Techniques such as graph coloring for register allocation were introduced to minimize memory access.

The 1990s witnessed a shift towards interprocedural optimization, where compilers analyzed and optimized code across function boundaries [39]. New algorithms for global common subexpression elimination [40], function cloning [41], and cross-function analysis were developed to improve the performance of entire programs. Compilers began automatically transforming scalar code into vectorized code [42] to leverage Single Instruction Multiple Data (SIMD) instructions, enhancing parallelism and performance.

The gap between processor speed and memory access times widened as processors became faster, leading to memory latency issues. To bridge this gap, increasing cache sizes became a common strategy. Larger caches were introduced to store frequently accessed data and instructions, reducing the need to fetch data from slower main memory. Techniques such as cache blocking [43], loop transformations [44], and prefetching enhanced spatial and temporal locality, minimizing cache misses and improving overall cache efficiency.

The 1990s also saw fruitful developments in programming languages. Java [32] was developed by Sun Microsystems, C++ [31] and Objective-C [45] were officially standardized, interpreted languages such as Python [46] and R [47] emerged, while languages such as Haskell [48] gained attention for their expressive power and functional paradigm features. Java Virtual Machine (JVM) [49] enabled Java to be platform independent, while just-in-time(JIT) [50] compilation enabled collecting the program's behavior during execution and dynamically optimizing the code accordingly.

The 2000s witnessed an explosion in web development, with the Internet becoming a ubiquitous application platform. This shift increased the demand for programming languages and frameworks that efficiently build dynamic and interactive web applications. JavaScript [51] emerged as the dominant language for client-side using Representational State Transfer (REST) [52] API while scripting languages gained prominence for building dynamic web applications. PHP [53], Python (Django) [54], Ruby (Ruby on Rails) [55], Java, C# [56], and DotNet [57] became popular choices for server-side development.

The internet-driven demands for faster development cycles and deployment led to the rise of DevOps practices. Automation and continuous integration tools became integral to the development process, influencing the choice of programming languages and workflows. On the other hand, handling user requests and the emergence of cloud computing brought scalability challenges for web applications. Programming languages and frameworks that supported horizontal scaling and distributed architectures became crucial. Languages such as Erlang [58] and Go [59] gained attention for their concurrency and parallelism features.

The LLVM (Low-Level Virtual Machine) [12] project proposed a modern, flexible compiler infrastructure to consolidate compiler design. LLVM's design allowed for

efficient code generation, optimization, and support for multiple programming languages and became the foundation for various compilers, including Clang [60] for C and C++.

Optimizing for larger cache sizes continued with the development of multi-core processors. With multiple cores sharing caches, optimizing for cache coherence [61] and minimizing contention [62, 63] became crucial. Additionally, compilers started using runtime profiles to guide optimizations [64, 65], tailoring the generated code based on actual program behavior. This allowed compilers to make more informed decisions about hot paths and frequently executed code.

As the pace of transistor scaling slowed down, increasing processor frequency became infeasible. At the same time, the demand for computational power increased exponentially. This challenge prompted numerous researchers to experiment with Graphics Processing Units (GPUs) for general-purpose computing tasks, extending beyond their traditional graphics-focused applications. In 2007, NVIDIA introduced CUDA (Compute Unified Device Architecture) [66], along with comprehensive programming models tailored for parallel processing. AMD announced its Stream Computing initiatives [67]. This opened up new horizons, empowering developers to leverage GPUs' immense parallel processing capabilities for diverse tasks, ranging from scientific simulations and data processing to the forefronts of artificial intelligence (AI).

With increased parallel processing capabilities on GPUs and the availability of enormous datasets on the Internet, neural networks have become incredibly popular since they can successfully learn intricate patterns from raw data. The crucial moment for further development of machine learning happened in 2012. when Alex Krizhevsky et al. proposed a deep convolutional network (CNN) [68] that could sort 1.2 million

high-resolution images into 1,000 different classes. Another breakthrough happened in 2017 when Vaswani et al. proposed a novel transformer architecture [69] that could generate and classify arbitrary text without any human-written heuristic.

These developments in machine learning reinforced the need for scalable software infrastructure for the training, which led to the development of PyTorch [70] and Tensorflow [71], as well as languages for high-performance scientific computing such as Julia [72]. These frameworks enabled significant improvement in the standardization and reproducibility of machine learning research, leading to numerous new inventions.

To improve the performance of deep learning models, PyTorch and Tensorflow implement numerous domain-specific optimizations. Both employ kernel fusion [73] to optimize the execution of operations on GPUs and utilize Tensor Cores when available. They automatically tailor kernel parameters to given hardware, leveraging parallelism and asynchronous execution while optimizing memory layout to enhance memory access patterns and reduce latency. They optimize the computation graph [74] and reduce the overall computational workload to eliminate redundant operations.

With the emergence of large language models [75], additional effort was invested in scaling up infrastructure and enabling fast execution across many nodes. Since models larger than 8B parameters are too big to fit on a single GPU, such as Nvidia's A100, it is necessary to implement fast communication protocols and efficient algorithms for multi-node execution. Projects such as PyTorch Distributed [76], Tensorflow [71], Ray [77], Horovod [78], DeepSpeed [79], Jax [80] were on forefront of these inovations.

Besides ML applications, significant effort is invested in building the systems for processing large amounts of data, known as Big Data Architectures. Scaling up Big Data Architectures involves distributing and managing computational workloads across clusters of machines or containers. Frameworks that support scalability in big

data environments often work seamlessly with container orchestration platforms such as Docker [81] and Kubernetes [82] and distributed data processing frameworks such as Apache Spark [83], Hadoop [84], Kafka [85], Dask [86] and many more.

In contrast to large-scale systems optimizations, the innovations in consumer electronics, robotics, and automotive increased the need for real-time processing with limited power supply. Virtual Reality (VR) [87] and Augmented Reality (AR) [88] headsets are capable of rendering 3D scenes and interacting with the user in real-time. Similarly, the development of self-driving cars [89] highly depends on fast rendering of the car's environment and using inference on neural networks to decide where the car should go. These systems require challenging real-time processing while being optimized for power consumption, which will undoubtedly lead to new hardware and compiler innovations.

Each decade brought new challenges and opportunities, driving the development of compiler optimization techniques to keep pace with advancements in hardware architecture and programming practices. These optimizations collectively contribute to improving code execution efficiency across a diverse range of applications. In the following chapters, we will examine the structure of modern compilers and how machine learning can help compiler design to keep up with these challenges.

2.2 Modern Compiler Design

As described in the previous section, compilers must deal with a complex interplay between programming languages and underlying hardware infrastructure. Modern compilers are separated into three modules - Frontend, Optimizer, and Backend (Figure 2.1) to make this task manageable. Such a modular approach enables fast integration of new programming languages and hardware architectures by simply writing

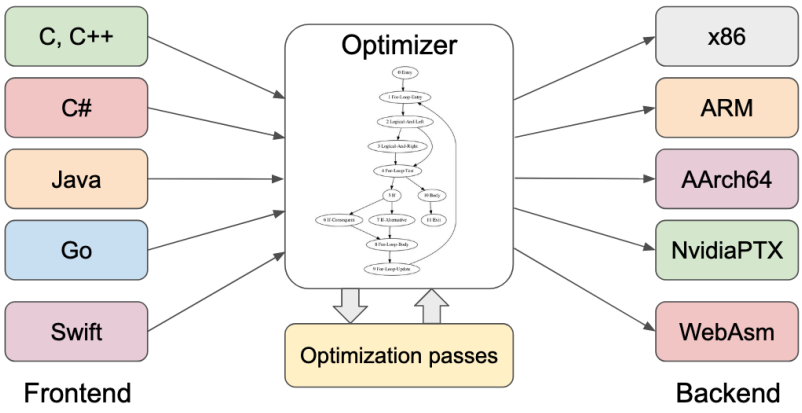


Figure 2.1 : Compiler modules. Frontends parse the input program to an IR, the Optimizer optimizes the IR with optimization passes, and Backends translate the IR to a hardware binary.

custom Frontends and Backends while reusing the core component – Optimizer.

Frontends compile and analyze input programs into a platform-independent intermediate representation (IR). Compilation begins with a lexical analysis or scanning phase, breaking the source code into tokens. Next, the parsing phase analyzes the syntax and structure of the code, producing an Abstract Syntax Tree (AST) that captures relationships between program elements. The AST is the basis for generating IR, a platform-independent and structured code form.

An Optimizer takes the IR generated by a Frontend and applies a series of transformations to enhance the efficiency of the code. These transformations include loop optimization, inlining of functions, and data flow analysis. The goal is to produce optimized code that executes faster, uses memory more efficiently, and minimizes redundant computations. Optimizations are crucial for maximizing the performance of the compiled program.

Backends apply hardware-dependent heuristics and translate IR to executable

code. The compiler then proceeds to the code generation phase, where the optimized IR is translated into assembly code specific to the target architecture. Finally, the assembly code undergoes further hardware-specific optimizations. It is translated into machine code through the assembly and linking processes, resulting in an executable binary that embodies the functionality of the original high-level source code.

To manage the complexity of optimizations, compilers typically partition optimizations into independent optimization passes or flags, and the combination of these passes contributes to the generation of optimized code. Each optimization pass focuses on a specific aspect of code improvement. While many optimization passes are standalone and operate without requiring additional parameters, some, like *loop-unroll* or *vectorize*, are parametric and require the specification of an integer value. Using parametric optimization passes enhances the adaptability of compilers, enabling developers to strike a balance between the desire for performance gains and potential trade-offs in code size or compile time.

The size of the optimization space is vast. Production compilers such as GCC [90] have more than 200 compiler flags on which we apply optimization selection problems. This makes the optimization space 2^{200} if we assume that all flags are standalone. The optimization space is even larger in reality. Similarly, LLVM [12] defines more than 150 optimization passes applied to Phase-ordering problems for optimization space of size 150^L, where L is the sequence length. These numbers are infeasible for simple enumeration, and navigating such optimization spaces requires careful exploration.

2.3 Autotuning in Compilers

Experimenting with different optimization flags, either informally or systematically, through exploration, compiler developers have found that simple search strategies

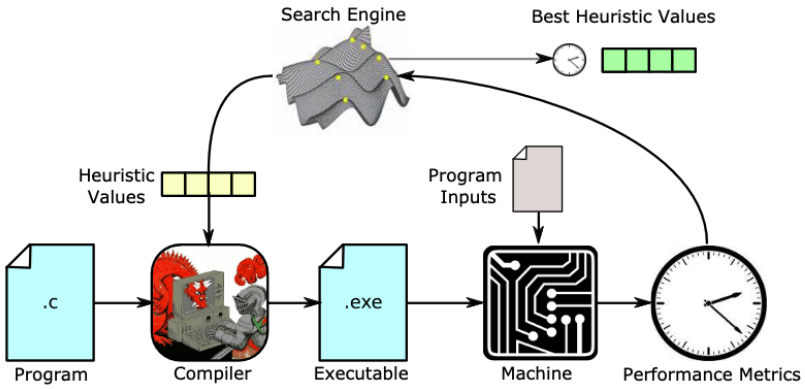


Figure 2.2 : Iterative Compilation. The compiler iteratively evaluates various sequences of optimization passes or heuristic values for a given time allocation. [1]

can outperform hand-written compiler heuristics (Figure 2.2). The core idea involves defining a space of optimization strategies, such as unrolling and tiling and iteratively evaluating various optimization parameters until we find the most effective one. The iterative compilation is platform-agnostic, evidence-based, and able to achieve substantial performance gains on diverse sets of programs [15].

There is a large body of work on iterative compilation, with each investigation focusing on a different heuristic or alternative search technique. Commonly used methods include random and greedy searches [91, 92], design-space exploration techniques [93–95] and genetic algorithms [15, 96–101].

Various compiler heuristics were targeted with these approaches, including phase ordering [96, 99, 102–104], multi-objective exploration [91], selecting loop transformations [105], optimizing for code size [97], and others.

Libraries such as ATLAS [106] and SPIRAL [107] optimize linear algebra and signal processing algorithms by empirically considering factors such as cache sizes, memory hierarchy, and instruction set architecture. Similarly, notable frameworks

such as PetaBricks [108] and LIFT [109] employ rule-based optimization systems that allow users to define transformation rules for algorithmic patterns. This approach enables users to define and optimize parallel patterns commonly found in numerical algorithms, such as map, reduce, and stencil computations [110, 111].

The inherent problem of iterative compilation is that it needs to be performed independently for each program. This introduces a significant computational overhead, increasing compilation times and resource utilization, especially for large software projects.

2.4 Machine Learning for Code Optimization

To understand the contribution of this thesis, it is necessary to understand what is machine learning. This section explains the machine learning methodology and elaborates on its application in optimizing compiler heuristics.

Machine learning (ML) is a scientific discipline that focuses on the development of algorithms and models that enable computers to learn patterns and make predictions or decisions without being explicitly programmed. To learn these patterns, we define a model with parameters *theta* that we adjust during a training procedure to map a feature vector to a given class we are trying to predict (Figure 2.3). Once training is done, our model should be able to predict the label of unseen feature vectors, under the assumption that it comes from a similar distribution like training data.

Feature Vector - Characterization of input program

In the context of compiler optimization, the feature vector is a set of observable attributes that we use to describe the input program. To describe the program successfully, the feature vector 1) must capture important properties in the input

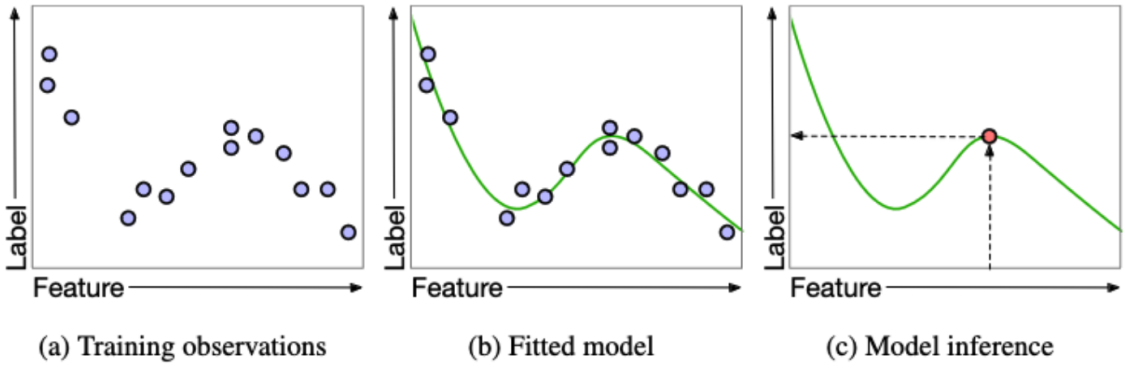


Figure 2.3 : Ilustration of training and inference of machine learning [2]. In (a), training observations have been collected, consisting of features and their corresponding classes we are trying to predict. A model is then fitted to these observations, shown as the green curve in (b). The model can then be used to infer the label of unseen feature values, shown in (c).

program and 2) must uniquely describe the input program. Poorly selected feature vectors could impede the learning process and significantly reduce the precision of the trained model. To characterize the input program, we define static and dynamic features.

Static features of a program include the number of loops, arithmetic instructions, code size, number of leaf nodes in its IR, or any other metric derived from the input program, its IR, or assembly code. These features are usually cheap to collect since they are always available at compile time. Cavazos et. al [112, 113] build compiler performance models based on static features and optimize programs to reduce their execution time. Agakov et. al [114] show that 36 commonly used code features can be reduced with Principal Component Analysis(PCA) to just 5 features preserving 99% of the variance in data. AutoPhase [115] parses a 56-dimensional feature vector

for programs and uses reinforcement learning to predict the best optimization passes.

Besides features based on source code, graph-based features are particularly useful since they can model definition-use relationships between variables [116, 117]. Park et. al [118] constructed features by calculating shortest path graph kernels from each basic block in IR to predict program performance. Projects such as Milepost GCC [119] and LLVM's Opt [12] enable feature extraction on source-code and IR level, respectively.

Dynamic features include all the metrics that we can collect during execution such as execution time, cache misses, and arithmetic intensity. These features are valuable additions to static features since program behavior often depends on inputs that are unknown ahead of time. By multiplexing hardware counters Cavazos et. al [120] were able to collect 60 dynamic features used for predicting performant compiler heuristics. Tools such as PAPI [121], HPCToolkit [122], and GProf [123] provide convenient interfaces for collecting dynamic features. Combining both static and dynamic features outperforms each of them independently [124].

Model Architectures

Machine learning architecture is a mathematical model capable of finding a function that maps feature vectors to output labels. Over time, many architectures were proposed. Depending on whether they are used to predict a given class or generate new data, they are divided into 1) Discriminative models and 2) Generative models.

Discriminative models include Logistic Regression [125], Support Vector Machines [126], Decision Trees [127] and Random Forest [128], Feed-forward Neural Network [129], Graph Neural Network [130], Convolutional Neural Network [68] and Transformers [69]. When applied to the problem of code optimization, these models are primar-

ily used for predicting the performance of a given program, optimization sequence, or on what device a given program should be executed.

Generative models include Bayesian Networks [131], Variational Autoencoders (VAE) [132], Diffusion Models [133], and Generative Adversarial Networks [134] and Transformers. These models are commonly used for the generation of new synthetic training data. Note that transformers are in both categories since they could be used for both generation and classification. In the rest of this section, we provide an overview of each of these classes of model architectures.

Logistic regression is a simple binary classification task, aiming to predict the probability of an instance belonging to a particular class. It uses the logistic function to transform a linear combination of input features into a probability score, and a decision threshold is applied to make the final classification. In the realm of optimizing compilers, logistic regression was used to predict performant compiler optimizations given the program's dynamic features [120].

Support Vector Machines (SVM) aim to find an optimal hyperplane that maximally separates different classes in the feature space. By using a technique called the kernel trick [135] it can use linear methods to separate non-linear class problems while being exceptionally stable. SVMs are used to optimize JIT compilation [136], predicting unroll factors [137] and find the most performant optimization sequence given performance counters data [138].

Decision Trees recursively split the input data based on features, forming a tree structure of decision nodes that lead to final predictions. This approach provides a clean and interpretable learning interface, able to handle non-linear relationships between features and the target variable. On the negative side, small changes in the data can result in a different tree structure, making them less stable and one decision

tree may not capture complex relationships as effectively as other models. Decision trees are used to reduce the code size [139], learn to unroll the loops [140], and inline heuristic [112].

Random Forests leverage the aggregation of multiple trees creating an ensemble of decision trees to enhance predictive accuracy and mitigate overfitting. This approach mitigates overfitting by averaging predictions across multiple trees, improving generalization, and increasing flexibility. However, this comes with the cost of lower interpretability, and increased memory and compute requirements. Herrera et al. [141] use Random Forest to optimize Big Data processing and analytics.

Feed-forward Neural Networks, also known as a multilayer perceptron (MLP), are the first model to introduce the concept of hidden layers, enabling the network to capture complex hierarchical features in data representations (Figure 2.4). Each node takes a linear combination of previous nodes as an input, stored in weighted connections, after which it uses a non-linear activation. The most commonly used activation functions are Sigmoid, Hyperbolic Tangent, RELU, and leakyRELU [142].

The most widely used technique to train neural networks is Backpropagation [129]. Backpropagation involves the iterative adjustment of weights to minimize the difference between predicted and actual outputs. During each iteration, the algorithm computes gradients of the loss function with respect to the network's weights, representing the direction and magnitude of the steepest increase in the loss.

The learning rate determines the step size of weight updates, influencing the algorithm's convergence and stability. A higher learning rate may expedite convergence but risks overshooting the optimal weights, while a lower learning rate may enhance stability but slow down convergence. Backpropagation uses these gradients and learning rates to update weights, allowing the network to learn complex representations

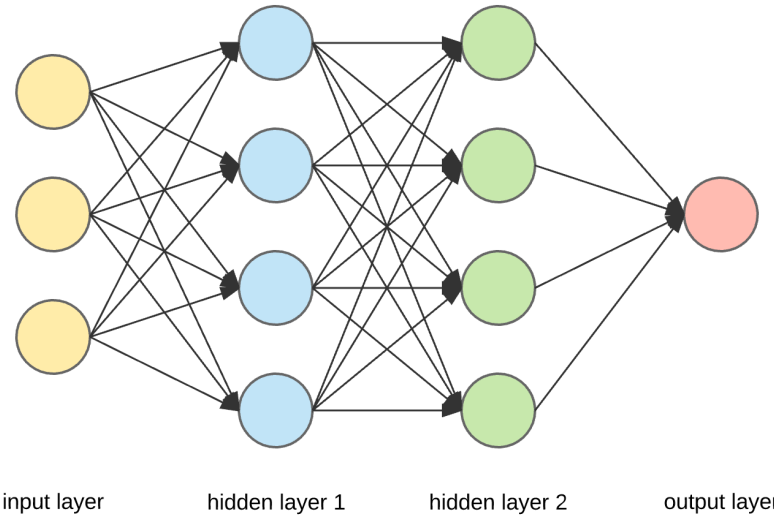


Figure 2.4 : Feed-forward neural network with two hidden layers. Each node contains a non-linear activation function. Each edge contains weight multiplied and summarized with values from previous nodes [3].

and improve its predictive capabilities over successive iterations. Careful tuning of these hyperparameters is crucial for achieving effective training and optimal neural network performance.

Optimization algorithms play a pivotal role in minimizing the loss function during the training process. Techniques such as Adam [143], RMSprop [144], and Adadelta [145] enhance the convergence speed and stability of the learning process by adapting the learning rates of individual parameters based on their historical gradients. The significance of these methods becomes apparent when dealing with complex models and large datasets, as they enable faster convergence and often help avoid issues like vanishing or exploding gradients.

Neural networks are also prone to over-fitting. When this happens, all parameters will be set to eliminate the loss on training data, while the loss on unseen validation

data will increase. To mitigate this problem, models use regularization techniques that enhance the generalization ability of neural networks, by introducing some kind of noise in data or the model. Popular regularization techniques include Dropout, L1/L2 regularization, data augmentations, and using ensemble methods.

Another commonly used regularization technique is Batch Normalization [146]. Batch Normalization addresses the internal covariate shift problem, where the distribution of activations in hidden layers changes during training, impacting convergence. It works by normalizing the input of each layer in a mini-batch to have zero mean and unit variance. Batch Normalization is applied independently to each layer, introducing learnable scale and shift parameters to allow the model to adapt and normalize data. Batch Normalization not only acts as a regularizer but also enables the use of higher learning rates and provides some degree of robustness to the choice of initialization parameters.

All of these techniques are ubiquitously used for more complex machine learning models that use backpropagation.

Convolutional Neural Networks (CNNs) are a specialized class of neural networks designed for processing grid-structured data, such as images. At a high level, CNNs operate by using convolutional layers to detect hierarchical patterns and spatial hierarchies within the input data. These layers consist of learnable filters that convolve over the input, capturing local features and gradually aggregating them to recognize complex patterns. Although CNNs are not typically applied for program optimization, Yang et. al [147] pioneered using CNN to identify compiler optimization levels in binary files, while Sharma et. al [148] used CNNs to evaluate the quality of the code. CNNs could be useful in analyzing the execution profile of parallel applications, but this research is yet to come.

Graph Neural Networks (GNNs) are designed to operate on graph-structured data, making them particularly effective for tasks such as program optimization that involve relational information and complex dependencies. At a high level, GNNs operate by iteratively aggregating information from neighboring nodes and updating node representations based on both node features and the graph structure. PrograML [149] proposes a graph-based representation based on compiler IRs and uses GNN to perform a set of 5 most commonly used compiler analyses. nGraph [150] passes its graph IR to a transformer and generates optimized code for the selected backend. XLA [151] automatically replaces subgraphs from Tensorflow with optimized binaries. Many more work have been proposed [152–158]

Reinforcement Learning (RL) sets up the optimization problem as a sequential decision process in which an agent interacts with an environment and learns to maximize a cumulative reward signal. The agent learns optimal strategies through exploration and exploitation, updating its policy and value function of the environment based on the observed outcomes. As a learning mechanism, RL can use any machine learning architecture including use decision trees, neural networks, and transformers.

This approach has become particularly popular in the compiler community since it provides a natural interface that enables the compiler to try and learn the best optimizations. Neurovectorizer [159] uses deep RL to improve the vectorization of CPU loops by tuning vectorization width and interleaving count. Chameleon [160] trains a policy network to guide an adaptive algorithm to sample well-performed parameters from configuration space. MLGO [161] uses Policy gradient and Evolution strategies to optimize binary size by inlining functions. PolyGym [162] explores loop schedules combining polyhedral representation with RL, while CompilerGym [163] enables the user to apply different RL algorithms for program optimization.

Recurrent Neural Networks (RNNs) and **Long Short-Term Memory (LSTM)** networks are specialized to process sequential data. RNNs are capable of capturing dependencies and patterns in sequential information by maintaining a hidden state that evolves. LSTMs, an extension of RNNs, address the vanishing gradient problem and facilitate the learning of long-term dependencies in sequences. At a high level, both RNNs and LSTMs operate by iteratively processing input sequences, updating their hidden states, and using these states to make predictions or capture relevant information.

Since they can process sequential data such as text, these methods are ideal tools for code optimization. Neural Programer [164] uses a small set of arithmetic and logic operations to perform complex arithmetic and logic reasoning, guided by RNN. Liang et. al [165] use two RNNs to map natural language to programs and use Lisp interpreter for execution. Wei et. al [166] use LSTMs for code generation and summarization, while Shido et. al [167] use LSTM to summarize code from abstract syntax tree.

Transformers are the groundbreaking architecture that upgraded RNNs and LSTMs by using just an attention mechanism. They process input sequences by attending to all positions simultaneously, enabling parallelization and efficient modeling of dependencies. The self-attention mechanism allows the model to weigh the importance of different elements in the input, capturing long-range dependencies effectively. Transformers consist of 2 elements: 1) Encoder - which applies bidirectional attention and transforms the input sequence in high dimensional embedding space, 2) Decoder - which applies masked attention and auto-regressively generates next tokens [69]. In addition to the original model, many alternative architectures were proposed including encoder-only and decoder-only transformers, even Graphormer [158] that accepts

graphs as input. Encoder-only transformers are usually used for classification tasks, and decoder-only- transformers are usually used for generation tasks.

Scaling up transformers to several billions of parameters led to the development of large language models (LLMs) that are heavily used for code generation. AlphaCode [168] uses exercises from programming competitions to generate code in Python and C++ from problem description. Models such as Code Llama [169], Codex [15], ChatGPT [170] and many more [171–176] are trained to perform multiple tasks including code generation, code search, code summarization, and documentation generation. LLMs trained on source code have also been used for program fuzzing [177–179], test generation [180], source-to-source translation [181], code weakness identification [182], and automated program repair [183, 184].

Challenges in Machine Learning for Compilers

Machine learning encompasses a range of issues that researchers and practitioners encounter while developing and deploying models. Key challenges include the demand for large labeled datasets to train robust models, issues related to the interpretability and explainability of complex algorithms, and the potential biases embedded in training data that might not contain representative data [185]. Additionally, a need for efficient transfer learning techniques [186] for adaptability on diverse architectures, and preserving semantic correctness further contribute to the intricate challenges [187].

Data availability. Unlike some well-explored domains in machine learning where large labeled datasets are readily accessible, compiler optimization tasks often lack diverse and comprehensive datasets. Gathering annotated data for compiler-specific optimizations is resource-intensive and may not capture the full spectrum of complexities present in real-world code. This scarcity of labeled data limits the capacity of

machine learning models to generalize across a wide range of programming languages, code structures, and optimization scenarios. Addressing the data availability problem requires innovative solutions, such as data augmentation techniques, transfer learning from related domains, and collaborations to establish shared repositories of compiler-specific datasets, ensuring that machine learning based compilers can robustly adapt to diverse code bases and optimization challenges.

To address this problem, two solutions were proposed: 1) Design a stochastic code generator, and 2) Scrape and build numerous Github projects. Compiler fuzzers, such as CSmith [188] and llvm-stress [12], represent convenient methods for the generation of diverse code examples with various properties. However, the test cases these generators produce are not meant to generate real-world code, which can limit their use for out-of-distribution programs. Alternatively, many machine-learning models were developed to generate diverse code examples from a given dataset [189–192]. Although these models represent a significant step forward, they usually generate single-function files that are often hard to compile and execute.

The second solution involves scraping code from a multitude of GitHub projects, thereby compiling a more extensive and diverse dataset. While this method raises ethical and legal considerations related to data usage and licensing, it has the potential to capture a broader range of optimization challenges encountered in real-world software development. However, the diversity of programming languages, build systems, and project structures on GitHub poses a significant obstacle. Automated code execution requires addressing issues such as dependency management, platform variations, and potential security risks associated with executing unknown code.

Feature Design and Model Selection. Designing a representation that captures relevant sets of features, and selecting an appropriate model represents an es-

sential step of the machine learning training process. Unlike images, where data is described with a 2D grid, or text where data is represented as a stream of tokens, code structure contains more information that should include not just algorithmic meaning but also its interaction with hardware. A meaningful representation of code involves both syntactic and semantic aspects and captures the intricacies of program flow, variable dependencies, and the underlying architecture. Furthermore, the representation should be sensitive to language-specific nuances and adapt to the diverse programming paradigms present in different codebases.

The substantial piece of previous work [112–115, 118, 120, 124] extracts a hand-written set of features for a given program, which is a challenging task. The process of manual feature extraction is not only time-consuming but also doesn't translate to diverse codebases, hindering its scalability and generalization across programming languages. Moreover, the narrow focus of hand-crafted features may overlook subtle yet crucial patterns and relationships within code that can significantly impact compiler optimizations.

Graph-based representation is a natural way to capture meaningful relations between variables and is heavily used for compiler analysis. These kinds of representations work well for the prediction of node and edge features but rarely can implement standard graph algorithms necessary for compilers. As graphs grow in size, the computational demands increase exponentially. Developing algorithms capable of handling large-scale graphs without compromising performance is a significant challenge.

Finally, with the development of LLMs, the text representations become relevant. Although this representation could encode all relevant information, reasoning about code semantics is significantly more challenging, which could impede the learning process. Overcoming these challenges is essential for harnessing the potential of machine

learning based compilers and this area is under active research.

Semantic Correctness. The Semantic Correctness problem presents a significant challenge when leveraging machine learning for code generation and optimization. Ensuring that the generated code not only compiles and runs but also adheres to the intended semantics and logic of the original code is a difficult task. machine learning models, especially those trained on diverse and extensive datasets, may prioritize syntactic correctness over semantic accuracy. This issue becomes particularly pronounced when dealing with subtle language-specific nuances, domain-specific constraints, or ensuring compatibility with specific libraries and frameworks.

There are two ways to address this problem: 1) Define a semantic-invariant set of actions that influence performance, and 2) Generate an extensive number of unit tests and validate the behavior of the generated code. LLVM and GCC made a step forward by introducing optimization flags, that are heavily used by machine learning based compilers. These principles could be lowered further by defining appropriate mathematical representations and a set of operations that describe semantic-invariant properties such as commutativity, associativity, and rearranging read instructions.

Programmers, similar to machine learning models, often introduce bugs, while writing and optimizing their code. To prevent major bugs, writing unit tests became a standard practice in software engineering, which ensures that code executes correctly most of the time. However, relying solely on extensive testing may introduce peculiar bugs and compromise software reliability. Achieving a balance between automated testing and formal verification methods is crucial for advancing the reliability of machine-generated code. As this interdisciplinary field evolves, the development of robust methodologies will be paramount to fostering trust in machine learning based code generators within the software engineering community.

Chapter 3

Optimizing Tensor Programs with Reinforcement Learning

3.1 Introduction

Contemporary advances in machine learning (ML) have led chip designers to develop extremely powerful chips to accelerate computationally intensive ML workloads. For instance, Nvidia introduced tensor cores [193, 194], Intel and AMD added Advanced Vector Extensions (AVX) [195, 196], Fused Multiply-Add (FMA) [197], and Vector Neural Network Instructions (VNNI) [198], while Google introduced Tensor Processing Units (TPUs) [199]. Moreover, hardware companies started making ML-specific chips such as Graphcore [200] and Cerebras [201].

Advanced compiler technology is necessary to fully harness the power of advanced hardware. However, traditional compilers have several limitations that impede their ability to do so.

First, traditional compilers have been developed for a limited set of Instruction Set Architectures (ISAs) with similar programming models, making it difficult to adapt them to exotic hardware with different chip resources. Even with the frontend/backend separation introduced by LLVM, the task remains challenging because traditional representations are not easily optimized for novel hardware. For example, IRs designed for conventional CPUs may not adequately represent the parallelism or

specialized instructions in GPUs or FPGAs.

Second, as traditional compilers are extended to cover more use cases, they become increasingly complex, with hundreds of optimization passes that frequently depend on one another. This complexity increases development and maintenance costs.

Finally, traditional "catch-all" compiler techniques fail to fully utilize novel resources on emerging hardware designed for specific workloads.

So, what are our alternatives to traditional compilers? Expert-optimized libraries, autotuners, or something else?

Expert-optimized libraries require experts to invest enormous amounts of time, and the work must be repeated for each new device. High-performance tensor operation libraries such as cuDNN [202], OneDNN [203], or XNNPACK [204] are usually tied to a narrow range of hardware devices and tend to be significant in code size, which may impede their use on mobile devices.

As an alternative approach, projects like Halide [205] and TVM [206] optimize a high-level representation of a loop nest performing a tensor computation with a discrete set of transformations such as loop reordering and tiling before compiling it to a particular target hardware with the LLVM compiler. This approach provides high performance and eliminates the need for expert-optimized libraries but introduces an astronomical number of possible loop nest configurations (schedules).

To optimize a simple problem, such as Local Laplacian Filters, Halide estimates a lower bound of 10^{720} possible schedules [205]. To find performant schedules in such a vast space, Halide and TVM use genetic algorithms and parallel simulated annealing with a trained cost model [206], respectively. Both approaches suffer from very large compilation times.

Contemporary breakthroughs in deep reinforcement learning (deep RL) in com-

plex video games, such as those in Atari [207] and AlphaGo [208], have inspired the compiler research communities to attempt to leverage deep RL [115, 159, 209, 210]. Similar to iterative algorithms, the deep RL agent explores an optimization space. However, there is one crucial difference - knowledge of the search space is embedded into a neural network. Inferring the neural network then replaces part of the search for optimizations. This example-driven, fast optimization-space search is precisely what ML-specific, as well as general compilers, need.

In our work, we further build on recent RL-based efforts in compiler research by developing LoopStack [4,5] – a novel RL-guided compiler toolset for optimizing tensor contractions that we explain in Section 3.3. Before going into details, let's formally define tensor contractions.

3.2 Tensor Contractions

The principal component of machine learning workloads can be expressed as a series of tensor contractions. Tensor contractions represent the generalization of matrix multiplication, trace, transpose, and other commonly used operations on matrices to higher dimensions.

We show an example of tensor contraction in Figure 3.1. Tensor contraction is a binary operation that consumes tensors, say tensor A and tensor B with dimensions (3, 4, 2) and (4, 2), respectively. We define a reduction dimension, marked with yellow boxes, which defines what elements of these tensors will be combined to produce a single scalar result. The reduction dimensions of both tensors have to be the same size. In this case, we select a dimension with size 2.

The computation between selected elements in tensors A and B consists of two steps. First, we apply the element-wise operation between each of the two components

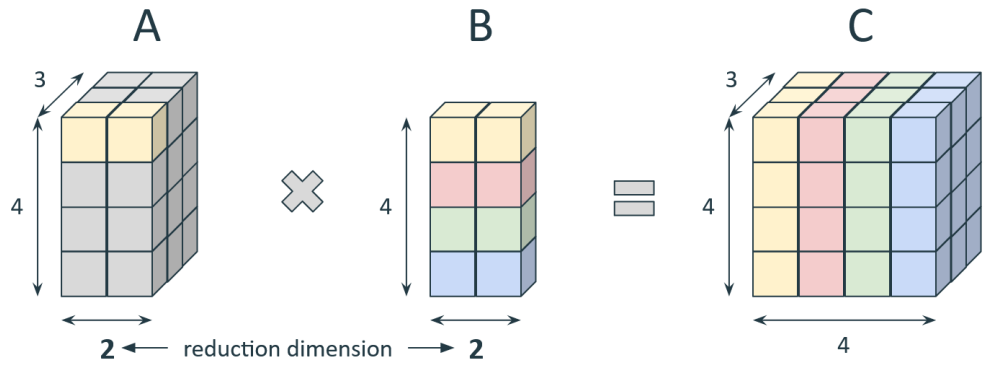


Figure 3.1 : Example of tensor contraction between tensors A and B. This computation consists of 1) element-wise operation between yellow boxes in tensors A and B and 2) reduction operation of the result of the previous operation to a single resulting box. The resulting tensor has a dimension of A and B without reduction dimension $\{4, 3, \not{2}, 4, \not{2}\} = \{4, 3, 4\}$. For simple matrix multiplication, element-wise operation is multiplication, and reduction operation is addition.

in the same relative position (one element from A and one from B). The resulting tensor will have the same shape as the selected elements from A or B. Second, we apply the reduction operation to sum the resulting tensor to a single scalar. This is shown as yellow boxes in tensor C. In the general case, we can select multiple reduction dimensions, which will be reduced to a single scalar of the resulting tensor.

Formally, we can define tensor contractions in the following way [211]. Let $\mathcal{A}, \mathcal{B}, \mathcal{C}$ be tensors with dimensions of d_A, d_B, d_C respectively. Similar to 2D matrix multiplication, for each pair of tensors $\mathcal{AB}, \mathcal{AC}, \mathcal{BC}$, we define the dimensions both tensors will iterate together. Namely, these indices will have dimensions $I_{AB} = (d_A + d_B - d_C)/2$, $I_{AC} = (d_A + d_C - d_B)/2$ and $I_{BC} = (d_B + d_C - d_A)/2$. Then, tensor contraction can be defined with:

$$\mathcal{C}_{\Pi_C(i_0..i_{I_{AC}},j_0..j_{I_{BC}})} = \sum_{k_0..k_{I_{AB}}} \mathcal{A}_{\Pi_A(i_0..i_{I_{AC}},k_0..k_{I_{AB}})} \cdot \mathcal{B}_{\Pi_B(j_0..j_{I_{BC}},k_0..k_{I_{AB}})}$$

where $\cdot$ is scalar multiplication and Π stands for all permutations of specified dimensions. Note that here, we have to do all permutations to keep the result consistent since the iterating dimensions may be chosen in any order. To simplify notation further, we can use Einstein notation and implicitly sum over dimensions that don't exist in the resulting tensor.

$$\mathcal{C}_{\Pi_C(I,J)} = \mathcal{A}_{\Pi_A(I,K)} \cdot \mathcal{B}_{\Pi_B(J,K)}$$

To allow the use of the non-linear activation function, used in deep learning, we extend our notation with an element-wise operation that transforms the final result (F).

$$\mathcal{C}_{\Pi_C(I,J)} = F(\mathcal{A}_{\Pi_A(I,K)} \cdot \mathcal{B}_{\Pi_B(J,K)})$$

With these extensions, we can express not only general matrix-to-matrix multiplication (GEMM), matrix-to-vector multiplication (GEMV), and vector-to-matrix multiplication (GEVM) operations, but also general machine learning primitives such as:

- Convolutions [68] : $O_{R,C} = I_{R+K,C+J} \cdot \omega_{K,J}$
- Pooling [68] : $O_{R,C} = max(I_{2R,2C})$
- Reductions [212] : $O_R = I_{R,C}$
- Transpositions [212] : $O_{R,C} = I_{C,R}$
- Concatenations [213] : $O_{R,C_1+C_2} = A_{R,C_1} | B_{R,C_2}$
- Broadcast [214] : $O_{R,C} = I_R$

Besides machine learning, tensor contractions are widely used in physics simulations, spectral element methods, quantum chemistry, and other fields. Despite many efforts [211, 215, 216], none of the state-of-the-art production compilers such as GCC [217], and LLVM [12] can automatically transform naive tensor contraction loop nests into expertly-tuned implementations.

3.2.1 Optimizing Generalized Tensor Contractions

The way we carry out these generalized tensor contractions has a significant impact on performance. For example, a naive 3-nested for-loop implementation of matrix multiplication of Figure 3.2a will result in long running times for non-trivially sized matrices. In contrast, efficient implementations, as proposed by many researchers over the past few decades [43,218–223], might modify the imperative loops to tile (i.e. split into sub-matrices of appropriate sizes) the input matrices to align with architecture-dependent resources such as caches, re-order loop dimensions to re-use data in the innermost loop, exploit architecture features such as vector instructions to operate on multiple elements at a time, and emit extended instructions such as fused-multiply-add, which combine multiplication and accumulation at the instruction level. Further, the data might be kept in exotic memory layouts, such as the channel–interleaved format often used on Intel machines [203, 224], or the matrix tile interleaved format proposed by Jia et al. [222, 223].

Typical scheduling operations include re-ordering, splitting, fusing, parallelizing, vectorizing, and unrolling loops [205, 225]. The scheduling options for a single task, such as matrix multiplication, highlight the combinatorial complexity of underlying scheduling. Different computations and platforms require varying schedules to deliver performance gains. This results in a challenging optimization problem. Historically,

```
for i:
    for j:
        C[i, j] = alpha * C[i, j];
        for k:
            C[i, j] += beta * A[i, k]
                * B[k, j]
```

(a) Pseudocode for basic imperative matrix-multiplication.

$$C_{i,j} = \alpha C_{i,j}^{in} \oplus \beta(A_{i,k} \otimes B_{k,j})$$

(b) Equivalent declarative Einstein notation for matrix multiplication, with $\oplus = +$ and $\otimes = *$.

Figure 3.2 : Matrix multiplication

varying techniques have been put forward to tackle this optimization, ranging from expert-based manual optimization, analysis-driven heuristic optimization [225, 226], to recent advances in data-driven automated schedule exploration [206, 227].

3.3 LoopStack

To optimize tensor contractions, we developed LoopStack, a domain-specific compiler stack that uses reinforcement learning to guide the optimization process. LoopStack is designed to produce highly efficient but also predictable code, allowing both experts and, more importantly, ML-based approaches to find performant loop schedules. Additionally, it is easily extensible and supports various processors and accelerators while incorporating HPC optimizations that are often missing from other machine learning compiler backends.

LoopStack consists of three parts:

- LoopNest [5] - an ultra-fast tensor compiler specialized for low-level optimizations for custom hardware.

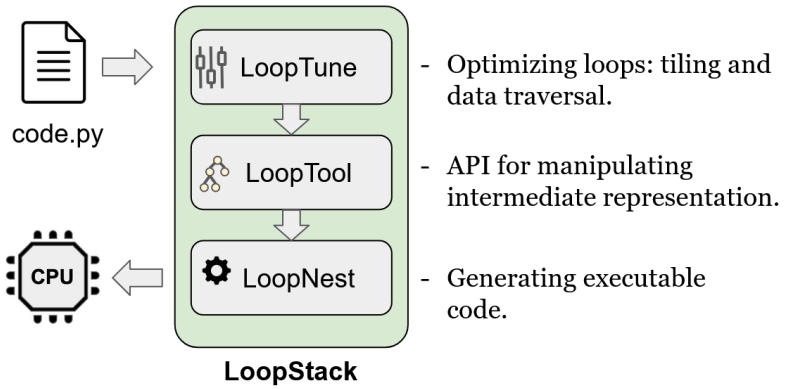


Figure 3.3 : LoopStack architecture [4].

- LoopTool [5] - interface that enables the user to define a specific loop nest order in which the computation should be performed.
- LoopTune [4] - reinforcement learning based framework for finding performant loop schedules optimized for LoopNest.

This strategy enables the quick development of customized compilers for novel hardware while leaving the complex problem of loop scheduling to LoopTune, which tailors loop schedules to a given backend utilizing its optimizations.

In the following chapters, we explain each component separately.

3.4 LoopNest – Backend Optimizer

LoopNest is a powerful, lightweight, domain-specific compiler designed to optimize deep learning workloads developed by Wasti et al. [5]. Rather than relying on extensive general-purpose compilers, it 1) utilizes a small set of expert-designed HPC optimizations and 2) enables the use of ML techniques for tuning in conjunction with it. LoopNest contains the following characteristics which are particularly important

for ML-based compilation research:

- Rapid compile times, a fraction of time compared to traditional compilers.
- Predictable and quick feedback.
- Generated machine code with performance comparable to or exceeding hand-optimized libraries such as MKL-DNN or XNNPACK.

A core challenge in high-performance tensor operations is identifying a good execution order — the schedule. There can be many valid schedules for a given computation: loops can be split and reordered, dimensions can be split and swapped, intermediate tensors can be packed, and so on. As a result, any scheduling tool faces the challenge of an exponentially large search space, making simple enumeration infeasible. Recent work [227–229] has explored using a combination of machine learning and structured search strategies to explore the search space automatically.

A key goal of LoopNest is to facilitate ML-based exploration of schedules. Effective exploration depends on rapid feedback, crucial to ML approaches, particularly when using reinforcement learning.

3.4.1 ML-centric Design

LoopNest is designed to generate optimized machine code efficiently that **strictly** adheres to the user's input. LoopNest exposes an API that allows the user to define a specific loop nest order in which the computation should be performed. LoopNest will **not attempt** to change the provided nest order or use any logic to recover user intent. In short, LoopNest will generate a machine code that performs the exact computation the user requested in the same order. This is crucial for ML-based autotuners, as the

hidden functionality in traditional compilers can make the learning problem extremely complex.

LoopNest applies low-level optimizations that are specific to the target hardware. It includes custom primitives in code generation, custom assembly codes, instruction reordering, reduction-sum, and other optimizations suggested by optimization manuals for the target hardware [230, 231].

Furthermore, LoopNest performs loop unrolling in a way consistent with hardware requirements and automatically vectorizes the innermost loop. It also applies register tiling [232], keeping a portion of the output tensor in registers at all times. To reduce pressure on load/store units, LoopNest never generates code that spills registers to the stack, unlike traditional compilers like LLVM and GCC. It achieves this by finding the largest scope in which modified values can fit in the register file.

Traditional compilers perform expensive analyses to understand user intent and optimize execution while performing equivalent computations. This can significantly increase the compilation times and obscure the intended schedule's impact on performance.

LoopNest takes a different approach. LoopNest does not attempt to understand the user's intent and/or reorder the intended schedule, except in very few hardware-specific cases, such as reordering loads and stores. Such optimizations highly depend on the limitations of the target hardware, don't require tuning, and are not beneficial to expose to the user.

This approach allows LoopNest to perform extremely fast compilation (quasi-linear in the number of issued instructions), which allows the user and/or ML agent to directly and rapidly evaluate the performance of their intended schedules. This approach greatly benefits autotuners [227,228] and reinforcement learning techniques,

which require both fast and accurate feedback on the quality of the schedule. Additionally, the vast amounts of data required for supervised-based methods have taken a long time and/or many resources to collect [229] with previous approaches. LoopNest provides orders of magnitude improvements in this domain.

3.4.2 LoopNest Optimizations

While traditional compilers typically perform multiple optimization passes, some of which might be repeated, LoopNest is designed to use only a limited number of well-studied optimizations to generate high-performance code. These include optimizations that are commonly used in expertly designed, custom assembly, or code generator primitives for specific problems, such as matrix multiplications [233–238] and other primitives used in machine learning [202, 203, 222, 223, 239–241]. Additional optimizations might include instruction reordering, reduction sum, and other optimizations suggested by optimization manuals for the target hardware [230, 231].

Vectorization

LoopNest assumes that the user intends to vectorize the innermost loop in the user-provided schedule. This design simplifies the logic while not introducing any limitations to the user – the vector operation elements are executed simultaneously and, thus, naturally belong to the innermost loop. LoopNest will, thus, attempt to vectorize the innermost loop. However, in certain scenarios, when the data accessed inside the innermost loop is not contiguous, and the target hardware doesn't support efficient gather operations, LoopNest will fall back to scalar operations.

No–Spilling of the Result Tensor

The concept of tiling (or "blocking") for the multiple levels of a cache hierarchy and the register file is a well-known optimization technique [43, 218, 220, 221]; referred to as *Cache Blocking Techniques* in the Intel's optimization manual [231]. LoopNest exposes these common HPC optimizations, where a subset of the output tensor is kept in the register file [203, 222, 223, 236, 237, 239, 241]. LoopNest never produces code that spills the content of the register file to the stack, an approach commonly used in traditional compilers, such as LLVM or GCC. Spilling the content of the register file to stack puts pressure on the hardware's load/store units, preventing full hardware utilization.

LoopNest does not decide on blocking or tiling sizes or the size used for the data in registers (register blocking). LoopNest instead identifies the outermost user-provided loop for which all compute can be performed with a subset of the output tensor kept in registers. Thus, it's up to the user to provide a well-chosen loop order and sizes, where the values kept in the register file can be reused many times. This approach gives the user greater control and more predictable performance than the case when spilling is allowed.

Reducing code size

Modern Intel processors can fetch and decode 16 bytes per cycle [231]. Thus, to maximize the number of instructions fetched and decoded, Intel's manual [231] suggests utilizing shorter encoded instructions, or instruction variants – keeping the combined size of every two adjacent instructions to 16 bytes or less. LoopNest achieves that in two ways. First, it prefers shorter instruction variants over longer ones, such as preferring arithmetic instructions where all arguments are in–register, and none are

in memory. Second, when an instruction has an in-memory argument, such as load instructions, LoopNest reduces the instruction sizes by using well-known SIB address encoding or small address offsets. [222, 223, 236, 237, 239].

The ARM (Aarch64) ISA has fixed instruction sizes; thus, reducing code size boils down to reducing the number of instructions. LoopNest achieves this by utilizing pre– and post– pointers for all memory accessing instructions, as described in the ARM's manual [230] and used by XNNPACK [204]. These instructions simultaneously update the value in the register holding the address.

Latency hiding

Finally, hiding the latency of memory accesses as well as instruction dependencies are another well-studied techniques [223, 236, 237, 239, 241]. LoopNest employs the two strategies described in [241]. First, to reduce exposed memory latency, it reorders loads so that they are seen by the instruction decoder as early as possible; and second, to reduce the instruction dependencies, when necessary, it introduces an additional set of register accumulators for the resulting tensor, and cycles through them. The introduced accumulators are then reduced at the end of the computation to produce the final result.

Single Operand LoopNest

LoopNest also provides functionality for generating efficient code for a simplified tensor contraction, where there are no reduction dimensions and only one input is provided. This functionality is typically used for *reshaping* a tensor (such as NumPy's **reshape** function) but can also be used for extracting a subset of a tensor into a smaller tensor or broadcasting elements along a tensor dimension. LoopStack requires

this functionality to allow the user's schedules to reorganize the memory for faster access [43, 218, 220, 221].

3.4.3 Evaluation

LoopNest's extremely low compilation times are the most important enabler for ML-based ML compilation. We perform a set of experiments to compare LoopNest's compilation time and execution time with state-of-the-art compilers often used for machine learning based autotuning.

Compilation time experiments

We compare LoopNest's compilation time to the LLVM compiler, a popular backend choice for prominent tensor autotuners such as Halide [205] and TVM [206]. We used 12 common operators found in machine learning workloads over varying input sizes for benchmarks. For LLVM code generation, we use Halide to emit schedules identical to the ones used with LoopNest.

Table 3.1 summarizes compile time across benchmarks. Our experiments show that LoopNest's code generation is faster than LLVM's compilation for all schedules, all workloads, and all target hardware. In most cases, LoopNest can generate code orders of magnitude faster. This is unsurprising because LoopNest generates code for a very specific kind of loop nest, while LLVM can generate any purpose code.

Runtime experiments

In Table 3.2, we show the average run–times of the top 5 fastest schedules found by our tuner. Our results suggest that LoopNest is generating code with comparable or better performance than the one generated by LLVM. We additionally compare

	x86 based CPUs						**Aarch64 (ARM) based CPUs (NEON)**											
	AMD (AVX2)			Intel (AVX512)			Cortex A57			NVIDIA Denver2			Cortex A73			Apple M1		
	LLVM	LN	Ratio	LLVM	LN	Ratio	LLVM	LN	Ratio	LLVM	LN	Ratio	LLVM	LN	Ratio	LLVM	LN	Ratio
CONV-1	820.78	**2.515**	**326.31**	9881.9	**6.206**	**1592.3**	2948.4	**9.211**	**320.1**	3972.4	**30.28**	**131.19**	4914.6	**22.077**	**222.61**	555.24	**24.763**	**22.422**
CONV-2	1590.1	**11.717**	**135.7**	9531.3	**5.803**	**1642.3**	2481.1	**8.947**	**277.29**	4798.5	**21.768**	**220.44**	4310.1	**31.053**	**138.8**	531.12	**15.261**	**34.802**
CONV-3	762.52	**5.932**	**128.53**	11061	**11.411**	**969.29**	3113.2	**17.749**	**175.4**	4184.3	**24.184**	**173.02**	3919.1	**25.51**	**153.63**	444.29	**20.435**	**21.742**
CONV-4	885.74	**41.163**	**21.518**	10706	**14.264**	**750.56**	4084.3	**66.102**	**61.788**	5169.9	**85.97**	**60.136**	6408.8	**99.952**	**64.119**	827.22	**35.605**	**23.233**
DWCONV-1	838.46	**0.294**	**2850.3**	10551	**0.280**	**37647**	2775.9	**3.801**	**730.28**	4108.2	**5.046**	**814.09**	4844.8	**2.805**	**1727.4**	571.46	**1.209**	**472.67**
DWCONV-2	1033.8	**0.472**	**2192**	11812	**0.679**	**17402**	2795.3	**2.524**	**1107.4**	3814.5	**5.204**	**733.06**	4160.9	**1.882**	**2210.3**	470.53	**0.524**	**897.79**
DWCONV-3	969.88	**0.300**	**3229.6**	13759	**1.460**	**9423.3**	2726.6	**2.158**	**1263.5**	3666.1	**4.233**	**866.15**	3320	**3.822**	**868.74**	506.63	**0.663**	**764.87**
DWCONV-4	1000.7	**0.334**	**2993.4**	10704	**0.813**	**13159**	3474	**5.511**	**630.34**	4577.5	**9.477**	**483.01**	4197.4	**2.877**	**1459.1**	449.89	**1.6**	**281.18**
MM-64	697.37	**0.384**	**1813.6**	9099.5	**0.853**	**10667**	3432.4	**7.359**	**466.43**	4779.2	**13.103**	**364.75**	4417.6	**9.171**	**481.7**	397.59	**1.327**	**299.68**
MM-128	925.47	**1.578**	**586.47**	9044.8	**0.795**	**11379**	4991.9	**11.153**	**447.57**	5754.2	**16.025**	**359.07**	6681.1	**13.956**	**478.72**	387.82	**1.119**	**346.63**
MM-256	1118.5	**2.692**	**415.41**	18119	**3.020**	**5999.4**	5003.1	**18.511**	**270.28**	5770.9	**26.485**	**217.9**	6800.6	**27.446**	**247.78**	390.29	**5.165**	**75.57**
MM-512	1262.3	**4.340**	**290.83**	12336	**4.485**	**2750.8**	4814.2	**7.485**	**643.17**	5698.9	**12.25**	**465.22**	6471.7	**14.545**	**444.94**	1084.4	**9.512**	**114**

Table 3.1 : Compile times, in milliseconds, for `LLVM` and LoopNest. LoopNest performs the compilation orders of magnitude faster [5].

the fastest schedules found by our tuner to the extremely efficient, hand-optimized ones. In nearly all cases, LoopNest matched or exceeded the performances of the hand-optimized ones.

3.5 LoopTool API

To enable users to effortlessly describe the desired computation of neural network workloads using LoopStack, we built a domain-specific language (DSL) embedded in Python and a compiler frontend named LoopTool. LoopTool exposes a declarative API for user computation definition and operates on an IR composed of an annotated data-flow graph (ADFG) consisting of N-dimensional tensor operations. The ADFG describes the underlying computation, memory layouts, and execution. LoopTool then lowers the ADFG into a series of loops, which are then compiled with LoopNest.

We use an Einstein-like notation [242] as it enables the user to quickly describe

	x86 based CPUs						**Aarch64 (ARM) based CPUs (NEON)**											
	AMD (AVX2)			Intel (AVX512)			Cortex A57			NVIDIA Denver2			Cortex A73			Apple M1		
	LLVM	LN	Ratio	LLVM	LN	Ratio	LLVM	LN	Ratio	LLVM	LN	Ratio	LLVM	LN	Ratio	LLVM	LN	Ratio
CONV-1	86.767	**87.638**	**1.01**	72.943	**162.48**	**2.227**	11.021	**11.126**	**1.009**	10.968	**14.782**	**1.348**	**9.847**	9.399	0.955	**98.121**	97.51	0.994
CONV-2	45.765	**71.482**	**1.562**	13.813	**156.78**	**11.35**	**11.276**	11.179	0.991	13.407	**14.511**	**1.082**	**9.810**	9.247	0.942	**95.743**	92.027	0.961
CONV-3	9.495	**72.433**	**7.629**	96.448	**184.4**	**1.912**	8.181	**10.183**	**1.245**	6.407	**13.27**	**2.071**	6.467	**7.998**	**1.237**	53.681	**83.252**	**1.551**
CONV-4	3.307	**90.722**	**27.434**	123.73	**181.72**	**1.469**	5.325	**9.569**	**1.797**	4.361	**12.532**	**2.874**	2.951	**7.686**	**2.604**	46.806	**77.952**	**1.665**
DWCONV-1	48.695	**62.541**	**1.284**	48	**57.592**	**1.200**	**6.352**	6.234	0.982	10.243	**11.858**	**1.158**	3.999	**4.502**	**1.126**	**40.133**	39.01	0.972
DWCONV-2	39.203	**53.465**	**1.364**	28.302	**37.671**	**1.331**	5.120	**5.703**	**1.114**	7.107	**7.969**	**1.121**	2.518	**2.773**	**1.101**	42.865	**43.438**	**1.013**
DWCONV-3	62.873	**84.848**	**1.349**	57.544	**88.094**	**1.531**	6.934	**7.721**	**1.113**	10.65	**12.551**	**1.178**	5.764	**6.457**	**1.120**	70.796	**76.261**	**1.077**
DWCONV-4	77.071	**84.21**	**1.093**	125.04	**159.38**	**1.275**	9.687	**10.174**	**1.050**	12.665	**14.717**	**1.162**	7.607	**8.017**	**1.054**	**81.588**	80.901	0.992
MM-64	85.808	**102.2**	**1.191**	144.67	**187.65**	**1.297**	12.751	**13.441**	**1.054**	14.039	**15.754**	**1.122**	9.523	**10.166**	**1.067**	91.18	**199.81**	**2.191**
MM-128	92.692	**102.5**	**1.106**	168.84	**185.19**	**1.097**	12.417	**12.496**	**1.006**	14.253	**15.291**	**1.073**	9.246	**9.506**	**1.028**	91.763	**98.4**	**1.072**
MM-256	92.862	**100.21**	**1.079**	170.18	**182.46**	**1.072**	**11.428**	11.401	0.998	14.241	**14.645**	**1.028**	**8.686**	8.610	0.991	95.597	**99.902**	**1.045**
MM-512	90.189	**98.199**	**1.089**	**160.42**	159.59	0.995	6.997	**8.327**	**1.19**	**14.395**	13.894	0.965	6.336	**6.890**	**1.087**	89.618	**97.65**	**1.090**

Table 3.2 : The average execution performance of top 5 schedules, in GFLOPS, for code generated by `LLVM` and LoopNest. LoopNest achieves comparable (within measurement error) or superior performances while taking a fraction of the time to generate the code [5].

the intended mathematical operations even for complex machine learning models. We further provide a minimal yet powerful intermediate representation (IR) and limit the API such that all relevant optimization operations can be decomposed into a series of operations on individual nodes in the ADFG. Despite resembling Halide's pipeline scheduling states [227], it is impossible to represent illegal schedules in our IR. This is particularly important for ML-based approaches, such as reinforcement learning.

3.5.1 Declarative API

Figure 3.4 shows an example of matrix multiplication in LoopTool's declarative Python DSL. The expression lt.Var("m") defines an indexing variable with the corresponding name. Next, the expression lt.Tensor([M, K]) defines a 2-dimensional

```
import loop_tool as lt
M, N, K = lt.Var("m"), lt.Var("n"), lt.Var("k")

A = lt.Tensor([M, K])
B = lt.Tensor([K])
C = lt.Tensor()

C[m, n] += A[m, k] * B[k]
```

Figure 3.4 : LoopTool's Python embedded declarative DSL [5].

tensor. LoopTool uses symbolic (i.e., named) dimensions [243], simplifying indexing semantics and encouraging a simple interaction model for manipulating traversal and memory layouts of higher dimensional tensors. The expression $C[m, n] += A[m, k] * B[k]$ defines computation using the aforementioned Einstein notation; here k is a reduction dimension. LoopTool's computation language supports element-wise computations (with broadcast semantics), associative reductions across arbitrary dimensions, and a restricted set of indexing semantics.

3.5.2 Intermediate Representation (IR)

LoopTool's ADFG is based on an intermediate representation with annotations on each node. Each node is associated with its output – a virtual buffer materialized according to the user-provided schedule. Figure 3.5 (left) demonstrates a matrix multiplication without annotations. A node's output size is not materialized until the IR is lowered to loops. The materialization logic always attempts to minimize the total memory used. For example, two nodes operating on a virtual buffer of size N in a shared loop over N do not need to allocate the full N memory elements for

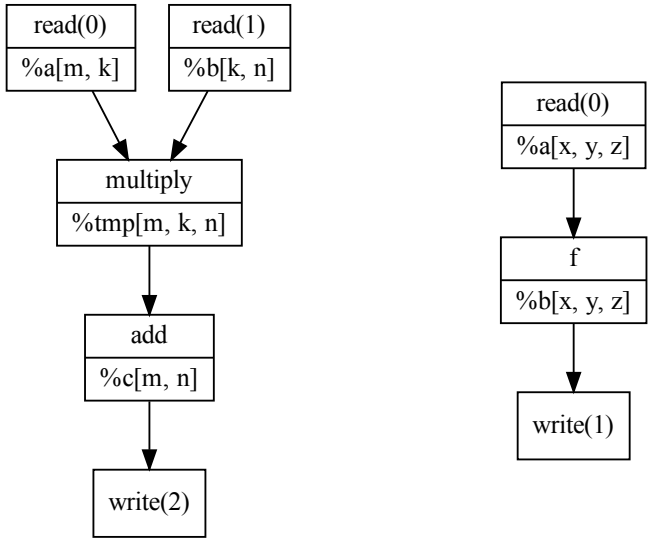


Figure 3.5 : Matrix multiplication in LoopTool (left). A point-wise application of the function f across all three dimensions of `%a` (right) [5].

their intermediate. Instead, the intermediate will be size one and reused N times.

LoopTool's ADFG has three fundamental types of nodes: 1) Read/write nodes, 2) Arithmetic nodes, and 3) View nodes.

Read/write nodes denote reads or writes from and into user-provided memory, as well as associated layouts and sizes. These nodes contain an ordered list of symbolic dimensions. The ordered list of dimensions represents a row-major order (lexicographical order) of either input or output memory. Because LoopStack is embedded in either Python or C++ applications, these nodes are used as interfaces with other operations that LoopStack does not handle. An example would be the specification of the input convolution layouts NCHW [70] or NHWC [71], where N is the depth of the network, C is number of channels, H is the chanal height, and

W is the chanal width. Both of these are well handled and trivially manipulated in the LoopTool IR. In the IR, read nodes have no inputs (predecessors), whereas write nodes have a single predecessor and no output (successors). These are effectively source and sink operations in the graph.

Arithmetic nodes operate on one or multiple input virtual buffers and output a single virtual buffer. The ordered list of dimensions associated with these nodes denotes how output memory should be laid out (given the corresponding scope of their execution). Arithmetic nodes, unlike read and write nodes, can take multiple inputs. This represents the typical operation applied to inputs. For example, adding two inputs works as expected to yield a single output.

Extending this concept to higher dimensions forces us to consider when two inputs do not have the same dimensions. Due to the symbolic nature of the dimensions in LoopTool, we can distinguish two dimensions of the same size as being mathematically distinct. To handle the application of arithmetic in higher dimensions, we employ implicit broadcasting akin to Numpy [244] semantics. Dimensions not present in the output are implicitly reduced according to the arithmetic of the associated node.

View nodes represent symbolic indexing constraints over tensor dimensions, allowing robust view semantics. Any affine combination of iteration over the output dimensions can be indexed into an input dimension. This enables us to represent windowed operations as well as concatenations. By using index constraints rather than index equations, we preserve the meaning of the underlying computation and can freely split and permute variables and layouts. Further, we can still apply scope-based memory minimization logic by keeping indexing math symbolic.

3.5.3 Lowering to loops

LoopTool lowers its IR to an internal loop tree structure before invoking LoopNest. This is done by traversing the ADFG topologically and eagerly emitting loops required for each node. A reference to the current innermost loop is maintained throughout this process, with each subsequent loop nesting inside the reference. If any node requires a loop that is an ancestor to the current reference, there is no need to emit the loop again, and it is skipped. This naturally induces loop coalescing (typically referred to as loop fusion).

Consider nodes shown in Figure 3.5 (right). We can assume an annotation of `%a` and `%b`, both with loop order `[x, y, z]`. When visiting `%a`'s node, we emit the loop tree shown in Listing 1.

Later, while visiting `%b`'s node, we note that the reference (which is at location `%a`) contains a loop for `x`, `y`, and `z` in order. We thus reuse all loops and implicitly "fuse" node `%b`. This is shown in Listing 2

We find the nodes can be executed in the same innermost loop. In this case, the resultant allocation size of `%a` would be 1.

Listing 1 Lowering `%a` emits three new loops [5].

```
iter x:
  iter y:
    iter z:
      %a[x, y, z] = read(x, y, z)
      ...
```

Listing 2 Lowering `%b` reuses all loops [5].

```
iter x:
  iter y:
    iter z:
      %a[x, y, z] = read(x, y, z)
      %b[x, y, z] = f(%a[x, y, z])
```

However, if the nodes had different loop annotations, such as `%b` annotated with `[x, z, y]`, we would not be able to share every loop. Schedule in Listing 3 shows code that only needs to allocate $|y| \cdot |z|$ elements in tensor a.

Listing 3 Only loop `x` is shared across the two nodes [5].

```
iter x:
  iter y:
    iter z:
      %a[x, y, z] = read(x, y, z)
  iter z:
    iter y:
      %b[x, y, z] = f(%a[x, y, z])
```

In the case of reductions, we cannot always share loops. Consider a reduction node `%R` over variable `z` with loop order `[x, y, z]` depended on by `%a` with the same loop order; see Listing 4. The loop for `z` must run twice for correctness. While necessary for reductions, this type of behavior may also be preferable in other contexts.

Listing 4 Sharing loop `z` between `%R` and `%a` would be mathematically incorrect [5].

```
iter x:
  iter y:
    iter z:
      %R[x, y] = reduction(...)
    iter z:
      %a[x, y, z] = %R[x, y] + ...
```

Manually preventing loop fusion can induce larger intermediate memory allocations. This is often beneficial when computation benefits from packing memory into a cache-friendly layout before computation [245]. Furthermore, reusing storage may reduce parallelism [246]. To express this, LoopTool has a second form of annotation for nodes called *staging* that prevents the reuse of specific loops. In Listing 2, `%a` and `%b` share an entire loop nest. If we were to *stage* the loop for `%a` over `z`, the resultant lowering would increase the allocation size of `%a` to $|z|$ (Listing 5).

Listing 5 `z` is staged, so `%a` is materialized with an allocation of size $|z|$ [5].

```
iter x:
  iter y:
    iter z:
      %a[x, y, z] = read(x, y, z)
    iter z:
      %b[x, y, z] = f(%a[x, y, z])
```

Generalizing to computations with multiple loop nests

Some computations or their schedules can result in a sequence of multiple nested loops that may share a set of outer loops, effectively forming a tree of loops. We developed a loop tree interface to generalize our approach to these workloads. The loop tree interface provides a simple API to build up a tree, where inner nodes correspond to for-loops, and leaves correspond to an innermost computation over tensors or a transposition of tensors. LoopNest then compiles all independent nests and executes the tree. The final result is a function that can be called with the appropriate input, intermediate, and output tensors to realize the tree-defined computation.

LoopTool Tuning Interface

To explore schedule space, LoopTool defines a tuning interface that enables an expert or tuning script to manipulate loop nest. From a given computation, we can derive various valid schedules by splitting and reordering loops, swapping and splitting dimensions, and packing intermediate tensors. Starting from a tensor computation defined by DSL (Figure 3.6), LoopTune constructs an intermediate representation from which we can begin schedule exploration.

The loop nest representation consists of a computation nest and a write-back nest. Computation nest contains a series of loops that could have annotations such as *unroll* or *vectorize* that direct LoopNest implementation. In the loop body of the inner loop, we define the element-wise operation between two tensors – *multiply*, and the reduction operation – *add*. Element-wise the operation takes tensors with a defined access pattern, resulting in a temporary tensor with dimension equal to the union of tensor %0 and tensor %1 dimensions. The reduction operation sums the temporary tensor's dimension k. Finally, the write-back nest defines how the temporary tensor is

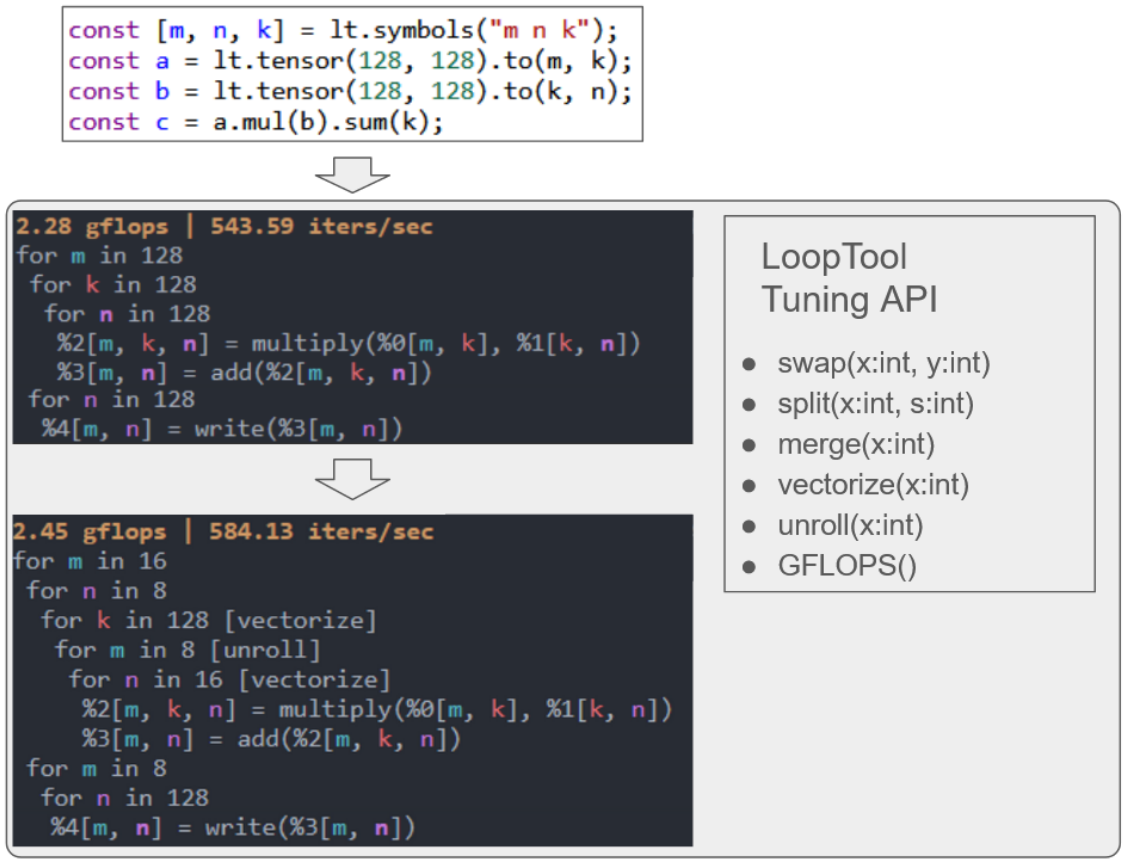


Figure 3.6 : Tuning loop schedule with LoopTool. Try it on https://loop-tool.glitch.me.

written back to memory. With this representation, LoopNest gets information about the memory access pattern, reduction dimension, and write-back pattern, uniquely identifying execution order.

Method *swap* enables LoopTool to swap any two loops with different iterators in the same loop nest. Method *split* divides the loop for a given factor and creates a new loop with the same iterator. Method *merge* revert split operation and merge given loop with the parent loop with the same iterator. Methods *vector* and *unroll* annotate a given loop that will be optimized with LoopNest. Finally, method *GFLOPS* eval-

uates the performance of the given loop nest. These methods allow us to manually explore and assess schedule space or incorporate them in searching scripts.

Since the number of possible schedules grows exponentially to loop nest depth, deriving an efficient exploration algorithm is paramount. In the next section, we explain how we use reinforcement learning to navigate optimization space efficiently.

3.6 LoopTune - Frontend Tuner

LoopTune is a reinforcement-learning framework for finding performant loop schedules. LoopTune manipulates loop schedule space through LoopTool's Tuning API and generates the binary from the given schedule with LoopNest. LoopTune aims to train the policy network to find a close-to-optimal schedule for the given loop nest in a few steps, decreasing auto-tuning time to the order of seconds.

This section introduces a novel LoopTune action space suitable for RL training and graph-based embedding of tensor computations and evaluates five popular RL algorithms in optimizing loop schedules. By combining reinforcement learning with appropriate representations and a well-chosen optimizer, we can generate faster code than baseline traditional search techniques, outperform popular autotuners such as autoTVM and MetaSchedule, and perform at the level of an expert-optimized library Numpy.

3.6.1 Learning to Optimize Tensor Computations

To optimize tensor operations, we separate the problem of finding optimal loop range and order (schedule) from hardware-dependent low-level optimizations, such as vectorization. To find performant schedules, LoopTune uses deep reinforcement learning to train a policy network, while LoopNest [5] applies low-level tensor optimizations

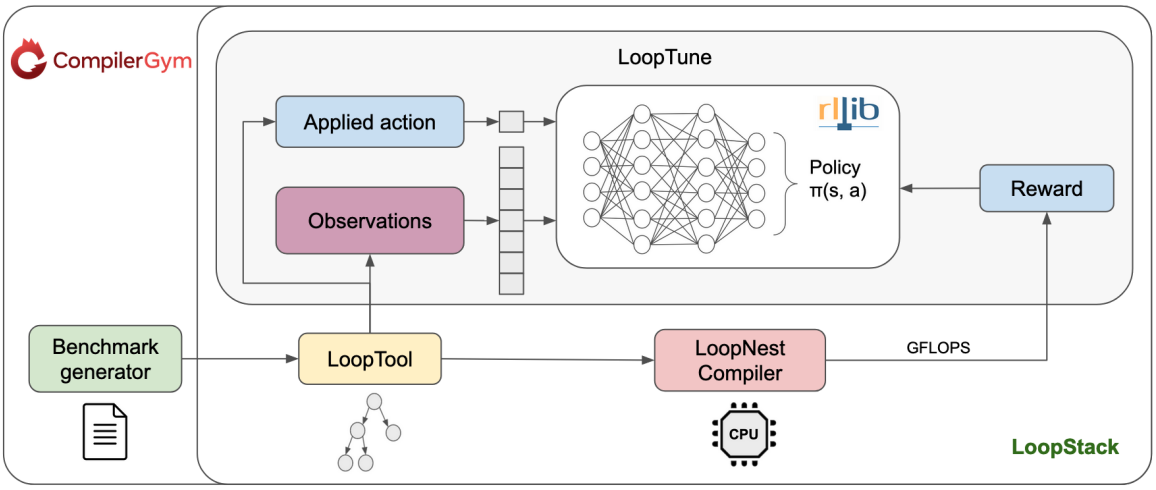


Figure 3.7 : LoopTune training loop. LoopTune transforms the generated benchmark to an intermediate representation (IR) and uses LoopTool API to apply actions and get observations, while LoopNest compiles and executes the loop nest, providing the reward [4].

and generates executable code given a schedule.

The process begins by creating an RL environment by using CompilerGym [247] (Figure 3.7). This framework allows us to map the problem of finding performant schedules to RL methodology and using state-of-the-art libraries such as RLlib [248] for training. To create an environment in CompilerGym, we define an action space, an observation space, and a reward that will be used as an optimization criterion during RL training. Additionally, we define a set of benchmarks that represent tensor operations. For all benchmarks, we assume that loop bounds are constant.

In each training epoch, we convert the benchmark to an intermediate representation by adding an "agent" annotation to the first loop (Figure 3.8). In each step, the agent applies an action from the action space to the current loop, changing the loop

schedule. LoopTune encodes our novel graph-based state to a vector representation (described in Section 3.6.4) and feeds it to the reinforcement learning training loop. Finally, LoopNest compiles and evaluates the loop schedule, which provides a reward signal for training the policy network.

In the inference phase, LoopTune iteratively calculates the policy network's best action and applies it to the current state. Since this procedure doesn't include loop nest evaluation, it is fast and constrained only by the speed of the inference. Practically, this enables the policy network to reach the desired state quickly in a matter of seconds.

3.6.2 Defining an Action Space

The LoopTool Tuning API allows LoopTune to swap the positions of two loops, given their line numbers, and split a loop, given its line number and a specified tile size. Rather than having such parametric actions that are inherently hard to train [249], LoopTune defines a novel action space shown in Figure 3.8 and introduces the abstraction of an agent that traverses loop nests and applies actions on each loop.

The agent uses *up* and *down* actions to move the cursor without changing the loop nest structure. The *swap_up* and *swap_down* actions direct the agent to exchange the position of the current loop with its neighbor, moving the agent's cursor, respectively. The *split* family of actions creates a new loop with the same iterator, dividing the loop range with the specified split parameter. If the split parameter does not evenly divide the loop range, the current loop will have a remainder or "tail", which will be executed at the end of the loop nest execution. We limit the size of the split parameter to 4, 8, and 16.

Since we always start from the basic implementation of loop nest, we can reach

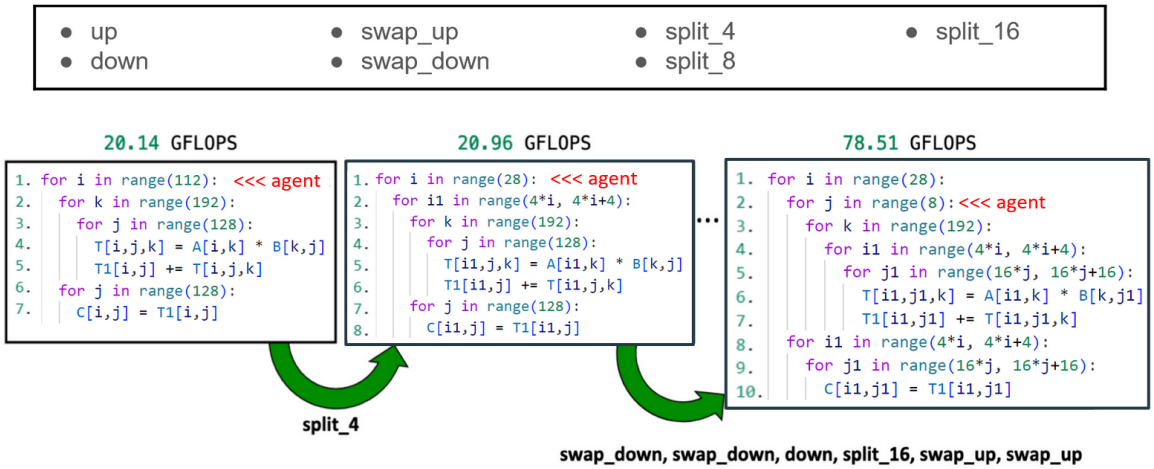


Figure 3.8 : Optimizing ranges and order of loops for matrix multiplication using LoopTune's action space [4].

any configuration without ever using the *merge* method. Similarly, instead of using *unroll* and *vectorize*, we expect LoopNest to vectorize the most-inner loop and unroll the rest automatically.

Limiting the action space in this way simplifies the problem in several ways. For example, a smaller number of possible actions enables the RL algorithm to explore and become more confident [249] with each action for different states. This might force the agent to use longer sequences of actions to reach certain states, but this is not a problem since each action other than *up* and *down* changes the loop nest and provides a non-zero reward signal. Furthermore, many states benefit from similar action sequences, which allows training to converge faster.

To simplify the design, we decided to apply a fixed number of actions for optimization rather than having an action that terminates the search. Our experiments have shown that having such an action often prevents exploration and converges to a local minimum.

3.6.3 Defining a Reward

For the evaluation metric, we use billions of floating-point operations per second (GFLOPS). To measure GFLOPS, LoopTune uses LoopNest to compile and execute loops on a CPU. To ensure reliable results, LoopNest excludes the first 20 iterations as a warm-up and times multiple executions of the loop nest, taking the fastest measurement.

During training, the agent applies action (A) from state S, moving it to the next state S'. The feature extractor maps the internal representation to the vector (S) used as input to the neural network. LoopNest calculates the reward for the applied action using the formula:

$$reward = \frac{GFLOPS(S') - GFLOPS(S)}{GFLOPS\_PEAK\_PERFORMANCE}$$

This normalizes all rewards, making training more stable. Rather than relying on peak performance from hardware specifications that may be imprecise, we evaluate peak performance empirically before the training by running the series of kernels with high arithmetic intensity, which always falls within a few percent of the theoretical peak. Finally, we send a tuple (S, S', A, R) to the RL library that performs one training step.

3.6.4 Defining the State Representation

Each loop nest consists of a nest that computes operations and a write-back nest that writes the result to the memory. We use the graph-based representation shown in Figure 3.9 for state representation. On this graph, there are three kinds of nodes: loops (rectangles), data (ellipses), and computation (diamonds). There are three kinds of edges. Black edges connect loops and computations that are nested from top

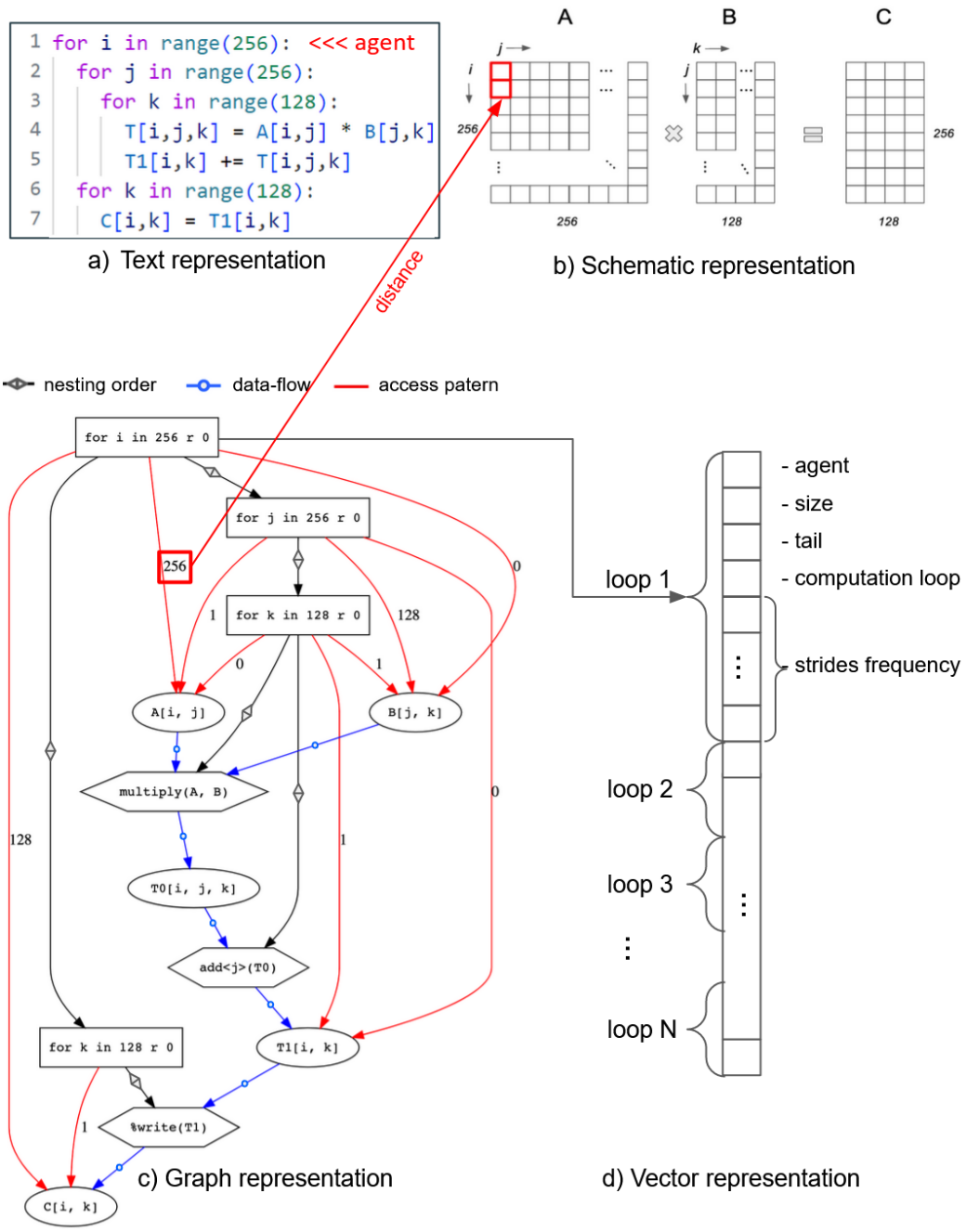


Figure 3.9 : Text representation shows the algorithm. Schematic representation shows the memory layout. Graph representation explains nesting order (black), access pattern (red), and data flow (blue). Vector representation aggregates graph representation for the training [4].

to bottom. Blue edges represent data flow, while red edges represent the strides of each loop accessing tensors that read from memory (A, B) or write to memory (T).

Stride is the distance in memory between two tensor elements when we increment only the index of a given loop. If this number is large, the iterating loop will try to fetch distant data in memory that may not be stored in the cache, resulting in a cache miss.

We map the key features in a vector to make our representation usable for standard RL optimizers. In our vector representation, each loop is described with 20 integer values, namely:

- (1) Is the agent's cursor on the loop
- (1) Loop size
- (1) Loop tail
- (1) Does loop belong to computation or write-back nest
- (16) Histogram of strides frequency

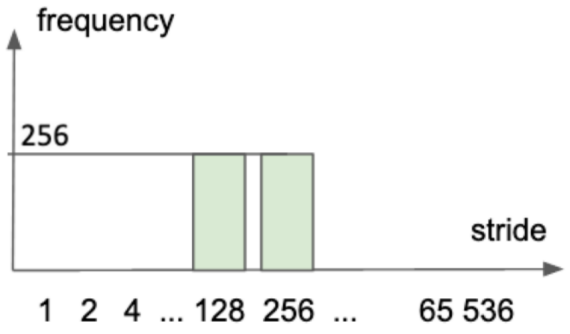


Figure 3.10 : Histogram of strides frequency [4].

The histogram of strides frequency (Figure 3.10) represents each loop's cumulative distribution of access strides. In other words, it shows how many accesses with given strides are produced from the given loop. We calculate strides from the tensor shape and iterator position for each loop. Since stride can be an arbitrary integer larger than zero, we discretize strides to bins of size 2^N, where $N \in \{0...15\}$ to match the sizes of cache lines.

Agent bits are necessary since they give meaning to all actions since they depend on the cursor position. Size and tail bits define how many times memory is accessed with each stride, which distribution is captured in strides frequency. The computation loop shows whether the loop is used for computation or write-back.

This is a minimal set of features for the RL algorithm to learn memory access patterns, which is critical to optimizing memory-bound computations such as tensor contractions. Adding features that describe computation in the loop body would be beneficial for compute-bound applications.

3.6.5 RLlib - Library for Reinforcement Learning

In our work, we evaluate several learning algorithms, supported by RLlib [248], including Deep Q Learning (DQN), Apex Deep Q Learning (APEX_DQN), Proximate Policy Optimization (PPO), Actor-Critic (A3C), and Impala.

DQN [250] attempts to learn the state value function by using experience replay for storing the episode steps in memory for off-policy learning, where samples are drawn from the replay memory at random.

APEX_DQN [251] creates instances of environment for each actor and collects the resulting experience in a shared experience replay memory, prioritizing the most significant data generated by actors.

PPO [252] alternates between sampling data through the interaction with the environment while using stochastic gradient ascent with minibatch updates.

A3C [253] calculates gradients on the workers directly in each episode and only broadcasts these gradients to the central model. Once the central model is updated, parameters are sent back to the workers.

IMPALA [254] provides a scalable solution for collecting samples from individual agents and running stochastic gradient descent in the central loop.

3.7 Search to Optimize Tensor Programs

We implement a set of traditional search algorithms to set a baseline for our comparison. Traditional approaches for auto-tuning tensor programs are based on hill climbing [92, 114], genetic [96–98], and various search algorithms [255]. These algorithms can find performant schedules for a single program, but the search time and the quality of the solution depend heavily on the smoothness of the optimization space. If the optimization sequence to highly rewarded states includes some actions that produce negative rewards, hill climbing algorithms can converge to local minima. Genetic algorithms, on the other hand, use many computational resources since they converge slowly.

We implemented the following set of algorithms (Figure 3.6):

- Greedy search with lookahead of 1 and 2
- Beam Depth First Search (BeamDFS) with width 2, 4
- Beam Breath First Search (BeamBFS) with width 2, 4
- Random search

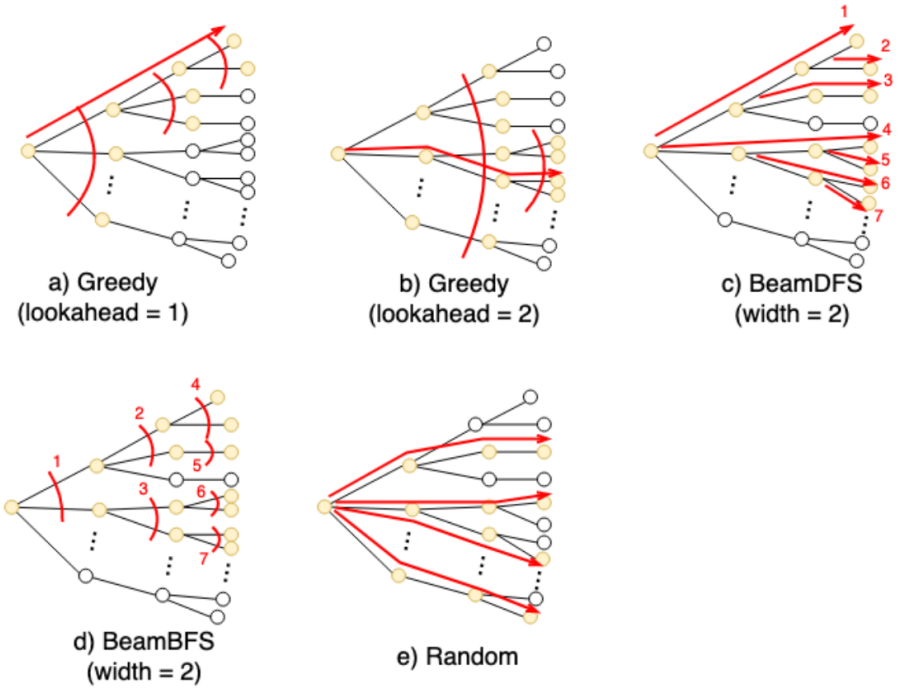


Figure 3.6 : Traditional search approach in finding the optimal sequence. Actions (edges) are sorted by the performance of the next state [4].

First, we introduce the family of Greedy search algorithms with arbitrary lookahead. In each step of this algorithm, we evaluate all possible states after applying lookahead steps and select the step toward the most promising state. With a lookahead of 1, the agent stops if there is no better action than the current state. In contrast, the lookahead of 2 enables the agent to tolerate one detrimental step that leads to a more promising solution. Ideally, with a large enough lookahead, Greedy Search could overcome the problem of local minima for actions with negative rewards. Unfortunately, such computation comes with the cost of $O(steps * |action\_space|^{lookahead})$, which is prohibitively expensive for large lookaheads.

Second, we implemented a family of Beam search algorithms with arbitrary width.

In each step, we calculate the best width actions and expand them further until we reach the specified depth of the search tree. Expansion of the states could be done in depth-first (BeamDFS) and breadth-first (BeamBFS) manner, and search properties drastically differ when search time elapses before the full search graph is constructed. Complexity of both of these approach is $O(width^{steps})$, where $width < |action\_space|$.

BeamDFS can be seen as an extension to the Greedy algorithm with a lookahead of 1, with few additions. It doesn't terminate if the next state is worse than the current, and it recursively visits all states of the search graph where each node has maximum *width* children. This enables it to tolerate non-convex parts of spaces as long as the optimal action ranks better than other actions in the current step.

The BeamBFS variant finds a performant action sequence iteratively as it builds a search graph for each number of steps. This approach would be beneficial if the performant sequence is shorter than the specified search depth.

Finally, Random Search randomly chooses a sequence of actions with a specified length. The benefit of this search is that it can uniformly explore many diverse states, providing a general idea of the landscape. From our experiments, random search provides surprisingly good results that we elaborate on in the next section.

3.8 Experiments

We evaluated LoopTune on a series of benchmarks to answer the following questions:

- How do different RL algorithms compare to each other?
- How does LoopTune compare to traditional search algorithms?
- How does LoopTune compare to optimized libraries and autotuners like TVM?

The benchmark dataset consists of synthesized loop nests for matrix multiplication. The matrix multiplication dataset has 2197 untiled loop nests for matrices with dimensions in the range from 64 to 256 with the step of 16. We train on the 80% split of the dataset (size 1757) while leaving 20% for the test dataset (size 440).

Experiments are performed on an Intel Xeon CPU running on 2.20GHz, with a peak performance of 114.204 GFLOPS, 40 CPU cores, and 2 Nvidia Quadro GP100 GPUs. The CPU has cache sizes L1 (data/instruction) 1.3 MB, L2 10MB, L3 52MB.

3.8.1 RLlib Training Analysis

We describe our problem as a CompilerGym environment with action space, representation, and reward function and use RLlib [248] library to train the policy network. We compare PPO, A3C, DQN, APEX_DQN, and Impala to find the best training algorithm. In all cases, we use a network with fully connected layers, arbitrary width, number of layers, and the same feature representation.

To find the optimal parameters for each training algorithm, we run a hyperparameter sweep for the learning rate, exploration factor, depth, and width of the neural network. After finding the best parameters for each training algorithm, we run the final training for 4000 iterations and stop training early if the average reward per epoch converges to some constant. The optimizer applies ten actions in each epoch and updates the neural network with a reward signal. Finally, we compare training algorithms by plotting the *episode_reward_mean*, which represents the averaged increase of GFLOPS achieved in the episode normalized to the peak performance of the device (Figure 3.7).

We found that the APEX_DQN training algorithm performs an order of magnitude better than other training algorithms, converging after roughly 200 steps and

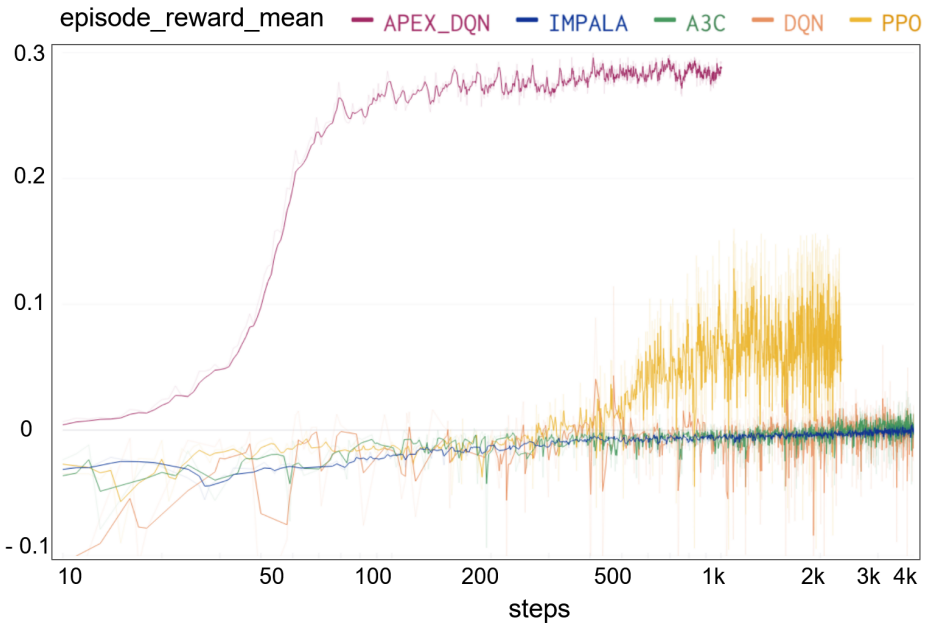


Figure 3.7 : Average reward per epoch for RLlib algorithms during training of 4000 steps [4].

providing an average increase of 30% of the peak performance. In contrast, PPO required more than 1000 steps to converge to an improvement of 8% of the peak, while Impala, A3C, and DQN have not been able to achieve positive results. We trained APEX_DQN for 17.5 hours with the following winning configuration: lr = 1e6, gamma = 0.95, network_depth = 10, network_width = 1000.

We believe that the superiority of APEX_DQN lies in the capability to prioritize the most significant experiences generated by the actors. Since all the algorithms are implemented in RLlib, it takes only one line of change to benefit from novel algorithms in the future. We further compare the APEX_DQN solution with non-RL approaches.

3.8.2 Comparison to Search Based Approaches

To evaluate the difficulty of searching the optimization space, we run a set of traditional search algorithms, including Greedy search with lookahead of 1 and 2, BFS and DFS variants of Beam Search with widths 2 and 4, and Random Search. We implemented each search with caching to avoid repeating evaluations of the same states. We run each search on a test dataset of 440 benchmarks, setting the time limit to 60 seconds. To compare traditional searches to policy generated from the RL approach, we plot the search time and achieved performance of the generated code of 25 random benchmarks from the test set in Figure 3.8.

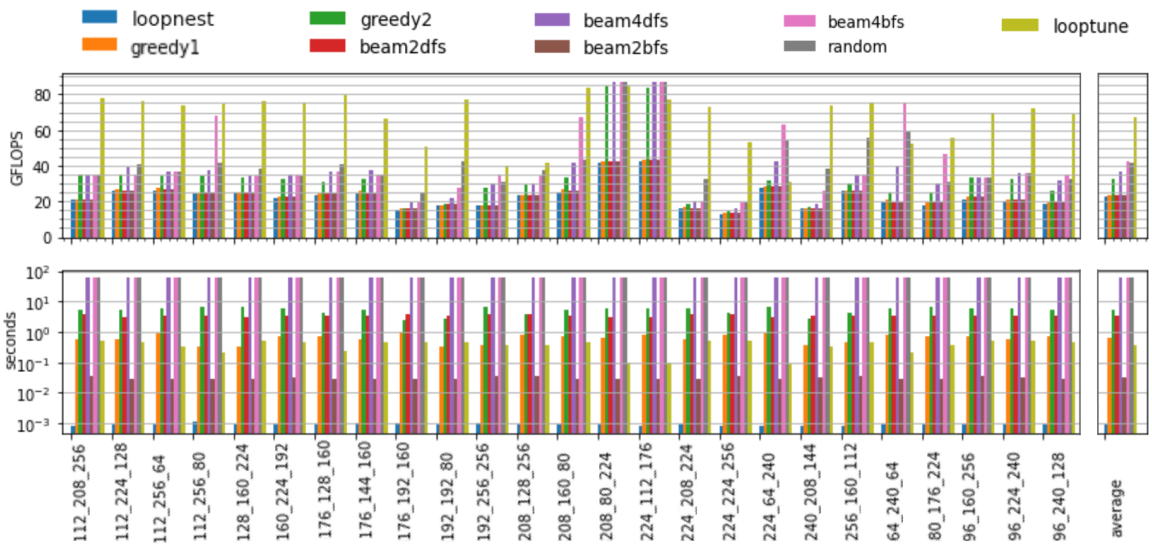


Figure 3.8 : Achieved performance (**higher** is better) and search time (**lower** is better) of randomly selected 25 test benchmarks given 60 seconds for search. The "Original" refers to LoopNest, which was used as a back compiler for greedy, beam, random searches, and the LoopTune method [4].

In 88% test benchmarks, the APEX_DQN policy network outperforms the best traditional searches by 1.8x on average in less than a second, which is an order of

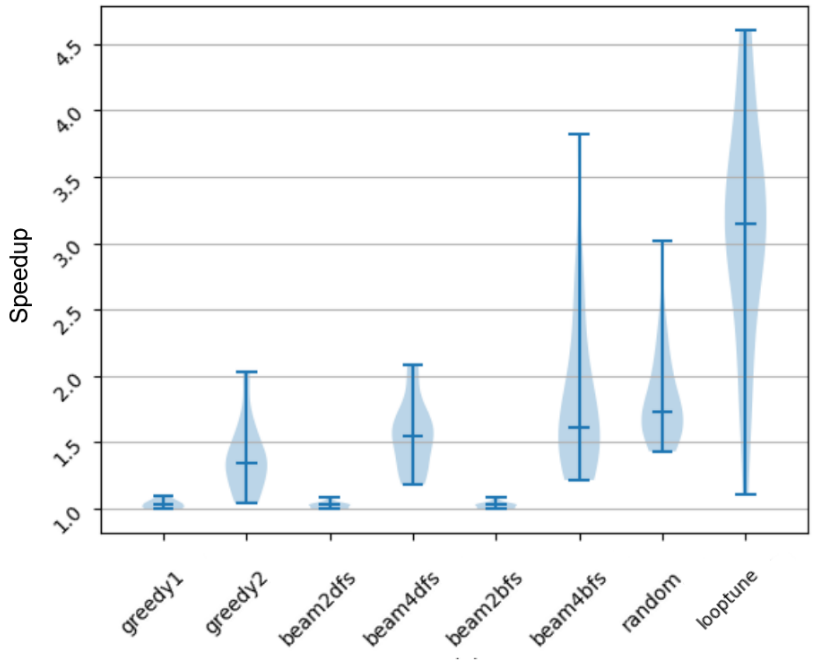


Figure 3.9 : Speedup distribution for searches from Figure 3.8 normalized with LoopNest results [4].

magnitude less time. To better understand the characteristics of each search, we present the speedup distribution in Figure 3.9.

Increasing lookahead to 2 improves Greedy search's performance. Beam2BFS and Beam2DFS achieve poor results despite exploring the entire search subtree, which implies that performant schedules include non-performant actions. Increasing the width to 4 significantly boosts performance, outperforming Greedy2. The success of Random Search further emphasizes that the optimization space is non-linear. Finally, the RL policy network significantly outperforms all search methods by optimizing for long-range rewards up to 10 steps ahead, avoiding local minimum.

3.8.3 Analysis of the loop schedule optimization space

Next, we visualize the performance and search speed of search algorithms and the RL approach for each step (Figure 3.10). The upper figure shows the reward signal in GFLOPS for the best-found schedule, while the lower figure shows how long it takes to choose an action for the given step. For the Depth-First search and Random Search, actions are not decided until the end of the search graph construction and appear flat on the plot.

Greedy1 stops at the local minimum after two steps. Greedy2 expands the graph to depth 6, avoiding a one-step local minimum and achieving better performance but still exploring only a small number of states.

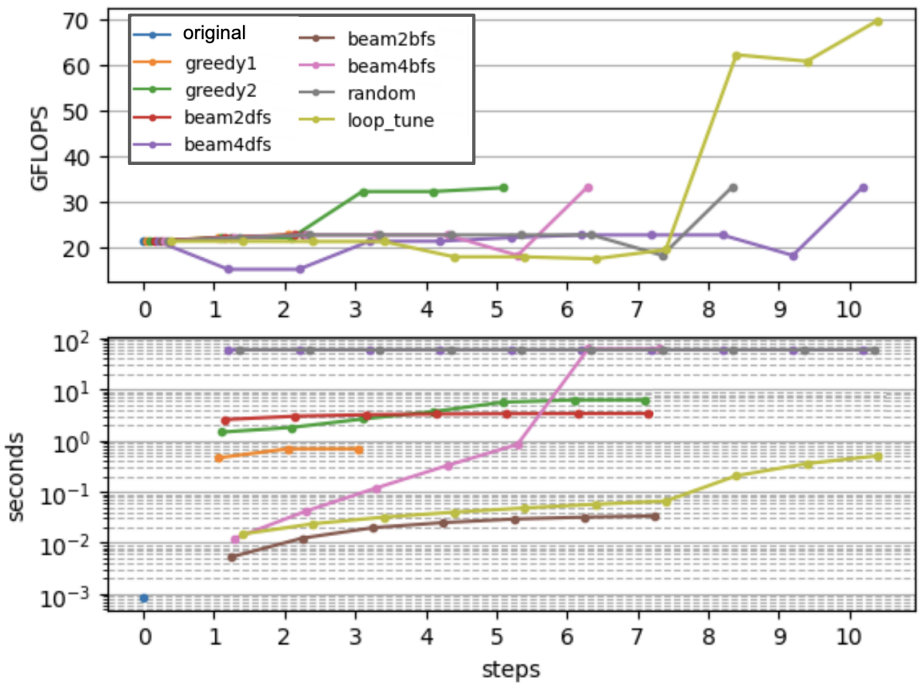


Figure 3.10 : Performance and time needed for expanding a search graph in each step [4].

Beam2DFS expands the graph in-depth, updating each layer during graph construction, keeping the time curve relatively flat. BeamBFS, on the other hand, builds the search space layer by layer, completing lower layers first. The fact that Beam2DFS and Beam2BFS finished before the deadline (60s) means that they constructed the whole search graph of spawn 2. Neither of the two searches found a performant solution indicating that all performant schedules consist of at least one action, which is best two actions.

Beam4DFS and Beam4BFS terminated with a deadline, meaning they only partially constructed their spawn search graph for spawn 4. Beam4DFS's search graph includes solutions with long sequences of up to 10 steps, while Beam4BFS completely explored all solutions with five steps. In both cases, the best solutions contain long sequences of actions with non-monotonically increasing performance, which enables these searches to see further than Greedy searches.

The Random Search uses all time to expand the search graph from root to depth ten without following any metric and evaluating each state in the graph. This way, Random Search can uniformly explore optimization space, including sequences of non-monotonic actions.

RL policy network outperforms all previous algorithms by learning optimization patterns that maximize future rewards, speeding up the execution by 3.2x on average compared to the original LoopNest implementation. It tolerates long sequences of non-performant actions, being worse than all other searches from the 4th to the 7th step to reach a performant state at the 8th step. Additionally, RL policy network search time grows linearly in the length of an action sequence, which enables us to use the policy network on more complex problems that require a larger number of steps. These capabilities are paramount for autotuning general compilers such as LLVM.

3.8.4 Comparison to Numpy, TVM, MetaSchedule, and AutoTVM

Next, we show performance profiles [256] for compilation and execution of generated code on the test dataset (440 examples) and compare it to a popular hand-tuned library for tensor operations – Numpy*, tensor compiler – TVM (base version and optimized version with blocking, permutation, and vectorization) and widely used autotuners – autoTVM and MetaSchedule (Figure 3.11).

LoopTune outperforms all other approaches in 67% of test cases and achieves at least 90% of the best performance in 92% of test cases. On average, LoopTune beats base TVM by 43x, optimized TVM by 9.7x, MetaSchedule by 2.8x, and AutoTVM by 1.08x while being 3% slower than Numpy. Unlike Numpy, LoopTune doesn't require hand-tuning, which reduces development and maintenance costs.

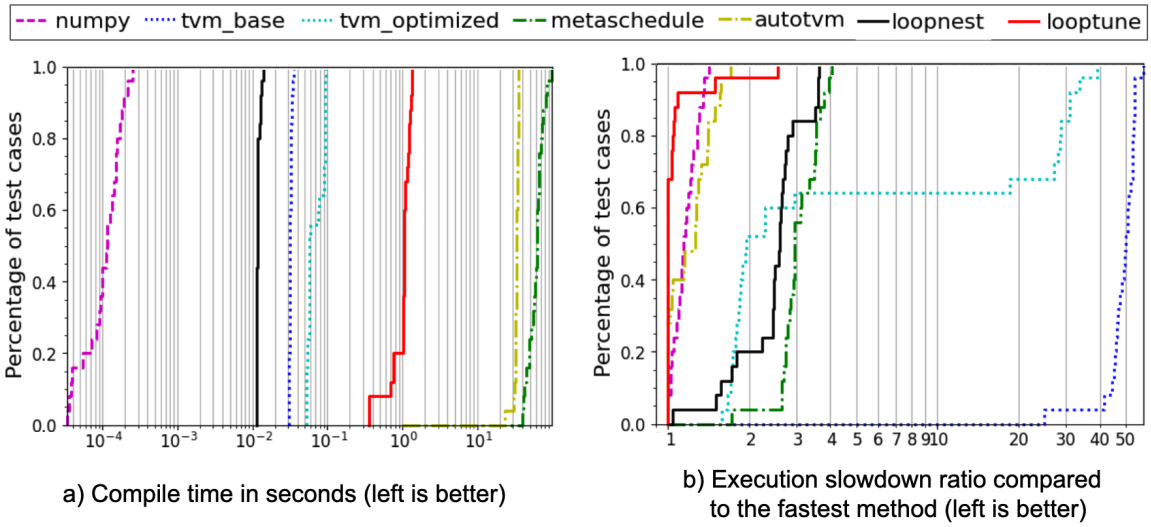


Figure 3.11 : Compile time and Execution ratio of test benchmarks. For Figure b), test cases were normalized with the best method sorted from best to worst on the y-axis [4].

*Numpy uses state-of-the-art OpenBlas implementation of BLAS.

Moreover, LoopTune makes real-time autotuning practical, generating code in just 1 second, while autoTVM and MetaSchedule require 33 and 62 seconds on average. This is particularly important for applications that require downloading and tuning in real-time from web-based repositories. An example can be tuning image/video filters for social media apps and video games for mobile or VR devices.

We used official documentation from TVM [257] to implement matrix multiplication for the examples from the test set. This implementation of TVM includes blocking, loop permutation, and vectorization optimizations, the same set of optimizations we use for LoopTune. We enable the *"llvm -mcpu=core-avx2"* option for TVM, MetaSchedule, and AutoTVM to get the best results for our architecture. For MetaSchedule, we used stochastic sampling, tiling, reordering, and unrolling, while for AutoTVM, we used XGBTuner, evaluating 64 possible schedules for both.

Evaluating more than 64 schedules would require more time, making it prohibitively long for our use case – autotuning in seconds. For the same reason, we don't include in our evaluation popular cost-model-based frameworks such as Ansor [228], Value Learning [229], and TenSet [258], and FlexTensor [259] since they have similar or longer search time.

3.9 Related Work

Tensor specific libraries. Tensor-based mathematical notation was first used by APL [260]. Similar to APL, modern frameworks such as NumPy [261], Matlab's Tensor Toolbox [262], Intel MKL [263], PyTorch [70] and Tensorflow [71] provide an intuitive interface for manipulating tensors, performing customized operations, and executing machine learning algorithms. Although these libraries often vectorize tensor computation, they hardly find the most performant order and sizes of the loop

for custom hardware.

Besides machine learning, tensor computations are used for quantum chemistry simulations. Libraries such as Tensor Contraction Engine [216], LibTensor [264], and Cyclops Tensor [265] provide general and sometimes domain-specific tensor computations. They often use distributed algorithms and tensor blocking.

Search-based compilers. Rather than relying on one-size-fits-all solutions from expert-written libraries, projects ATLAS [106] and FFTW [266] empirically optimize BLAS and FFT routines given custom hardware. PetaBricks [108] chooses the most appropriate computation algorithm for the given platform and tunes its parameter using iterative methods. OpenTuner [267] provides an ensemble of method-agnostic search techniques for program autotuning. Although these methods can be highly effective, autotuning requires significant search times for each program, which can be prohibitively expensive. LoopTune takes only a second to tune tensor computations.

Graph-based compilers. nGraph [150] passes its graph internal representation to a transformer and generates optimized code for the selected backend. XLA [151] automatically replaces subgraphs from Tensorflow with optimized binaries. Glow [268] applies domain-specific optimization at a high level, memory-related optimizations at instruction-based intermediate representation, and hardware-specific optimization at the lowest level. MLIR [269] provides extensible compiler infrastructure that aims to unify domain-specific optimizers, providing multiple representations and layers of optimization. Rather than using a complex representation, LoopTune encodes graph-based representation to simple vectors with relevant features to describe memory access patterns. This enables fast inference with a simple multi-layer perceptron.

Scheduling-based compilers. Halide [205] is the first influential work to propose the separation of computation and schedule for optimizing image processing

and tensor computations. It uses a declarative language to specify tensor computations and a separate language for scheduling its execution. Like LoopTune, Halide's scheduling language includes operations such as splitting and reordering, with the addition of vectorizing, unrolling, and parallelizing loops. TVM [206] extends Halide's compute/schedule concept with hardware intrinsics and defines new optimizations such as tensorization and latency hiding. AutoTVM [270] extends the TVM cost model and adds a template-guided search framework. FlexTensor [259] search directly schedule primitives on the finner-graned level than templates. In contrast to these approaches, LoopTune defines the action space with a policy model in mind, eliminating parametrized actions that are hard to learn [249].

Polyhederal based compilers. To represent tensor computation polyhedral optimizers Polly [225] and others [271] [272] use linear programming and affine transformations to optimize loops static control-flow. Tensor comprehensions [273] uses Halide's intermediate representation to represent computation, polyhedral representation to represent loops, and just-in-time compilation for GPU.

Cost-model based compilers. To speed up the evaluation of a computation, popular frameworks such as Ansor [228], Value Learning [229], and TenSet [258] learn a cost model to evaluate performance and use decision trees, evolutionary search, and Monte-Carlo tree search to identify the best one. Although the performance cost model reduces evaluation time, converging to the optimal state in highly non-convex action spaces is challenging. Additionally, the inference with the cost model and basic greedy search requires *actions_sequence_len * number_of_possible_actions* inferences, while the policy network requires only *actions_sequence_len* inferences, which is precisely the case for LoopTune.

Policy-model based compilers. Neurovectorizer [159] uses deep RL to improve

the vectorization of CPU loops by tuning vectorization width and interleaving count. Chameleon [160] uses a policy network to guide an adaptive sampling algorithm with domain knowledge to search configuration space. MLGO [274] uses Policy gradient and Evolution strategies to optimize binary size by inlining functions. PolyGym [209] explores loop schedules combining polyhedral representation with RL and provides infrastructure for the user to apply different RL algorithms utilizing their representation. In contrast to these approaches, LoopTune employs a novel graph-based representation, action space, and methodology for optimizing loop nests.

3.10 Discussion

LoopStack is a novel, extremely efficient code generator tailored to machine learning based approaches for optimizing machine learning workloads. It has orders of magnitude faster compilation and performs better than hand-tuned libraries. Besides this, LoopStack can replace the cost models in some ML-based autotuners by providing performance measurements.

LoopStack optimizes tensor contractions by providing the ultra-fast, lightweight domain-specific code generator LoopNest and using reinforcement learning to find performant loop schedules with LoopTune. LoopNest implements limited numbers of HPC optimizations, following the order of computation the user requested. This is crucial for ML-based tuners, as hidden functionality in traditional compilers makes learning problems extremely complex. LoopTune uses deep RL to train a policy network that reorders, and tiles loop nests. To map this problem to reinforcement learning, LoopTune introduces a unique action space, graph-based state representation, and reward signal.

Using RLlib's APEX DQN algorithm, LoopTune speeds up the original LoopNest

implementation by 3.2x, given one second on a suite of test problems. In contrast, the best traditional search algorithm achieved 1.8x, given 60 seconds. LoopTune achieves an order of magnitude better results than the optimized implementation of TVM, which includes blocking, loop permutation, and vectorization. Additionally, LoopTune outperforms MetaSchedule and AutoTVM by 2.8x and 1.08x on average, generating code again in 1 second, while MetaSchedule and AutoTVM require 33 seconds and 62 seconds, respectively. This makes real-time auto-tuning possible.

Finally, LoopTune consistently performs at the same level as the expert-optimized library Numpy, significantly reducing development efforts. This finding further supports the belief that deep reinforcement learning techniques will play an essential role in the next generation of compilers.

3.10.1 Constant Loop Bounds

One of LoopTune's limitations is that the loop nest shape needs to be known at compile time. For most ML computations, forward and backward propagation are defined as chains of matrix multiplications together with element-wise activation functions. These operations are fixed in size, which is defined by design.

At the time, there is no support for symbolic ranges of the loop, which means that one loop end cannot depend on the previous loop iterator. In principle, we could approximate triangular memory access patterns, but such an approximation would be complex to derive in a general case.

3.10.2 Computation Size

Another limitation is the size of the computation workload. The training time of the policy network is proportional to the execution time of the loop nest since we

evaluate performance explicitly. This is not the problem for small kernels, while for larger kernels, it might be necessary to use a cost model during training. The cost model would take our vectorized graph-based representation and predict performance in GFLOPS of a given loop nest. Once the cost model is trained, the reward signal for each loop nest will be derived in constant time.

3.10.3 New Hardware Support

LoopTune only supports single-core CPUs with vector-level parallelism. To support thread-level parallelism and other architectures, such as GPUs or FPGAs, we need to add their support to our backend, LoopNest, and retrain the policy model with LoopTune. A newly trained policy network will create a loop schedule that utilizes special backend features, such as vector parallelism. This is achieved through the reward signal, which implicitly depends on these features.

Support for a new backend could be added much faster than developing an optimized library. For example, extending LoopNest to support Apple's M1 chip took less than ten engineering days in fewer than 1,000 lines of code [275]. After this, we would need to train the policy model with LoopTune, which takes less than one day.

Chapter 4

Large Language Models for Compiler Optimization

4.1 Introduction

There is increasing interest in large language models (LLMs) for software engineering domains such as code generation [168, 169, 174, 175, 276–280], code translation [181, 281], and code testing [177,178,180]. Models such as Code LLaMa [169], Copilot [282], and ChatGPT [170] have a good statistical understanding of code and suggest likely completions for unfinished code, making them helpful in editing and creating software. However, it appears that they have not been explicitly trained to optimize code. ChatGPT, for instance, will make minor tweaks to a program, such as tagging variables to be stored as registers, and will even attempt more substantial optimizations, such as vectorization. However, it quickly gets confused and makes mistakes, frequently resulting in incorrect code.

Prior work on machine learning guided code optimization has used hand-built features [1,161,283], all the way to graph neural networks (GNNs) [149,284]. However, in all cases, the machine learning algorithm's representation of the input program is incomplete, and some information is lost along the way. For example, MLGO [161] uses numeric features to provide hints for function inlining but cannot faithfully reproduce the call graph or control flow, etc. ProgramL [149] forms graphs of the

program to pass to a GNN, but it excludes values for constants and some information that prevents reproducing instructions with fidelity.

In this work, we ask: Can large language models learn to optimize code? LLMs can accept source programs as is, with a complete, lossless representation. Using text as the input and output representation for a machine learning optimizer has desirable properties: text is a universal, portable, and accessible interface, and unlike prior approaches, it is not specialized to any particular task.

We began our investigation into the code-optimizing power of LLMs by replicating the optimizing transformations of LLVM [12] compiler. LLVM's optimizer is extremely complex and contains thousands of rules, algorithms, and heuristics in over 1M lines of C++ code. We expected that while LLMs have shown significant progress in natural language translation and code generation tasks, they would be incapable of emulating such a complex system. Understanding and applying compiler optimizations require multiple levels of reasoning, arithmetic computation capabilities, and using complex data structure and graph algorithms, which are capabilities LLMs have been shown to lack [285, 286].

We thought this line of research would reveal LLMs' obvious failings and motivate future clever ideas to overcome those failings. We were entirely taken by surprise to find that, in many cases, a sufficiently trained LLM can not only predict the best optimizations to apply to an input code but can also directly perform the optimizations without resorting to a compiler at all!

Our approach is simple. We begin with a 7B-parameter LLM architecture, taken from LLaMa 2 [287], and initialize it from scratch. We then train the model on million examples of LLVM assembly to predict performant optimization sequences that reduce code size. Each example contains input LLVM assembly, the best compiler options

found by a long-running search, and the resulting assembly from performing those optimizations. From these examples alone, the model learns to optimize code size with remarkable accuracy.

Given one compilation, our approach achieves a 3.0% improvement in code size reduction over the compiler, while a search-based approach achieves 5.0% with 2.5e9 compilations. Furthermore, the two state-of-the-art ML approaches we evaluated caused regressions and required thousands of compilations.

We provide auxiliary experiments and code examples to further characterize the potential and limits of LLMs for code reasoning. The model shows surprisingly strong code reasoning abilities, generating compilable code 91% of the time and perfectly emulating the output of the compiler 70% of the time. Overall, we find their effectiveness remarkable and think that these results will be of interest to the community.

4.2 LLVM Pass Ordering with Large Language Models

In this work, we target compiler pass ordering. The pass ordering task is to select from the set of optimizing transformation passes available in a compiler the list of passes that will produce the best result for a particular input code. Manipulating pass orders has been shown to have a considerable impact on both runtime performance and code size [1, 288].

Machine learning approaches to this task have shown promising results previously but struggle with generalizing across different programs [163]. Previous studies typically need to compile new programs thousands of times to try various configurations and find the best-performing option, making them impractical for real-world use. We hypothesized that a large language model with sufficient reasoning power could learn to make good optimization decisions without needing this.

Most prior work on LLMs for code operates on source languages such as Python. Instead, for the pass ordering problem, we require reasoning at the lower level of LLVM Intermediate Representation (IR). While there exist curated datasets of source languages for pretraining LLMs (e.g. [289–291]), compiler IRs do not make up a significant portion of these datasets, and their ability to reason about IR is far inferior to source languages.

We target optimizing LLVM pass orders for code size as in prior work [161, 163], using IR instruction count as an (imperfect) proxy for binary size. This approach is agnostic to the chosen compiler and optimization metric, and we intend to target runtime performance in the future. For now, optimizing for code size simplifies the collection of training data.

4.2.1 Prompt Structure

We present the model with an unoptimized LLVM-IR (such as emitted by the *clang* frontend) and ask it to produce a list of optimization passes that should be applied to it. Figure 4.1 shows the format of the input prompt and output text we use during training.

We target LLVM 10 and use the optimization flags from `opt`. There are 122 optimization passes, which can be selected more than once in a single sequence. We also include the six meta-flags (-O0, -O1, -O2, -O3, -Oz, and -Os) that may occur only once per pass list. Pass lists can be any length, though, in our experiments, we found typically up to 9 passes long for a combinatorial search space of around 10^{18}.

As shown in Figure 4.1, we also include two auxiliary tasks: 1) generating the instruction counts of the code before and after optimizations are applied and 2) generating the output IR after optimizations are applied. We hypothesize that these

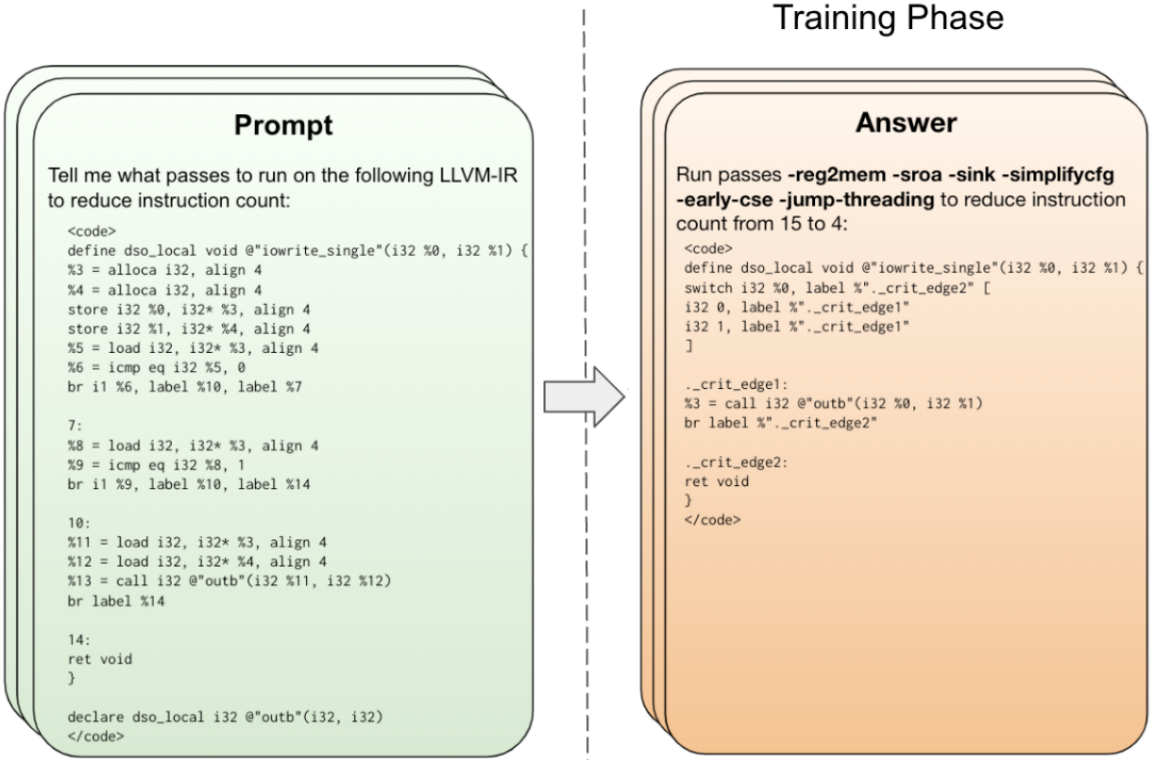


Figure 4.1 : Overview of our approach, showing the model input (Prompt) and output (Answer) during training. The prompt contains unoptimized code. The answer includes an optimization pass list, instruction counts, and the optimized code [6].

would enable better pass-ordering decisions by forcing a deep understanding of the mechanics of code optimization. We verify this experimentally in Section 4.5.2.

While the model is trained to generate instruction counts and optimized IR, we do not need those auxiliary tasks for deployment. All we need to do is generate the pass list that we could execute by the compiler (Figure 4.2). This enables us to keep generation costs on the order of seconds. Moreover, we sidestep the correctness problems that plague techniques that require the model output to be trustworthy [181, 281, 292, 293].

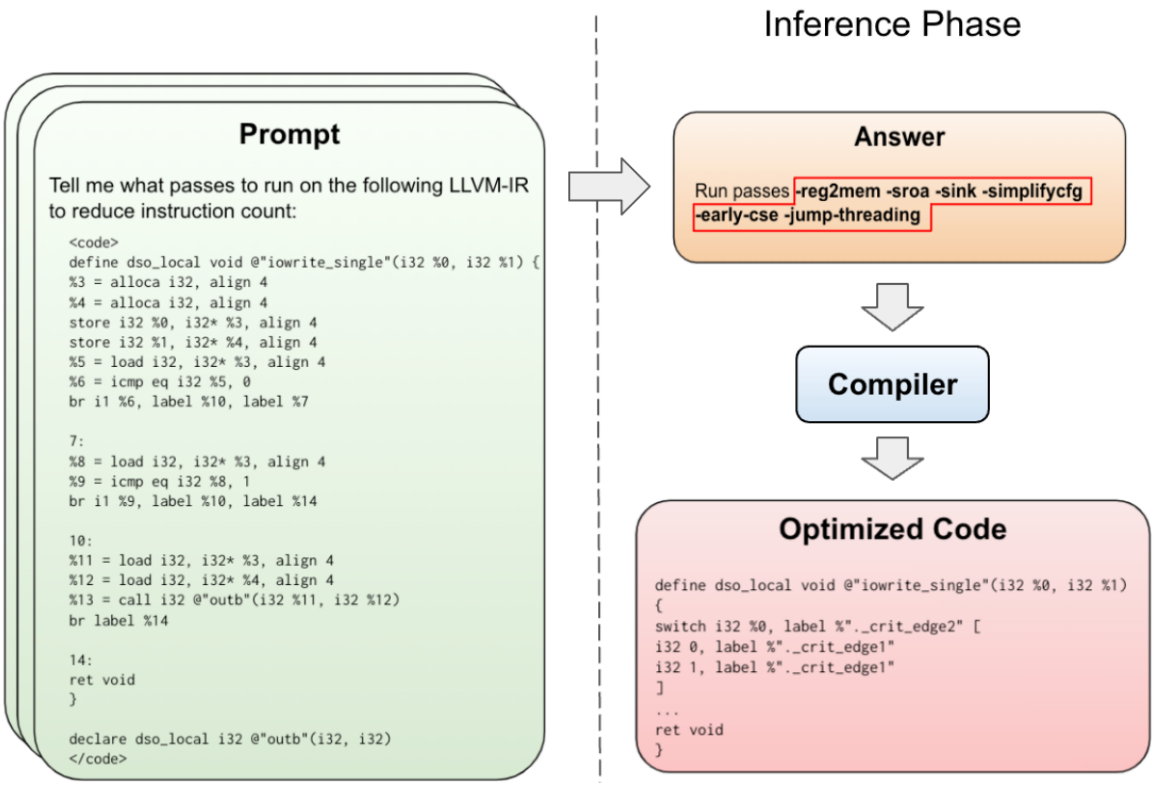


Figure 4.2 : Overview of our approach, showing the model input (Prompt) and output (Answer) during inference. The prompt contains unoptimized code. The answer comprises only an optimization pass list, which we feed into the compiler, ensuring the optimized code is correct [6].

4.2.2 LLVM-IR Normalization

We normalize the LLVM-IR that is used for training the LLM using the following rules: we discard comments, debug metadata and attributes, and ensure consistent whitespace by feeding the IR through a custom lexer that retains newlines but standardizes other whitespace and strips indentation. We do this to reduce the length of the LLVM-IR to make maximum use of the limited input size of the LLM.

4.3 Training Methodology

We use the ubiquitous transformer architecture [69]. The transformer is an artificial neural network that employs self-attention over a fixed-size context window.

The input text is first tokenized into words and subword units. These are embedded into continuous vector representations and provided as input to the transformer's decoder, where self-attention mechanisms capture contextual relationships between tokens to encourage the model to understand and process the input text's semantic structure.

The output text is produced by iteratively generating one token at a time. The decoder takes the encoded input and any previously generated tokens and uses self-attention to predict the next token in the sequence. We greedily sample during decoding to select the most likely token sequence. This process continues until an end-of-sequence token is generated or a predefined maximum length is reached.

4.3.1 Model Architecture

We use the same model architecture and Byte Pair Encoding (BPE) [294] tokenizer as Llama 2 [287], but train our model from scratch. We use the smallest Llama 2 configuration: 32 attention heads, 4,096 hidden dimensions, and 32 layers for a total of 7B parameters.

The maximum length of a (prompt, answer) pair is defined by the sequence length. In this work, we use a sequence length of 2,048 tokens. The Llama 2 tokenizer achieves an average of 2.02 characters per token when encoding LLVM-IR, so this provides an approximate upper limit on the longest LLVM-IR we can train on at 2KB (since 2KB prompt and 2KB answer $\approx$ 2,048 tokens).

4.3.2 Training Data

We assembled a large corpus of unoptimized LLVM-IR functions, summarized in Figure 4.3. We extracted the functions from publicly available handwritten C/C++ code datasets and supplemented this with synthetic code generated by C/C++ compiler test generators. The training corpus comprises 1,000,000 deduplicated IR functions, totaling 373M training tokens. We operate at the level of individual IR functions rather than entire modules to maximize the data we can fit inside a 2,048-token sequence length.

To find the list of optimization passes that will produce the smallest instruction count, we employ *autotuning*. Our autotuner combines random search and all-to-all results broadcasting between functions, inspired by the work of Liang et. al. [284]. For each function, we run a random search for a fixed amount of time (780 seconds) and then minimize the best pass list by iteratively removing randomly chosen passes to see if they contribute to the instruction count. If not, they are discarded. After performing this on each function, we aggregate the set of unique best pass lists and broadcast them across all other functions. Thus, if a pass list works well on one function, it is tried on all others.

In total, the autotuner compiled each training program an average of 37,424 times, achieving a 5.8% improvement in instruction count reduction over the baseline fixed pass ordering in the compiler provided by -Oz. For our purposes, this autotuning serves as a gold standard for optimizing each function. While the instruction count savings discovered by the autotuner are significant, the computational cost to reach these wins was 9,016 CPU days. This work aims to achieve some fraction of the performance of the autotuner using a predictive model that does not require running the compiler thousands of times.

	n functions	unoptimized instruction count	size on disk	n tokens
Handwritten	610,610	8,417,799	653.5 MB	214,746,711
Synthetic	389,390	13,775,149	352.3 MB	158,435,151
Total	1,000,000	16,411,249	1.0 GB	373,181,862

a) Training data

	n functions	unoptimized instruction count	-Oz instruction count
AI-SOCO [31]	8,929	97,800	47,578
ExeBench [32]	26,806	386,878	181,277
POJ-104 [33]	310	8,912	4,492
Transcoder [12]	17,392	289,689	129,611
CSmith [34]	33,794	647,815	138,276
YARPGen [35]	12,769	285,360	144,539
Total	100,000	1,716,354	645,773

b) Test data

Figure 4.3 : Training and test data. Each LLVM-IR function is autotuned to create a (Prompt, Answer) pair. The n tokens column shows the number of tokens when the prompt is encoded using the Llama 2 tokenizer. -Oz instruction count is instruction count after applying -Oz flag [6].

4.3.3 Training Configuration

Starting from randomly initialized weights, we trained the model for 30,000 steps on 64 V100s for a total training time of 620 GPU days. We use the AdamW optimizer [295] with β_1 and β_2 values of 0.9 and 0.95. We use a cosine learning rate schedule with 1,000 warm-up steps, a peak learning rate of $1e{-}5$, and a final learning rate of 1/10th of the peak. We used a batch size of 256; each batch contains 524,288 tokens for

15.7B training tokens. The full 30,000 training steps are 7.7 epochs (iterations over the training corpus).

During training, we evaluated the model on a holdout validation set of 1,000 unseen IRs that were processed similarly to the training set. We evaluate every 250 steps.

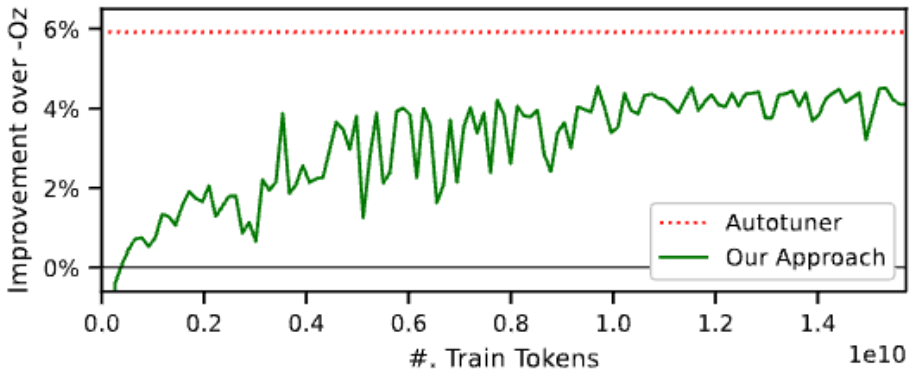
4.3.4 Training Results

Figure 4.4 (a) shows the performance during training when evaluated on a holdout validation set of 1,000 unseen LLVM-IR functions. The model achieved peak validation performance at 10.9B training tokens.

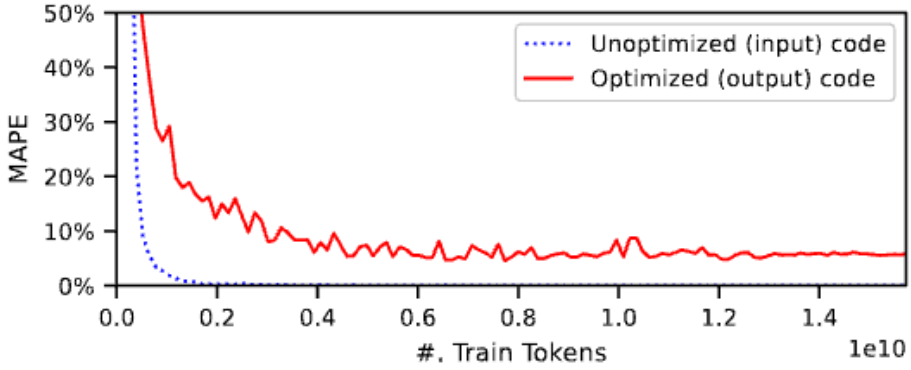
At peak performance, the code optimized using model-generated pass sequences contains 4.4% fewer instructions than when optimized using the compiler's built-in pass ordering (-Oz). The autotuner achieves a greater instruction count reduction of 5.6%, but this required 27 million compilations of the validation set. The model makes its predictions without invoking the compiler once.

Figure 4.4 (b) shows the error of predicted input and output instruction counts. Prediction of instruction counts for unoptimized code rapidly approaches near-perfect accuracy. Prediction of output instruction count proves more challenging, reaching a Mean Average Percentage Error (MAPE) of 5.9%.

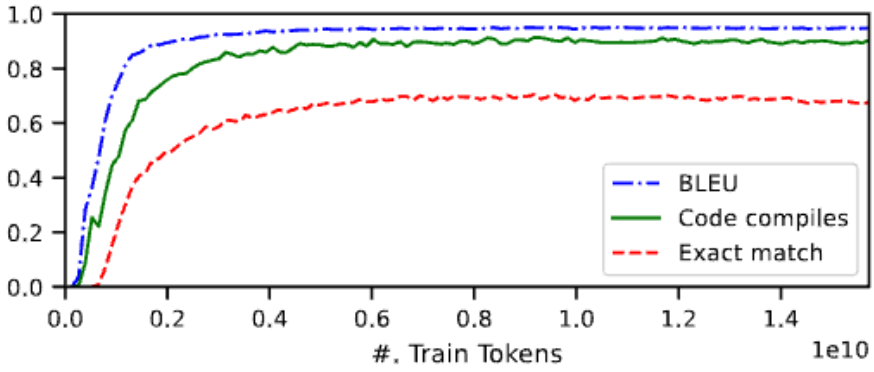
Figure 4.4 (c) evaluates the quality of the generated code using three metrics. The *BLEU* [296] score shows the similarity between the model-generated code and a reference ground-truth code produced by the compiler using the generated pass list. *Code compiles* is the frequency model-generated code compiles without error. *Exact match* tracks the frequency that the model-generated code is a character-by-character match of the compiler-generated code when optimized using the generated pass list.



(a) Performance of generated pass lists.



(b) Accuracy at predicting instruction counts.



(c) Model-optimized code metrics.

Figure 4.4 : Performance on holdout validation set during training. We evaluate performance every 250 training steps (131M train tokens). Parity with -Oz is reached at 393M tokens and peak performance at 10.9B tokens [6].

At peak performance, the model generates code that compiles without errors 90.5% of the time. A BLEU score of 0.952 shows that the model-optimized code closely approximates that of the compiler, and the exact match frequency is 70%. For comparison, a baseline that copies the unoptimized code to the output would achieve a BLEU score of 0.531 and an exact match frequency of 0%, demonstrating that significant manipulation of the input code is required to achieve such high scores.

By the end of training, performance on the validation set had plateaued. We use the best-performing checkpoint and switch to a 100× larger-scale evaluation.

4.4 Experiments

In this section, we evaluate the model's ability to generate pass lists for unseen code and correctly perform optimization.

4.4.1 Comparison to State-of-the-Art

In this experiment, we perform a large-scale evaluation on 100,000 test examples to compare the LLM's ability to predict pass lists to baselines.

Datasets

We aggregate a broad suite of benchmark datasets for evaluation, summarized in Figure 4.3 (b). We deduplicate and exclude IR functions identical to those we used as training inputs. Our test data comprises code from a variety of domains, including coding competitions (AI-SOCO [297], POJ-104 [298]), compiler test case generators (CSmith [188], YARPGen [299]), and miscellaneous publicly available code (ExeBench [300], Transcoder [181]).

Baselines

We compare our approach to three baselines: AutoPhase [301], Coreset-NVP [284], and the autotuner.

AutoPhase [301] is a reinforcement learning approach in which an agent is trained using Proximal Policy Optimization [252] to select the sequence of optimization passes that will maximize cumulative instruction count savings over a fixed-length episode. At each step, the program being optimized is presented to the agent as a 56-dimensional vector of instruction counts and other properties. We replicate the environment of [301] but use the implementation and expanded training regime from [163] in which the agent is trained for 100,000 episodes. We train the agent on the same data as our language model (Figure 4.3) and evaluate agent performance periodically during training on a holdout validation set. As in prior work, we use an action space and episode length of 45.

Coreset-NVP [284] is a technique that combines iterative search with a learned cost model. First, a greedy search is run on 17,500 benchmarks to determine a *Core set* of best pass lists. Then a *Neural Value Prediction* (NVP) is trained on the results of this search, using ProGraML [149] graphs processed by a Graph Convolutional Network as program representation. At inference, Coreset-NVP predicts the normalized reward and tries the first few pass sequences with the highest normalized reward. The total number of passes it is allowed to try for each benchmark is 45, following prior work. We use author-provided model weights to perform inference on our test set.

Finally, we compare it to the autotuner we used to generate training data. We autotuned the test dataset in the same manner as the training data, described in Section 4.3.2.

	additional compilations	functions improved	functions regressed	instructions saved	instructions regressed	overall improvement
Autotuner	2,522,253,069	6,764	0	30,948	0	5.03%
AutoPhase [39]	4,500,000	1,558	8,400	6,522	32,357	-3.85%
Coreset-NVP [20]	442,747	3,985	6,072	16,064	28,405	-1.88%
Our Approach	0	4,136	526	21,935	3,095	3.01%

Figure 4.5 : Performance of different approaches for pass ordering on a test set of unseen LLVM-IR functions from Figure 4.3. All metrics are *w.r.t.* -Oz. *Instructions saved* is summed over *functions improved* and *instructions regressed* is summed over *functions regressed*. *Overall improvement* is the sum total instruction count savings *w.r.t* -Oz. The autotuner performs best but requires 2.5B additional compilations (949 CPU days). Our approach achieves 60% of the gains of the autotuner without invoking the compiler once [6].

Results

Figure 4.5 summarizes the results. Our approach outperforms -Oz, AutoPhase, and Coreset-NVP across all datasets. AutoPhase and Coreset-NVP can identify pass lists that outperform -Oz but negatively impact instruction count negatively due to many regressions. We propose a simple "-Oz backup" extension to overcome this: if a model predicts a pass list *other than* -Oz, we also run -Oz and select the best of the two options.

This prevents regressions *w.r.t.* -Oz but increases the number of additional compilations by the number of times the model predicts a pass list other than -Oz. Figure 4.6 shows the results of the techniques when evaluated in this manner. While this does not help the models find further improvements, the lack of regressions means

	additional compilations	overall improvement
AutoPhase [39]	4,600,000	1.02%
Coreset-NVP [20]	542,747	2.55%
Our Approach	5,721	3.52%

Figure 4.6 : Extending the models in Figure 4.5 with "-Oz backup". If a model predicts a pass list *other than* -Oz, it also evaluates -Oz and selects the best. This prevents regressions *w.r.t* -Oz at the expense of additional compilations [6].

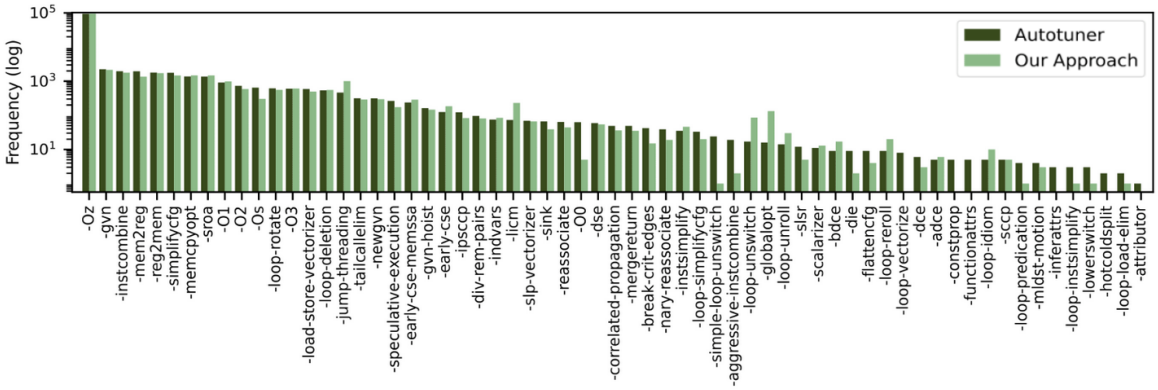
that AutoPhase and Coreset-NVP improve over -Oz, though still less than the LLM with or without the -Oz backup.

4.4.2 Evaluation of Generated Pass Lists

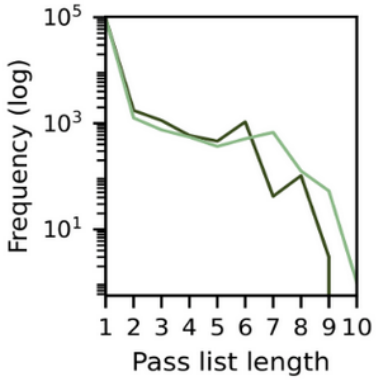
Figure 4.7 shows the frequency with which passes are selected by the autotuner and our model from the previous experiment. The distribution of passes selected by the model broadly tracks the autotuner. -Oz is the most frequently optimal pass. Excluding -Oz, model-generated pass lists have an average length of 3.4 (max 10), and autotuner pass lists have an average length of 3.1 (max 9). Additionally, the model generated 105 pass lists that are not present in the training data.

Figure 4.8 breaks down the improvement of each approach to pass ordering by benchmark dataset. The most significant improvements over -Oz is found in the POJ-104 and Transcoder datasets, which both aggregate large amounts of handwritten code, while the programs generated by YARPGen (used for testing compilers) have the fewest opportunities for improving over -Oz.

We discovered a strong correlation between the input program size and the poten-



(a) Frequency of a pass in the generated pass list for each test program.



(b) Length of generated pass list for each test program.

Figure 4.7 : Frequency of individual passes and the length of generated pass list for each of the 100,000 test programs. -Oz is the starting point for the autotuner and is the dominant result, being the best-found result for 93.2% of autotuned test programs and appearing in an additional 0.6% of pass lists as part of a longer sequence. The model-generated pass distribution tracks the autotuner but slightly overpredicts -Oz (94.3%) and includes nine passes that the autotuner used on the training set but not the test set. Results are ordered by decreasing autotuner frequency [6].

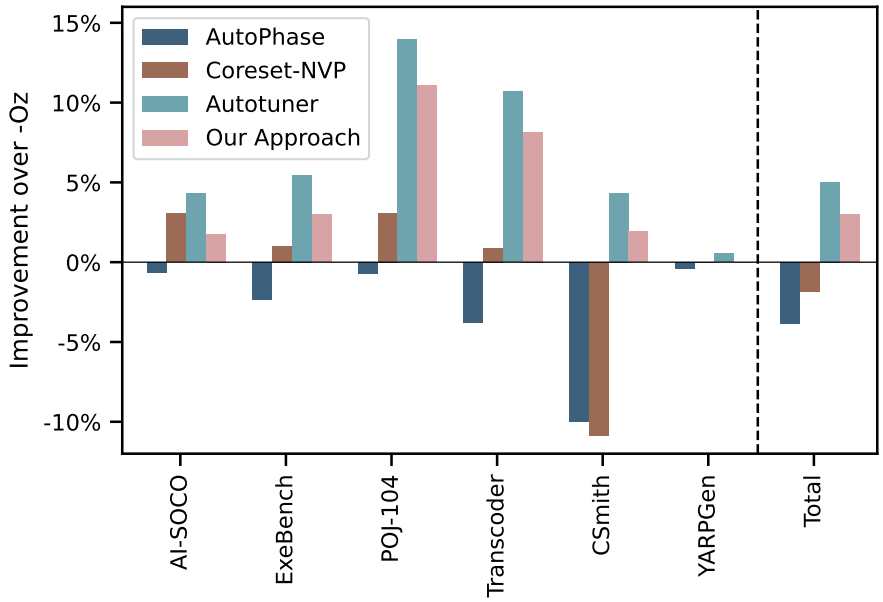


Figure 4.8 : The improvement over -Oz by dataset [6].

tial performance improvement over -Oz found by both the autotuner and the model. Figure 4.9 plots this trend, showing that larger programs have more opportunities to improve over -Oz.

This is an encouraging result since our training set of 1M IR functions consists of only 15% of the data available for training. Once the efficiency of LLMs with large context sizes increases, we can train models with functions larger than 1024 tokens or even entire programs and achieve significantly more significant improvements.

4.4.3 Evaluation of Generated Code

In this section, we evaluate the quality of model-generated code. To do this, we ran the auxiliary training task of generating optimized code for all 100k functions in the test set. Note that this is not required to generate the pass lists evaluated in the

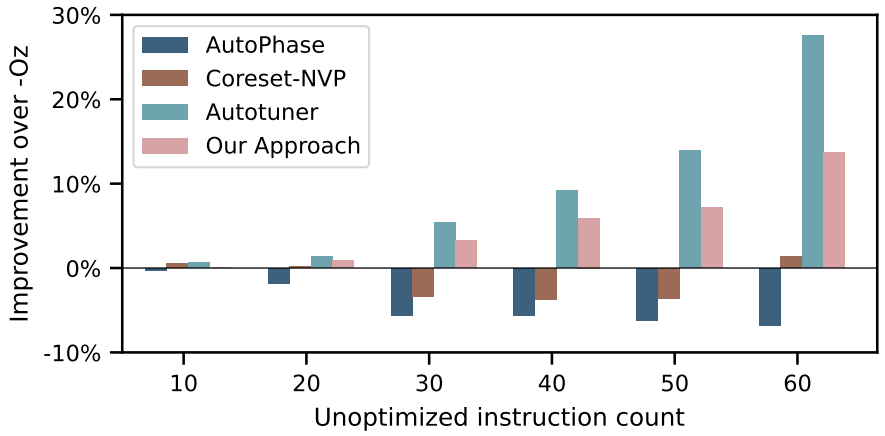


Figure 4.9 : The improvement over -Oz compared to the input size. Larger codes optimize more [6].

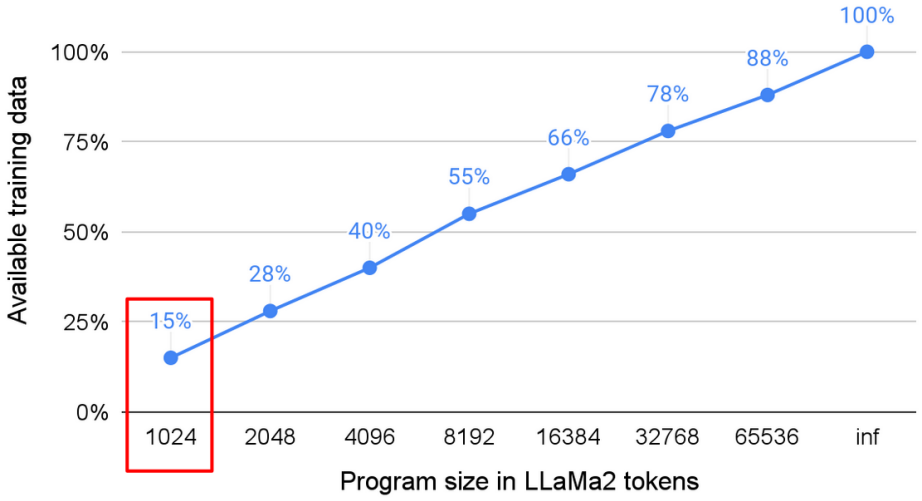


Figure 4.10 : Distribution of data given maximum program size in LLaMa2 token count. Our training dataset consists of programs smaller than 1024 tokens — only 15% of available data.

error category	n
type error	5,777
instruction forward referenced	1,521
undefined value	1,113
invalid redefinition	616
syntax error	280
invalid value for constant	144
undefined function	112
index error	98
other	83
Total	9,744

Figure 4.11 : Compiler errors of model-optimized code on 100,000 unseen inputs [6].

previous section. We have made minor edits to the code samples in this section for brevity, such as omitting superfluous statements and shortening identifier names.

In 90.3% of cases, the model-generated optimized IR compiles, and in 68.4% of cases, the output IR matches character-for-character the ground truth generated by the compiler. We categorize different classes of errors for the 9.7% of cases where the generated IR does not compile in Figure 4.11. Figure 4.12 provides code examples.

Most challenging to evaluate are the 21.9% of cases where the model-optimized code compiles but is not a character-by-character match with the compiler output. There are two challenges: the first is that text precision metrics such as BLEU score are sensitive to differences in the code, such as variable names and commutative operand order, that do not affect the code's behavior. Tools such as LLVM-Canon [302] can help here but come with their own set of drawbacks. However, in many cases, it is unclear whether the behavior of two IRs is the same, so the second challenge we face is evaluating *semantic* equivalence. Since not all datasets we use

```
error: '%15' defined with type 'i32' but expected 'i1'
%or.cond = or i1 %14, %15
```

(a) The model defined `%15` as an integer but later tried to use it as a bool *(type error)*.

```
error: constant expression type mismatch
@.str = private unnamed_addr constant [493 x i8]
    c"<snip 492 chars ...>", align 1
```

(b) The model omitted a single character when transcribing a 493-character string-literal from the input code *(type error)*.

```
error: floating point constant invalid for type
%1 = tail call i32 @f1(float -0.47799998483256463,
                       float -1.8159999847412109)
```

(c) LLVM requires exact decimal values for floating-point constants. These model-generated values have repeating decimals in binary so are rejected *(invalid value for constant)*.

Figure 4.12 : Compiler errors in model-optimized code [6].

for testing provide driver scripts and input datasets for their code, we cannot use execution-based equivalence checks such as differential testing [303].

Figure 4.13 shows an example of model-generated code with incorrect program semantics. Here, the lower 8 bits of a 64-bit literal are truncated and returned. The compiler performs this calculation and substitutes the correct value. The model recognizes that the expression can be calculated at compile time but fails to compute the proper value. Mathematical reasoning is a known weakness of LLMs [286].

Sometimes, the model generates correctly optimized code but fails to produce the pass list needed to achieve it. Figure 4.14 shows one such example. The model-optimized code and instruction count predictions match the performance of the autotuner, but the model omitted the `-mem2reg` pass needed to achieve this code. The model-generated pass list yields 10 instructions instead of 7.

```
define hidden signext i8 @f1() #0 {
  %1 = alloca i64, align 8
  store i64 3718042838174166437, i64* %1, align 8
  %2 = load i64, i64* %1, align 8
  %3 = trunc i64 %2 to i8
  ret i8 %3
}
```

(a) Input unoptimized code.

```
define hidden signext i8
@f1() #0 {
  ret i8 165
}
```

(b) Desired optimized code.

```
define hidden signext i8
@f1() #0 {
  ret i8 1
}
```

(c) Model-generated code.

Figure 4.13 : An example where the model generates compilable code but fails to compute the correct answer for a numeric expression. Producing the correct result for this expression requires non-trivial mathematical reasoning [6].

Run passes ***-instcombine -simplifycfg*** *to reduce instruction count from* ***14*** *to* ***7:***

```
define dso_local i32 @f1(i32 %0) {
 %2 = load i64, i64* getelementptr inbounds(
   %struct.t2, %struct.t2* @gvar, i64 0, i32 0), align 8
 %3 = icmp eq i64 %2, 0
 %4 = icmp eq i32 %0, 0
 %or.cond = or i1 %3, %4
 %5 = load i32, i32* @S64_MAX, align 4
 %6 = select i1 %or.cond, i32 %5, i32 %0
 ret i32 %6
}
```

Figure 4.14 : An example where the model generates correctly optimized code but fails to produce the pass list needed to produce the desired code [6].

```
define i32 @f1(
  i32 %0,
  i32 %1
) align 2 {
 br label %3

3:
 %i = phi i32 [%7, %6], [2, %2]
 %4 = mul nsw i32 %i, %i
 %5 = icmp sgt i32 %4, %1
 br i1 %5, label %8, label %6

6:
 %7 = add i32 %i, 1
 br label %3

8:
 ret i32 2
}
```

(a) Desired optimized code.

```
int f1(int x, int y) {
  int i = 2;
  while (i * i < y) {
    i += 1;
  }
  return 2;
}
```

(b) Equivalent (hand-written) C code.

```
define i32 @f1(
  i32 %0,
  i32 %1
) align 2 {
 ret i32 2
}
```

(c) Model-optimized code.

Figure 4.15 : An example of an unsafe optimization by the model. The 33-instruction input program (not shown) contains a loop that is not always safe to optimize away. For example, when $y = INT\_MAX$ the loop never terminates [6].

Another class of error is when the model makes unsafe optimizations by failing to analyze the input code (Figure 4.15). In this example, the model eliminates the *while* loop that doesn't change the return value. This is wrong since this loop can become infinite if $i * i$ overflows 32 bits and becomes a negative number. For this to happen, y must be set to an exceptionally high value.

We observe an interesting connection between the quality of pass lists and the corresponding optimized code, shown in Figure 4.16. When the model produces a poor-performing pass list, the quality of the generated code is lower. This indicates that the model "doesn't know" what the optimized code should look like and is less likely to produce a performant optimization pass list. Interestingly, programs that

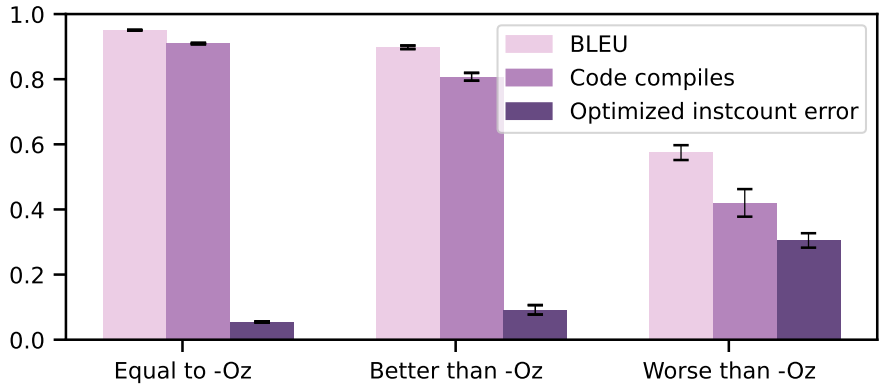


Figure 4.16 : Model-optimized code quality as a function of the performance of the generated pass list. Code quality is lower when the pass list performs worse than -Oz. The model-optimized code resembles the ground truth less (lower BLEU score), the code is less likely to compile, and the model struggles to estimate the instruction count (higher error). Error bars show 95% confidence intervals [6].

perform better than the -Oz code metric also drop slightly. This is expected since the model is expected to generate optimized code after the sequence of optimization passes, rather than just -Oz, for which it has many examples in the training dataset.

4.5 Additional Experiments

In the previous section, we evaluated the performance of an LLM trained to optimize LLVM-IR for code size. This section provides a qualitative analysis of LLMs when generating directly optimized LLVM IR for code size. All models use the same architecture and parameters as in Section 4.3.

4.5.1 Ablation of Dataset Size

We ablate the contribution of dataset size by training two additional models and varying the amount of the training data from 50% (500k examples) down to 25% (250k examples) by random dropout. Figure 4.17 shows progress during the training of the models. For dataset sizes of 50% and 25%, the models begin to overfit the training set after around 8B training tokens. Figure 4.18 shows the peak performance of each configuration. With 50% and 25% of the training data, downstream performance falls by 21% and 24%, respectively.

Figure 4.17 : Ablating the impact of training data size and the auxiliary co-training task of generating optimized code (denoted *No Aux*). Data size is measured as a number of training examples. The graph shows performance on a holdout validation set during training [6].

n training examples	generate optimized code?	overall improvement
1,000,000	✓	4.95% (—)
500,000	✓	3.91% (-21%)
250,000	✓	3.74% (-24%)
1,000,000	×	4.15% (-16%)

Figure 4.18 : Ablation experiments. We evaluate the impact of varying training data size while training the model to optimized code size. We train each model for 30k steps and report the performance of the best model checkpoint on a holdout validation set of 1,000 unseen IR functions [6].

4.5.2 Ablation of Code Optimization Task

We train the model to generate not just a pass list but also the optimized code resulting from this pass list. One may expect this to degrade model performance – not only must it learn to predict good pass lists, but also how to produce correctly optimized code, a more difficult task. In fact, we believe this to be crucial to model performance. By forcing LLMs to learn the semantics of LLVM-IR, we enable them to make better optimization decisions.

To ablate this, we trained a model to generate only pass lists without the corresponding optimized code. We kept the data mix and all other parameters the same. Figure 4.17 and Figure 4.18 show that without training the model to generate optimized code, downstream performance falls by 16%.

4.5.3 Evaluation of Single Pass Translation

In previous sections, we trained LLMs to select optimization passes to produce the best-optimized code. In this section, we evaluate the ability of LLMs to emulate the different optimizations themselves. For this experiment, the model input is an unoptimized IR and the name of an optimization pass to apply, the output is the IR after applying this pass.

Dataset

We generate a new dataset for this task using 60 optimization passes and apply them randomly to the programs from Figure 4.3. We augment the dataset of unoptimized code with partially optimized code by running a sequence of randomly selected passes on unoptimized IRs before the desired target pass. We collect 10,000 unique (prompt, answer) examples for each of the 60 passes for a total of 600k examples.

Model

We trained a new model from scratch on this pass translation dataset. It reached peak performance after 11B training tokens (74 GPU days).

Results

Figure 4.19 summarizes model performance. The average BLEU score overall passes is 0.846, with exact character-by-character matches 73.7% of the time and compilable code 82.3% of the time. We also plot the frequency with which each optimization appears in a model-generated pass list that improved or regressed performance over -Oz in Figure 4.5. We find no correlation between code quality metrics and its frequency in generated pass lists.

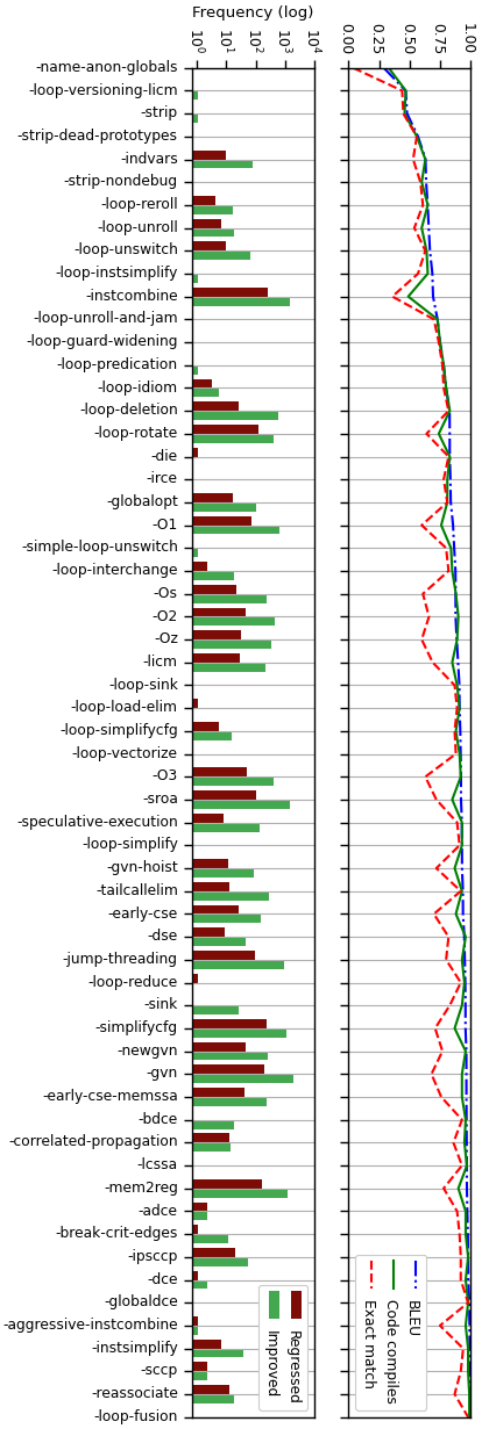


Figure 4.19 : Training a model to predict single optimization passes. The right subplot evaluates the quality of the generated code for the corresponding pass (ordered by BLEU score). The left subplot shows the frequency that the corresponding pass contributed to an improvement or regression of instruction count over -Oz [6].

As can be seen, many passes are learned near-perfectly while others prove more challenging. Some passes that perform poorly hint at simple improvements to the representation, while others result from more profound limitations of the model's reasoning. Figure 4.20 (a) shows an example from the *-name-anon-globals* pass, which is a simple utility pass that renames anonymous global variables using a hash of the module name. Since we do not provide the module name in the prompt, the LLM is forced to hallucinate random values.

Figure 4.20 (b) shows an example from the *-instcombine* pass. This is a complex pass that is implemented in over 4.5k lines of C++ code in LLVM. We see that the model correctly identifies the instructions to combine but makes an error in data flow analysis and substitutes an incorrect value. This important optimization frequently occurs in pass lists that outperform -Oz. We will explore an active learning approach in which more examples are provided for complex and challenging passes.

Finally, we present an example of correct model optimization in Figure 4.21. The example combines several non-trivial code manipulations: register allocation, control flow graph simplification, and instruction combining. We visualize the control and data flow graphs to help interpret the model's changes. Even on the scale of these small IR functions, the sophisticated grasp of LLVM-IR semantics demonstrated by the LLM is remarkable. The model has learned to perform these optimizations entirely from examples without access to the compiler implementation.

4.6 Related Work

Compiler pass ordering for performance has been exploited for decades [91, 288, 304]. Over the years, several approaches have been using machine learning [1, 114, 283, 284, 301, 305]. The application of machine learning in compilers is not limited to pass

```
Optimize the following LLVM-IR using -name-anon-globals:

@0 = private
@anon.2ef3bda806391c61822366a2a59f2569.0 = private
@anon.95277a486ffed0b6ba33ab3385b3d7bd.0 = private
 ↪unnamed_addr constant [14 x i8] c"<snip>", align 1

define dso_local i32 @f1(i8* %0) {
%2 = call i32 @f2(i8* %0, i8* getelementptr inbounds(
 ↪[14 x i8], [14 x i8]*
 ↪@0,
   @anon.2ef3bda806391c61822366a2a59f2569.0,
   @anon.95277a486ffed0b6ba33ab3385b3d7bd.0,
  ↪i64 0, i64 0))
 ret i32 %2
}
```

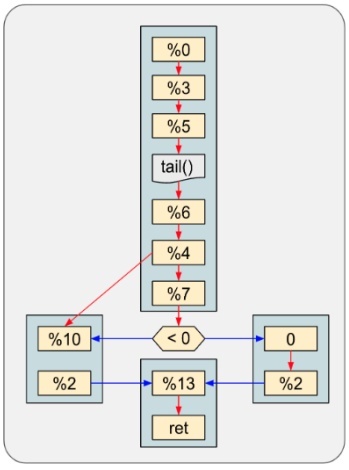
(a) Failure due to incomplete information. The pass *-name-anon-globals* uses the module name to compute a hash. Lacking this, the model hallucinates a random hash.

```
Optimize the following LLVM-IR using -instcombine:

@var_12 = external dso_local global i64, align 8
@var_13 = external dso_local global i32, align 4
@var_14 = external dso_local global i32, align 4

define dso_local void @f1(i64 %arg) {
 %tmp = alloca i64, align 8
 store i64 %arg, i64* %tmp, align 8
 %tmp1 = load i64, i64* %tmp, align 8
 %tmp2 = sub i64 0, %tmp1
 %tmp3 = sub i64 0, %tmp2
 store i64 %tmp3, i64* @var_12, align 8
 store i64 %arg, i64* @var_12, align 8
 store i64 0,    i64* @var_12, align 8
 store i32 1, i32* @var_13, align 4
 store i32 0, i32* @var_14, align 4
 ret void
}
```

(b) Failed data-flow analysis. The model correctly removes redundant instructions but substitutes the wrong value for a variable. The model-optimized code compiles and has a high BLEU score but is incorrect.

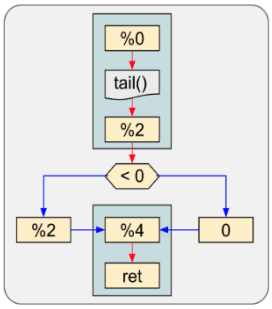
Figure 4.20 : Example failures from the pass translation experiment. We combine the model input (red), ground-truth (blue), and model-generated (green) texts into a single unified diff for brevity. Black text is common to all three [6].

```
Optimize the following LLVM-IR using -Oz:

%s1 = type { i32 }
@"llvm.used" = appending global [1 x i8*]  [i8*
 bitcast(i32(%s1*)* @f1 to i8*)], section "llvm.metadata"

define dso_local i32 @f1(%s1* %0) {
 %2 = alloca i32, align 4
 %3 = alloca %s1*, align 8
 %4 = alloca i32, align 4
 store %s1* %0, %s1** %3, align 8
 %5 = load %s1*, %s1** %3, align 8
 %6 = tail call i32 @f2(%s1* %5)
 store i32 %6, i32* %4, align 4
 %7 = load i32, i32* %4, align 4
 %8 = icmp slt i32 %7, 0
 br i1 %8, label %9, label %11
9:
 %10 = load i32, i32* %4, align 4
 store i32 %10, i32* %2, align 4
 br label %12
11:
 store i32 0, i32* %2, align 4
 br label %12
12:
 %13 = load i32, i32* %2, align 4
 ret i32 %13
}
```



(a) Model prompt.

```
%s1 = type { i32 }
define dso_local i32 @f1(%s1* %0) {
 %2 = tail call i32 @f2(%s1* %0)
 %3 = icmp slt i32 %2, 0
 %4 = select i1 %3, i32 %2, i32 0
 ret i32 %4
}
```



(b) Model-optimized code.

Figure 4.21 : The example of correct generation of optimized IR. The model performed several complex optimizations, including control-flow simplification and replacing if-then-else code blocks with instructions [6].

order and has been applied to many other problems [159, 161, 306–308]. No one has applied LLMs to the problem of pass ordering; we are the first to do so.

Neural machine translation is an emerging field that uses language models to transform code from one language to another. Prior work includes compiling C to assembly [293], assembly to C [292, 309], and source-to-source transpilation [281]. In these works, code correctness cannot be guaranteed, while we use code generation solely as an auxiliary learning task – correctness is supplied by the compiler.

Although language models are used for coding tasks, they are not often used for compilers. Gallagher et al. train a RoBERTA architecture on LLVM-IR for code weakness identification [182] and Transcoder-IR [181] uses LLVM-IR as a pivot point for source-to-source translation. Neither use LLMs for optimization as we do.

Many language models have been trained on source code, including CodeBERT [171], GraphCodeBERT [172], and CodeT5 [173], which are trained to perform multiple tasks, including code search, code summarization, and documentation generation. LLMs trained on source code have also been used for program fuzzing [177–179], test generation [180], and automated program repair [183, 184, 310]. Many practical applications have been explored for language models; however, this is the first work where an LLM is explicitly used to optimize code.

Most LLMs are trained partly on code [276, 277, 287, 311]. Some LLMs are trained similarly to general models but especially target programming languages and can be used for code completion, such as Codex [280], which powers Copilot [282]. The introduction of fill-in-the-middle capabilities is beneficial for real-world code completion use cases and has become common in recent code models such as InCoder [278], SantaCoder [175], StarCoder [174], and Code Llama [169]. Code Llama was also trained to follow instructions, generate code, and explain its functionalities.

While the multi-terabyte training corpora for these models contain some assembly, we believe that focused exploration of the value of LLMs in the domain of compilers will be of value to the community.

4.7 Discussion

We present the first steps towards LLMs for code optimization. We construct a model that can predict good optimization strategies for unseen LLVM-IR. LLMs can near-perfectly emulate many compiler optimizations and outperform prior approaches, but limitations exist. First, LLM operates on fixed input sizes that enable only small program fragments. The model has limited ability to perform arithmetic reasoning and predict the outcome of some optimizations. Finally, the speed of the optimization is limited to the speed of token generation.

4.7.1 Context Window

The main limitation of LLMs is the limited sequence length of inputs (context window). In this work, we target 2k-token context windows and split IRs into individual functions to maximize the amount of code we can fit into the context window. This is undesirable for several reasons. First, it limits the context available to the model when making optimization decisions; second, it prevents intra-function optimization; third, we cannot optimize code that does not fit within the context window. Figure 4.9 suggests larger programs have more interesting optimization opportunities.

Researchers are adopting ever-increasing context windows [312], but finite context windows remain a common concern with LLMs. As new techniques for handling long sequences continue to evolve, we plan to incorporate them and apply them to code optimization, e.g. Code Llama's variant of positional interpolation [313] which is based

on Rotary Position Embeddings period scaling [169] or recent length extrapolation techniques [314].

4.7.2 Math Reasoning and Logic

Compilers perform lots of arithmetic. Whenever possible, expressions are evaluated at compile time to minimize work at runtime and to expose further opportunities for optimization. We see examples of LLMs struggling with this type of reasoning, e.g., failed constant folding (Figure 4.13) and failed data-flow analysis (Figure 4.20).

We think the chain-of-thought approach [315], in which models are taught to decompose complex reasoning problems into incremental steps, will prove fruitful. We took the first step in this direction by breaking optimizations into individual passes in Section 4.5.3. We also plan to focus training on a curriculum of arithmetic and logic and train LLMs that use tools to compute intermediate results [316, 317].

4.7.3 Inference Speed

Compilers are fast. It takes two orders of magnitude more time for the model to generate a pass list than for the compiler to execute. While this is much faster than the autotuner it is trained on, it remains an overhead that may prove prohibitive for some applications. That is to say nothing of the difference in compute resources needed to evaluate compiler heuristics vs. a 7B-parameter LLM running on multiple GPUs.

In addition to aggressive batching and quantization [318], significant inference speedups can be achieved by specializing the vocabulary to a use case. For example, we can reduce entire subsequences of passes to single vocabulary elements using Byte Pair Encoding so that fewer tokens need to be generated at inference time.

Chapter 5

Feedback-directed Large Language Models for Compiler Optimization

5.1 Introduction

As shown in the previous chapter, LLMs showed advanced reasoning capabilities in learning and applying the best optimization sequences for a given LLVM IR. Our model achieves 3.01% better results than -Oz on 100k examples from test set in code size reduction. There is, however, still room for improvement since the model still didn't match the autotuner performance of 5.03%.

We want to extend this approach and increase performance on example models that couldn't match the autotuner's performance. To achieve this, we give a model a second chance by constructing feedback from the model's generation with the help of a compiler and asking the model to try again. This way, the model should get insight into its generation and hopefully improve. Additionally, we want to add stochastic sampling, enabling the model to generate multiple possible solutions and take the best one. By generating many potential solutions and being able to refine them by fixing generation-given feedback, we hope to boost performance closer to autotuner.

5.2 Motivation and Background

The performance of LLMs improves significantly when they are allowed to generate a series of reasoning steps to get to the final solution [315]. This behavior is particularly true for complex problems such as arithmetic, symbolic reasoning, and code generation. Additionally, increasing randomness and generating multiple solutions leads to superior performance. On the other hand, with increasing randomness, the model often generates incoherent solutions that impede their ability to reason correctly. Being able to get feedback on its generation and fix its errors could enable LLM to take one step further toward coming to a favorable solution.

We found that it is possible to derive a correlation between metrics available at inference time and model performance (Figure 5.1). There is a negative correlation between tgt_inst_cnt_error(C) (the difference between predicted target instruction count and instruction count of IR got by compiling predicted passes) and improvement_over_autotuner. In other words, a smaller error in predicting the target instruction count means that the model is more likely to find a good solution. This is also the case if tgt_IR_BLEU(C) (generated IR compared to compiled IR BLEU score) is high and the number of generated flags is large. Additionally, as one might expect, a positive correlation exists between the length of the generated optimization list and model performance.

To understand the relation between tgt_inst_cnt_error(C), tgt_IR_BLEU(C), and performance, we plot their distributions in Figure 5.2. When the model correctly predicts the target instruction count, the performance of the autotuner is also matched. This means that when we detect this case in the inference, we can stop prompting and accept generated optimization passes. Similarly, we can stop prompting if the compiled and generated IR are equal, which results in tgt_IR_BLEU(C) being 1.

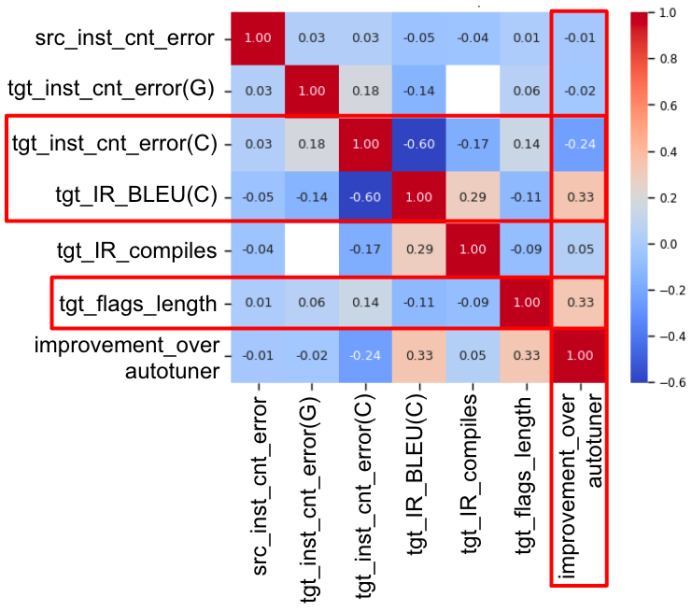


Figure 5.1 : Correlation heatmap of metrics available at inference time. Input and output prompts are described with prefixes (src, tgt). Instruction counts are abbreviated with inst_count. (G) stands for generated, while (C) stands for compiled.

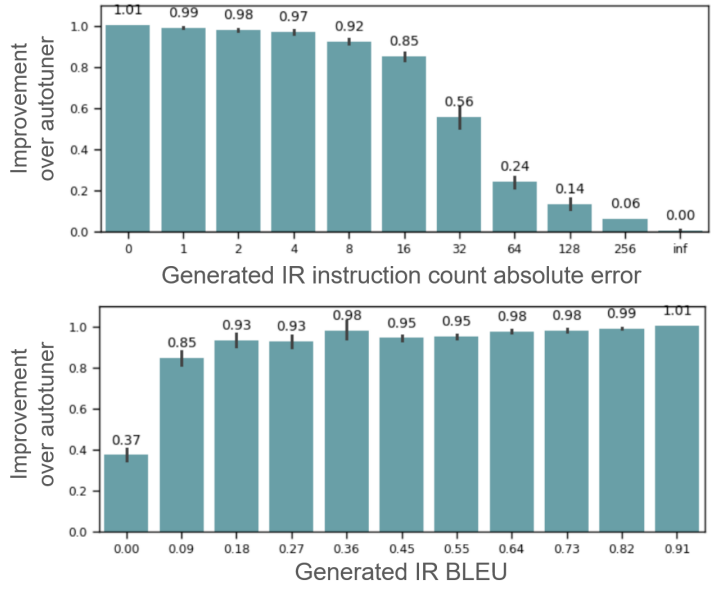


Figure 5.2 : Distribution of absolute error in predicting optimized IR instruction count and BLEU score with respect to performance compared to autotuner [7].

5.3 Feedback-directed LLMs

To use metrics correlated to performance, we propose a three-step process [7] shown in Figure 5.3. In the first step, the model starts from the prompt that contains only unoptimized IR and generates an optimization pass list, instruction count, and optimized IR itself. In the second step, we derive available metrics from generation with the help of the compiler and construct a feedback. The purpose of the feedback is to quantify the consistency of generation and to point out where the internal model of the LLM diverges from the actual compiled IR. In the third step, we provide feedback to the model and give it a second chance.

To construct feedback, we evaluate if the generated pass list is valid. Then, we compile source IR with the generated pass list, producing compiled IR. Next, we count the number of instructions of compiled IR and evaluate if the predicted source IR and optimized IR are correct. Since optimized IR could be derived from both generated IR and compiled IR, we save both metrics in the feedback. Additionally, we validate if the predicted IR is compilable, save the error message, if any, and calculate the Bleu score between the generated IR and compiled IR.

We compare three kinds of feedback (Figure 5.4). Short Feedback contains predictions and answers for metrics and error messages. Long Feedback contains all derivable metrics and extends Short Feedback by Compiled IR. Since Short and Long Feedback both contain the metrics from generated IR, they require an entire generation to be constructed. Fast Feedback avoids this by providing only metrics calculated from the pass list and instruction counts. This enables the model to stop generation early, terminating in just a few seconds, which is about 10x faster than other kinds of feedback.

When it comes to hardware efficiency, the process of appending feedback data is

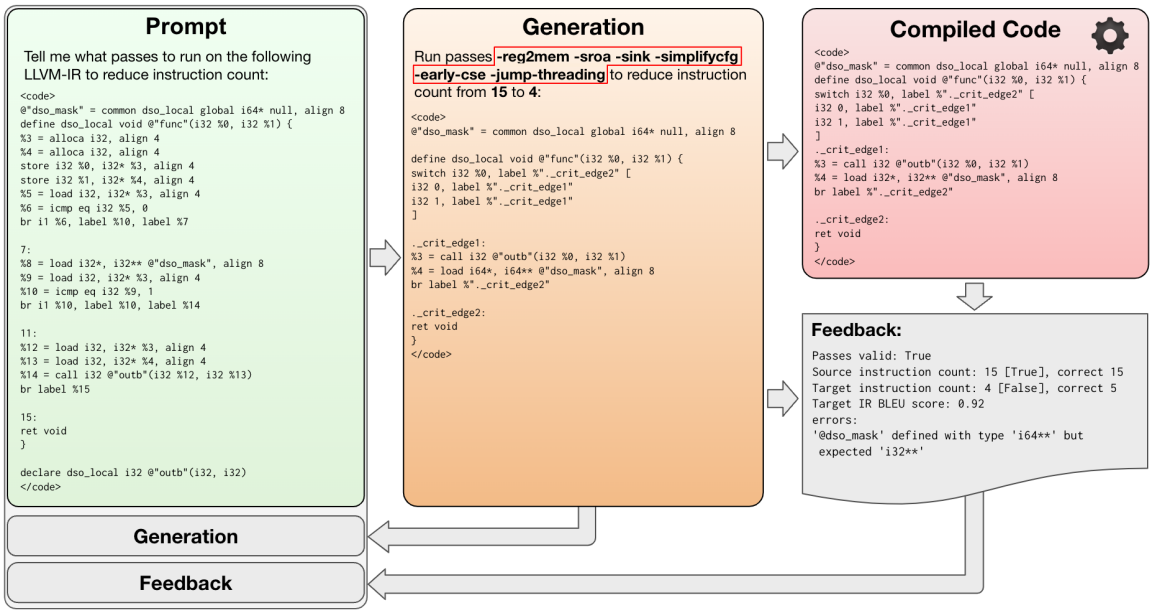


Figure 5.3 : Feedback-directed model. First, we ask LLM to optimize the instruction count of the given IR. LLM generates the best optimization passes, instruction counts for starting and generated IR and generated IR itself. Next, we compile the generated pass list and create feedback by checking if the generated pass list is valid, evaluating instruction counts, examining if the generated IR contains compilation errors, and calculating the BLEU score between the generated IR and the compiled IR. If some feedback parameters are problematic, we extend the original prompt with the generation, compiled code, and feedback and ask it to try again [7].

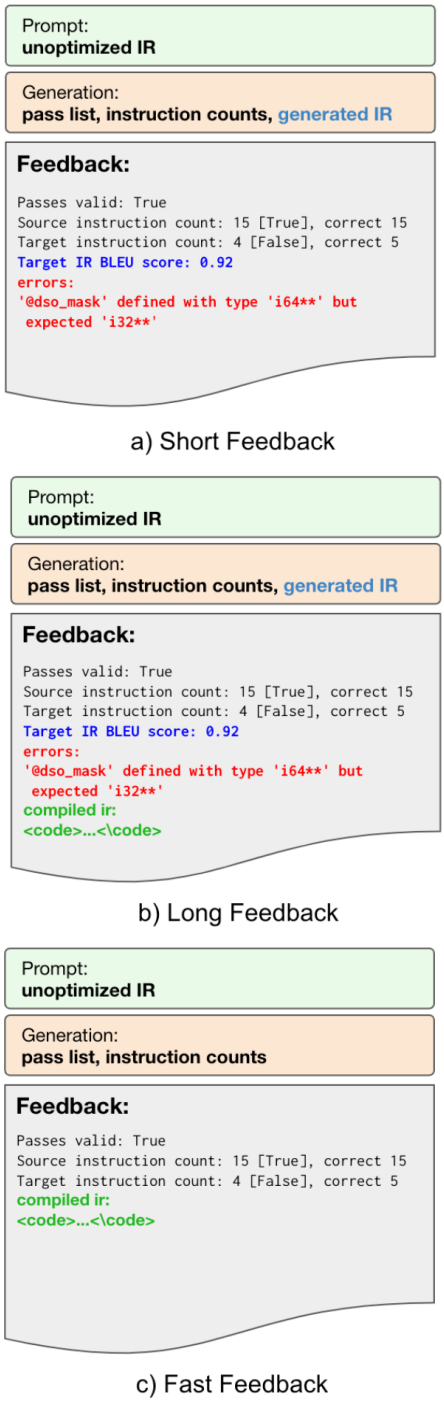


Figure 5.4 : Prompt structure of Feedback models. Short Feedback is the smallest in size and extends the prompt with just calculated metrics and error messages. Long Feedback contains the most information including compiled IR. Fast Feedback is the fastest to generate since it doesn't need the generation of IR to be calculated [7].

highly efficient. When the model generates the last output token, the GPU memory already contains prompt and generation. Feedback would just be written in already allocated GPU memory, and the model would be ready for evaluation a second time.

Structuring the feedback task after the prompt and generation has one additional benefit. It reinforces learning of optimization tasks without feedback as well. This happens because the probability of one token depends only on the previous tokens. Since we are appending a feedback task after the optimization task, it will not influence it. This way, we can use the same model for both optimization without feedback and with feedback.

Combining the Feedback approach with sampling can be an effective way of tuning applications. By increasing the temperature in LLM generation, the model creates multiple optimization strategies. Since this process is stochastic, there is a higher chance of some errors in the generation. Learning the model and how to correct itself could enable it to develop the idea further by fixing itself until it comes to a promising solution.

5.4 Training Methodology

We train a 7B-parameter model with LLaMa 2 architecture [287] for each of the Feedback forms. As the starting point for training, we use the best checkpoint from Chapter 4, which only predicts the best optimization passes for the given IR. We use the same Byte Pair Encoding [294] tokenizer and model architecture that includes 32 attention heads, 4,096 hidden dimensions, and 32 layers for a total of 7B parameters.

5.4.1 Datasets

We construct a training data set for each of the feedback forms. We evaluate 1M training examples for each form and construct the new dataset that includes the feedback. We use 100k test examples from Figure 4.3 to find the best model. Additionally, we extract half of the examples from the test set to serve as a validation set.

The prompt will have the structure described in Figure 5.4 for all feedback forms. For expected generation, we keep the same format as the original work with an addition of the first line that indicates if the model is sure in its generation. Model outputs *"I am sure!"* if the model correctly predicted the target instruction count, which is a strong indication that the model matched the performance of the autotuner. Otherwise, the model outputs *"Let me try again."*.

5.4.2 Training

We trained all our models for 20,000 steps, with 64 A100 for about 60 GPU days. We use the AdamW optimizer [295] with β_1 and β_2 values of 0.9 and 0.95. We use a cosine learning rate schedule with 1,000 warm-up steps, a peak learning rate of $1e{-}5$, and a final learning rate of 1/10th of the peak.

We used a batch size of 256, and each batch contains 786,432 tokens for Short and Fast Feedback and 1M tokens for Long Feedback, for a total of 16B and 21B tokens, respectively. The full training of 20,000 steps made 5.12 iterations over the training corpus.

5.5 Experiments

In the evaluation, we answer the following questions:

- How do the feedback models compare to the original in Task Optimize and Task Feedback?
- How does the best feedback model perform when sampling is enabled?
- Can we use the feedback model to generate feedback and repair the current solution iteratively?

We found that the feedback model keeps the ability to optimize IR even without feedback. When allowed to apply two inferences, it can outperform the original model by 0.53%, closing the gap to the autotuner by 10%. On the other hand, when the sampling is enabled, we show that the original model achieves up to 98% of the autotuner improvement over -Oz given 100 samples. We evaluate three sampling strategies for the feedback model and show that they all fail to match the sampling of the original model. Finally, we compare the performance of the iterative feedback model with the original model given the same amount of computation per sample. We show that the original model outperforms the feedback model with two or more samples and a temperature higher than 0.4.

5.5.1 How does the feedback model compare to the original in Task Optimize and Task Feedback?

We compare all three feedback models with the original on the Task Optimize and Task Feedback (Figure 5.5). In Task Optimize, the input prompt consists only of the input IR, while in Task Feedback, each model will append the input prompt with

the feedback they got from the previous generation in the format defined in Figure 5.4. Additionally, we show performance on: 1) all examples; 2) examples where the autotuner found a non-Oz optimization pass; 3) examples where the original model was worse than the autotuner, and 4) examples where the original model mispredicted the instruction count. Furthermore, we show the performance of the model combined with *-Oz*.

All of the feedback models perform similarly on average to the original on Task Optimize even without being trained on that Task explicitly in the feedback fine-tuning. Moreover, the feedback models even improved the performance for examples where the original model performed worse than the autotuner by 0.6% for Fast Feedback. This is because we add extra information to the input, enabling the model to discriminate complex examples and learn them more easily.

In Figure 5.5, we feed the output from Task Optimize to each feedback model and apply Task Feedback while keeping the results from Task Optimize for the original model. All the feedback models improve the performance of the original model by 0.19% with Short Feedback, 0.4% with Long Feedback, and 0.53% for Fast Feedback. Most of the improvement comes from the examples where the original model performed worse than the autotuner and the examples where the model mispredicted the generated instruction count. Here, the Fast Feedback model outperforms the original model by 1.48% and 1.07%, respectively.

Interestingly, the Fast Feedback model performs better than Long Feedback despite using a subset of information in the input prompt. In this case, adding generated IR to the input prompt adds noise and confuses the model. Since the Fast Feedback model doesn't need to generate IR to create feedback, we can iterate much faster. We use the Fast Feedback model for further evaluations for the following experiments.

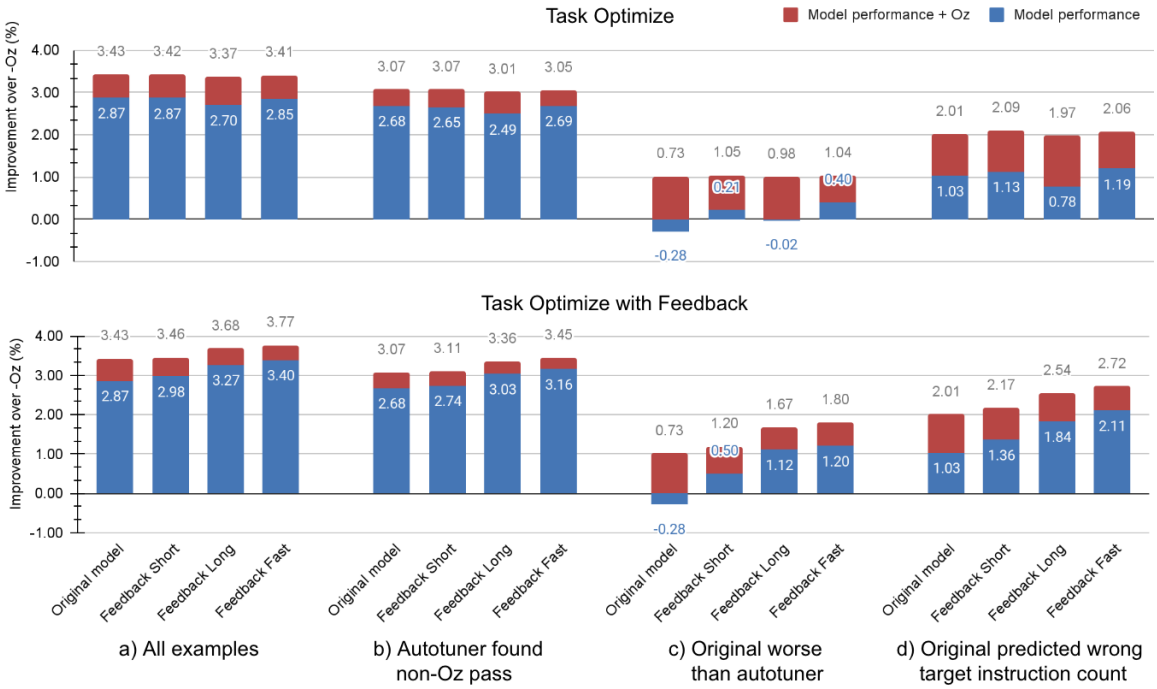


Figure 5.5 : Comparison of the original and feedback models in reducing instruction count. The upper figure shows the performance on Task Optimize. The lower figure shows the performance on Task Feedback, where each model uses their format for feedback. Horizontally, we show the performance on all examples: examples where the autotuner's best pass is non-Oz, the original model was worse than the autotuner, and the original model mispredicted target instruction count. All the models keep the ability to perform Task Optimize while improving the performance when feedback is provided [7].

5.5.2 How does the feedback model achieve when sampling is enabled?

The performance of the LLMs could be further improved with sampling. Sampling is a process of generating new answers on the same prompt by choosing the next token from the probability distribution of tokens rather than just selecting the best one. This way, the model can generate solutions that could match the prompt better than the original answer.

The common sampling technique is Temperature Sampling, which idea originates from Boltzman machines [319], but is recently applied to text generation [320–322]. In Temperature Sampling, we divide each next token probability with the constant T, which makes distribution sharper if T is closer to 0 or more uniform as we increase T. The parameter T is usually in the range (0, 2), where we refer to T=0 as the original (greedy) sampling, while T closer to 2 often produces incoherent results. We conduct a comprehensive sampling analysis in the Chapter 6.

In our work, we use Nucleus Sampling [9], the most common sampling technique for text generation. Nucleus Sampling avoids text degeneration by truncating the unreliable tail of the probability distribution, sampling from the dynamic nucleus of tokens containing the vast majority of the probability mass. As suggested in the original paper [9], we sample only tokens whose cumulative probability mass exceeds *top-p=0.95*.

Original model sampling

We evaluate Nucleus Sampling on randomly selected 50k unseen test examples (Figure 5.6). We chose the best result from the given sample number for each example and averaged it across 50k examples. Additionally, we sweep the temperature parameter in the range (0, 1.6) with the step of 0.2.

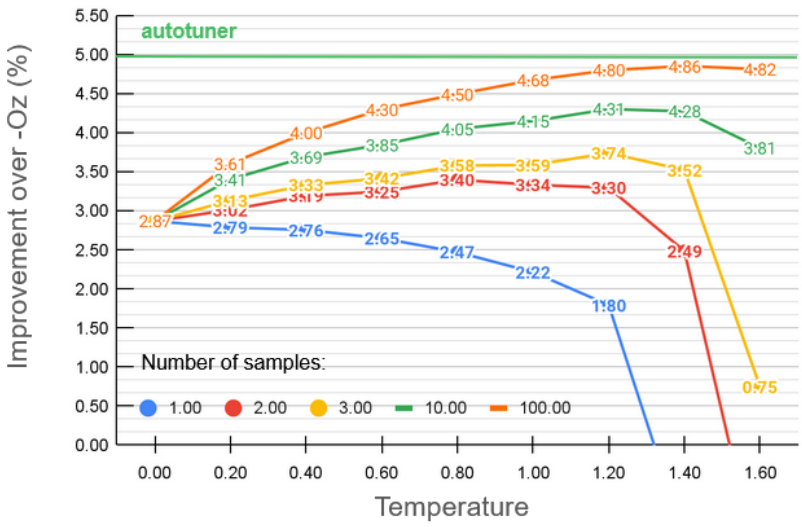


Figure 5.6 : Sampling diagrams of the original model for 50k unseen randomly selected test examples [7].

For temperature 0, we use greedy decoding, which always chooses the token with the highest probability, resulting in a performance of 2.87% above -Oz independently of the number of samples. The performance increases with the temperature increase, especially for runs with more than 1 sample, while it decreases for one available sample. This is expected since, with multiple samples, the model has more room to make errors while discovering potentially better optimization passes.

Given 100 samples, our model achieves a remarkable 4.86% improvement over -Oz on a temperature of 1.4. This is 98% of the improvement that the autotuner achieves compared to -Oz. With ten samples, the original model achieves about 86% of the autotuner, which is a significant win compared to 57% of the temperature 0. This means the model learns to construct reasonable solutions, which are reachable by smarter sampling techniques.

Feedback model sampling

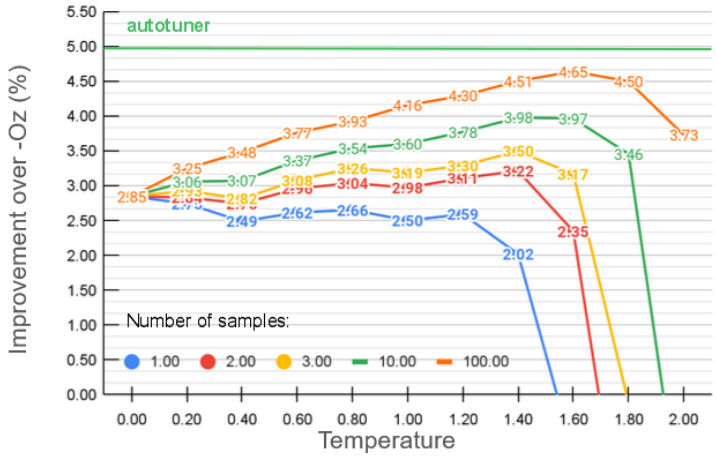
Can we improve the sampling performance of the original model with the feedback mechanism? We explore the idea behind feedback models in sampling. Since the feedback models improve generation performance on temperature 0, can we apply the same technique and improve each potentially wrong generation during sampling?

To check this hypothesis, we evaluate the Fast Feedback model in 3 additional experiments:

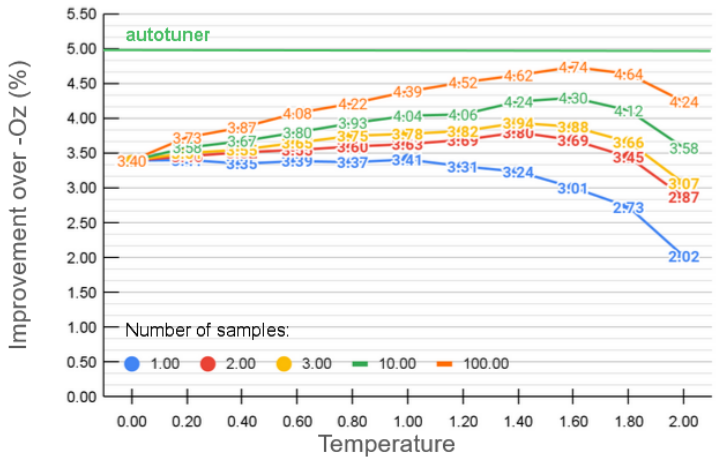
1. Task Optimize with Nucleus Sampling with given temperature T.
2. Task Feedback with Nucleus Sampling with T=0, starting from (1).
3. Task Feedback with Nucleus Sampling with given temperature, starting from (1) with T=0.

In the first experiment, we sample the Fast Feedback model in direct generation of optimization flags (Task Optimize), from which we generate feedback data for the following two experiments. In the second experiment, we set the temperature to 0 and asked the model to correct the previously generated flags on a given temperature. This way, the model gives it the best guess to correct previous generations. In contrast, in the third experiment, the model starts from the best guess for Task Optimize and provides up to 100 samples to fix it.

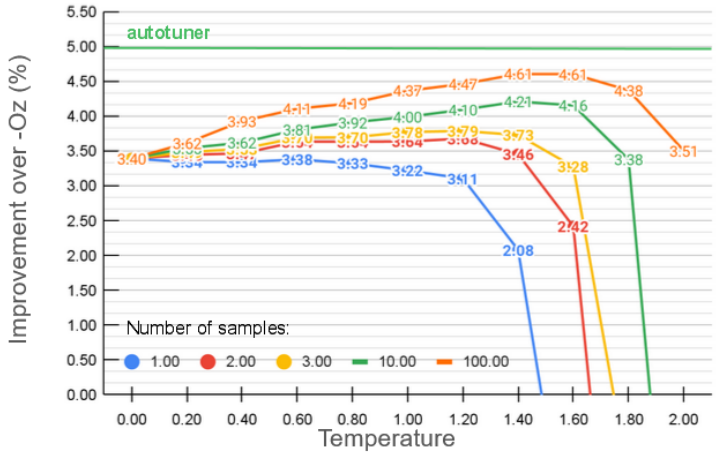
In Figure 5.7, we can see the results. As expected, the sampling curves of the Fast Feedback resemble the original model but perform slightly worse since it was not trained on Task Optimize. Interestingly, the Fast Feedback model achieves the peak performances on higher temperatures for each number of samples, reaching 4.65% over -Oz on temperature 1.6 with 100 samples.



a) Fast Feedback model on Task Optimize T=X



b) Fast Feedback model on Task Optimize T=X, Task Feedback T=0



c) Fast Feedback model on Task Optimize T=0, Task Feedback T=X

Figure 5.7 : Sampling diagrams of the feedback models. T=X means that temperature could be any value [7].

In the second experiment (Figure 5.7(b)), the Fast Feedback model achieves up to 4.74% improvement over -Oz on the temperature 1.6 on Task Feedback, increasing the performance of previous Task Optimize. It outperforms the original model for up to 3 samples but doesn't keep the advantage for 10 or more samples. Similarly, it outperforms the original model for temperatures 0 and 0.2 but not higher temperatures. This is because we don't train the Fast Feedback model to correct the generation of Task Optimize when the temperature is on, which causes the generations to become significantly different.

In the third experiment (Figure 5.7(c)), the Fast Feedback model starts from generations with temperature 0, but increasing the temperature for Task Feedback fails to provide better results. Similar to the previous example, the lower number of samples (1, 2, 3) and temperature in the range [0, 2] achieves higher performance than the original model, but not for 10 or more samples.

From these results, we can conclude that sampling the Fast Feedback model could slightly outperform the original model for samples smaller than 3, but not more. Additionally, training the Fast Feedback model on generations with non-zero temperatures could increase performance. Unfortunately, such an approach introduces significant complexity to the training process for non-significant improvement.

5.5.3 Can we use the feedback model to generate feedback and repair the current solution iteratively?

In this section, we evaluate the capability of the fast feedback model to iterate based on previous solutions and compare it to the sampling of the original model with the same number of inferences. First, the Fast Feedback model applies Task Optimize, which generates feedback. Then, the model applies Task Feedback iteratively by

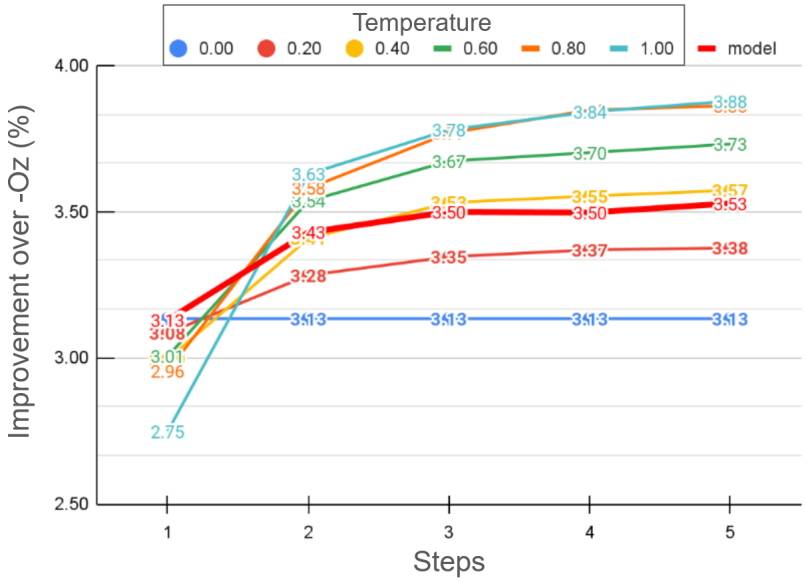


Figure 5.8 : Comparison of the iterative approach (model) versus the sampling of the original model with the same amount of computation. In each step, the Fast Feedback model generates feedback for the next step, applying Task Optimize in the first step and Task Feedback afterward. Once the model outputs "I am sure!" we stop. We allow the same number of generations for the original model [7].

using the feedback of the previous Task and generates the feedback for the next step for a total of 5 steps. After every generation, we check if the Fast Feedback model generated *"I am sure!"* and use that as the final result. For the original model, we applied Task Optimize and sampled each example as many times as we made steps with the feedback model. We use temperatures in the range [0,1] and show the cumulative performance of each approach in Figure 5.8.

The Fast Feedback model performs similarly to the original model with temperature 0 on Task Optimize while making the most of performance improvements in the second step (Task Feedback) and slowly increasing by iterating on Task Feedback,

achieving performance similar to the original model with temperature 0.4. The original model with temperature 1 starts with the lowest performance but outperforms all other approaches from step 2. This result demonstrates that sampling is more powerful than iteration for our problem and should be used instead.

5.6 Related Work

Large language models are able to generate code and documentation [169, 174, 175, 276–279], translate code between programming languages [181, 281, 293], write unit-tests [177, 178, 180], as well as detect and fix bugs [310, 323].

The availability of various open-source code datasets [300, 324] and the accessibility of platforms like GitHub have enabled models such as CodeLlama [169], ChatGPT [170], and Codex [280] to elevate their coding capabilities. However, it's important to note that these models were not explicitly designed for code optimization. For instance, ChatGPT can perform minor optimizations, like tagging variables for storage in registers, and even attempt more substantial improvements, such as vectorization. Nevertheless, it encounters confusion and makes errors, leading to incorrect code outcomes.

On the other hand, models such as AlphaCode [168] generate a performant solution by optimizing code on the source level. AlphaCode is fine-tuned on competitive programming problems from the Codeforces platform, using 715.1 GB of code data from GitHub for pretraining. Furthermore, it generates a large corpus of potential solutions from which it chooses the 10 best solutions by implementing sophisticated clustering and filtering mechanisms. Similar to Alphacode, we demonstrate the power of sampling while targeting compiler-level optimizations instead of the optimizations on the source code.

When it comes to fundamental LLM improvements, Wei et al. [315] showed that significant improvement of the LLM can be obtained by splitting answers in step by step manner for models with more than 10B parameters. Brown et al. showed that a few-shot prompting [75] based on extending prompt with similar (question, answer) pairs additionally increases the model's performance. Yang et al. [325] extend this approach further by iteratively adding generated solutions and their evaluations of the original prompt, together with a few-shot prompting. Our approach provides more informative feedback based on inference time evaluation of model generations, including validation of generated pass list, evaluation of predicted instruction counts, and optimized IR.

Finally, the problem of compiler pass ordering has been explored for many decades [91, 288,304]. In recent years, machine learning has become an avenue for this optimization task, with various approaches proposed [1,105,283,284,301,305]. It's noteworthy that the implementation of machine learning in compiler technology extends beyond just optimizing pass order and includes a wide range of other challenges [159,161,306–308]. We extend these efforts further by pioneering the use of large language models in compiler optimization.

5.7 Discussion

Sampling has exceptional potential to improve the original model's performance. With simple Temperature Sampling, the model reaches 98% of the autotuner's performance with 100 samples. To improve this performance further, we provide compiler feedback to the original model and give it another chance. We explore three formats of feedback with various degrees of information. All feedback forms outperform the original model on temperature 0 by 0.11%, 0.4%, and 0.53%.

Next, we explore the sampling properties of the Fast Feedback and compare it to the original model. We evaluated the feedback model on task: 1) Generate LLVM flags given temperature sampling, 2) Generate LLVM flags using temperature sampling and use feedback with temperature 0, and 3) Generate LLVM flags with temperature 0 and use feedback with temperature sampling. In all these cases, the Fast Feedback model failed to improve the performance of the original model when there were more than 10 samples. This indicates that either the Fast Feedback model doesn't know how to deal with generations at higher temperatures or always generates the same output.

To mitigate this problem, we could generate a dataset by sampling the original model and train the Fast Feedback model on these data. Additionally, our model contains only 7B parameters, and scaling it up might increase the performance further. Models larger than 10B parameters perform better when using chain-of-thought prompting [315], which provides additional information similar to our approach.

Another approach is to discard the feedback component and implement a smarter sampling heuristic for the original model. We can use beam search and similar techniques when sampling the tokens. AlphaCode [168], for example, generates many possible solutions and then uses clustering and filtering techniques to select the best answers. Developing methods in this area could be more generally applicable in the LLM community.

The next chapter describes a novel Priority Sampling method that increases sampling efficiency and provides better generation control.

Chapter 6

Sampling of Large Language Models for Compiler Optimization

6.1 Introduction

The performance of LLMs could be improved by generating an ensemble of diverse solutions from which we evaluate and choose the best. This is usually done by increasing the entropy of generation [326,327], or expanding the search tree [10,11,328,329]. Sampling also enables us to better understand an LLM's capacity for a given task and the range of possible solutions. This is particularly important in code generation, where generating a variety of responses can be valuable in exploring different implementation ideas.

Current sampling approaches have few major problems. Temperature-based sampling [326, 327] requires a significant amount of computation to find the optimal temperature. The optimal temperature may also depend on the context, which requires additional evaluations. Once we set the temperature, sampling often produces many duplicates and semantically meaningless answers, wasting available samples.

To motivate our approach, we show the average number of unique samples generated by Nucleus Sampling compared to Priority Sampling (Figure 6.1). On 50K test examples, Nucleus Sampling generates less than five unique examples on average for 100 samples, while Priority Sampling generates 100. This is because Nucleus Sampling

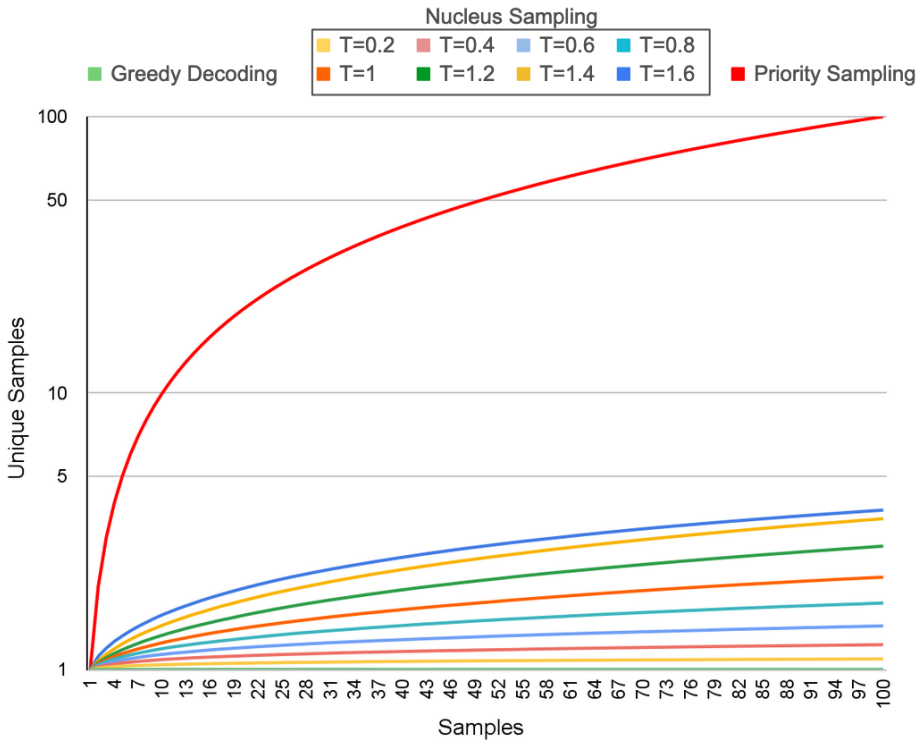


Figure 6.1 : Average number of unique samples generated from 50K unseen test programs. Priority sampling produces a higher ratio of unique samples than nucleus sampling [8]

either chooses the high probability generation repeatedly or generates non-coherent output, which we don't count as a meaningful unique sample.

We present Priority Sampling [8], a deterministic sampling technique that guarantees unique samples for which the model has the highest confidence. Furthermore, we guarantee that produced samples will adhere to the regular expression (inspired by Willard [330]), which is particularly important for code generation and optimization.

We evaluate Priority Sampling on optimizing LLVM optimization passes [6] in which the model is trained to predict optimizations found by the long-running autotuner. Priority Sampling outperforms Nucleus Sampling [326] for any number of

samples, reaches 91% of the autotuner improvement over -Oz in just five samples, and even outperforms the autotuner used to generate labels for finetuning the original model with 30 samples (Figure 6.3).

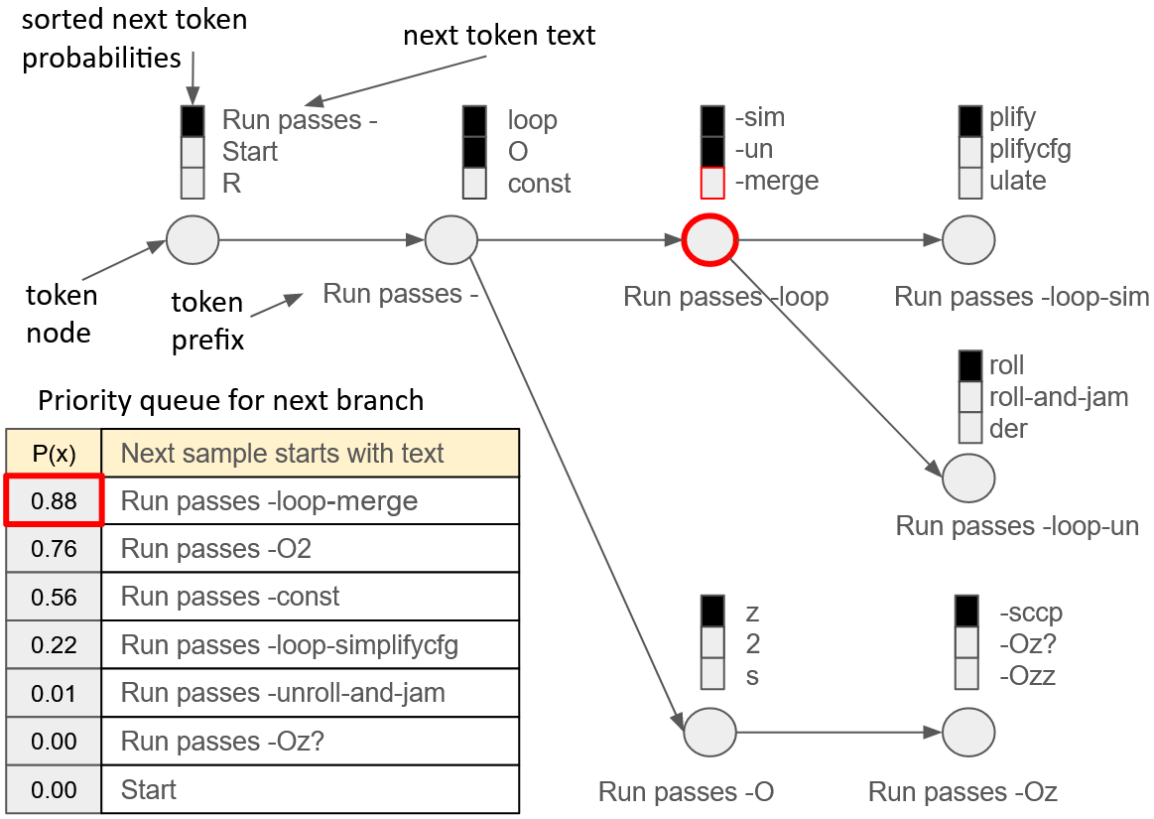
6.2 Priority Sampling of Large Language Models

At a high level, Priority Sampling works by augmenting a search tree and determining which unexplored path to expand next based on the model's highest confidence. Our idea is simple. Always focus the search towards the most interesting direction based on previous samples rather than determining this in advance. Additionally, avoid sample repetition, which decreases sampling power.

We sketch the algorithm for Priority Sampling in Figure 6.2. The model is equivalent to Greedy Decoding for the first sample but with an important addition. Each generation saves top K alternative tokens with their predecessors in the priority queue with the token's probability as a metric. Once a sample is generated, we can quickly find what token to expand with the next sample by poping with the token with the highest probability from the queue and expanding the search tree from there. Since we add unexpanded tokens to the queue only once, each new sample will be unique. Additionally, we need the same number of inferences as the number of tokens generated in the search tree.

Going into more detail, the Algorithm 1 defines priority_queue and token_mask, which will determine the best tokens prefix-sequence to expand and steer token generation to that point. Since we know the number of samples we generate, we can fix the length of priority_queue and set token_mask length to the generation length of the model.

With two *for* loops, we iterate through sample space and perform token generation



Priority queue for next branch

P(x)	Next sample starts with text
0.88	Run passes -loop-merge
0.76	Run passes -O2
0.56	Run passes -const
0.22	Run passes -loop-simplifycfg
0.01	Run passes -unroll-and-jam
0.00	Run passes -Oz?
0.00	Start

Figure 6.2 : Priority Sampling tree expansion. Each node contains a token generated by inference and the probabilities of the next potential tokens. In the first sample, we create a branch from the root to the end-of-sequence (EOS) token and put all valid potential tokens with their probabilities in the priority queue. For every next step, branch the token with the highest probability and generate that branch until the EOS [8].

Algorithm 1 Priority Sampling

```
priority_queue ← queue()
token_mask ← list()
sample_tokens ← list()
for sample : range(samples) do
    generated_tokens = list()
    for pos : range(generation_length) do
        if pos < len(token_mask) then
            next_token ← token_mask[pos]
        else
            logits ← inference(generated_tokens)
            best_tokens ← choose_best_tokens(logits, generated_tokens, regex)
            next_probability, next_token ← best_tokens[0]
            for probability, token : best_tokens[1:] do
                priority_queue.push(probability, generated_tokens + [token])
            end for
        end if
        generated_tokens.append(next_token)
    end for
    sample_tokens.append(generated_tokens)
    token_mask ← priority_queue.pop()
end for
return sample_tokens
```

for each sample while keeping track of the previously generated tokens for a given sample. To determine the next token, we either follow the sequence from token_mask until we reach the branching point or expand the search tree by applying the inference.

With inference, we get the probability distribution of tokens, from which we choose K tokens with the highest probabilities. An important addition here is that we exclude all tokens that don't satisfy the regular expression we define when combined with previous tokens. This can be done in constant time by using a finite state machine as described in the previous work [330]. This technique enables us to steer generation only toward legal format, which is particularly useful for code generation.

Once we select the best tokens, we expand the search tree directly with the best token while putting the remaining tokens in the queue. We repeat this until we finalize the generation of the current sample. After all potential tokens for expansions are saved to the queue, we update *token_mask* with the token prefix with the highest probability. Finally, we use the token prefix to locate the node that needs to be expanded and start the generation of the new sample from there.

Priority Sampling has the algorithmic complexity of *O(T*(inference + Klog(V)))*, where T is the number of generated tokens, K is the number of top-k samples we consider, and V is vocabulary size. In practice, this is similar to Nucleus Sampling since the cost of inference is much higher than *Klog(V)*, and Priority Sampling reuses inferences for the samples with the same prefix.

Additionally, the memory requirements are significantly reduced by keeping the size of the priority queue equal to the number of samples we generate. This way, we avoid saving the probabilities of all tokens in the vocabulary for each node while ensuring that there will be enough candidates for branching the search tree.

6.3 Experiments

We evaluate the Priority Sampling technique on generating efficient LLVM optimization passes with LLM that reduces code size [6]. First, we train the 7B parameter model with Llama2 architecture for 30,000 steps on 64 V100s for a total training time of 620 GPU days. The training dataset consists of 1M LLVM IR labeled with the LLVM optimization sequence found by autotuner. The autotuner spends 13 minutes exploring 37,424 optimization passes on average to generate a label for each example. Finally, we autotune 50K unseen test examples for 13 minutes for a total improvement of 4.98% over -Oz.

To evaluate the effectiveness of Priority Sampling, we compare it to Random Sampling, Greedy Decoding, and Nucleus Sampling for 100 steps. Random Sampling evaluates 100 random optimization passes and calculates the best optimization pass so far for each sample. Greedy Decoding generates an optimization sequence by deterministically predicting the next token with the highest probability. For the Nucleus Sampling, we evaluate the model for temperature in the range {0.2, 0.4, 0.6, 0.8, 1, 1.2, 1.4, 1.6}. For our problem and model architecture, temperature 1.2 is the most effective under 20 samples, while temperature 1.4 is the most effective for more than 20 samples.

We present the comparisons in Figure 6.3. Priority Sampling outperforms Random Sampling, Greedy Decoding, and Nucleus Sampling for any number of samples in reducing code size. Moreover, Priority Sampling is much more sample efficient than Nucleus Sampling achieving even the performance of the autotuner in 30 steps. Increasing the performance of the original model from 2.87% to over 5% with just 30 samples means that a significant part of knowledge is accessible by expanding the search tree wisely.

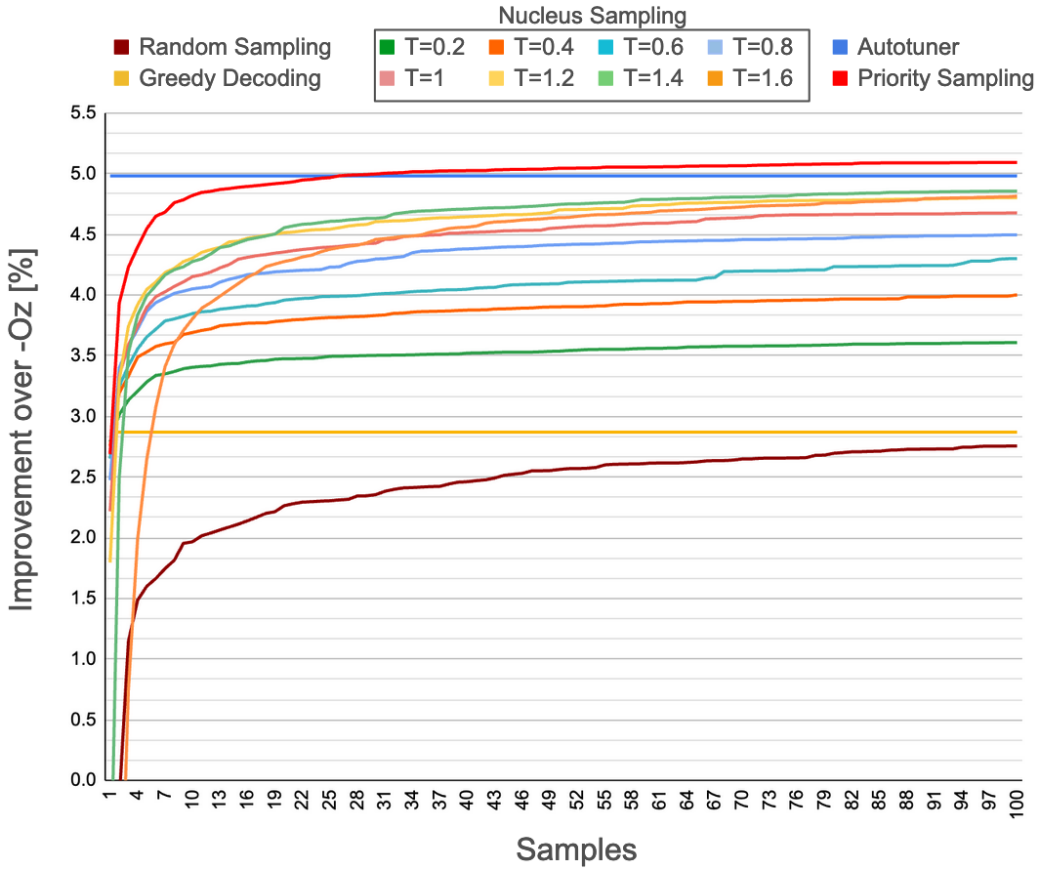


Figure 6.3 : Average improvement in code size over -Oz optimization on 50k unseen test examples. Autotuner spends 760s optimizing each example and sets the labels for LLM fine-tuning [6]. Greedy Decoding, Nucleus Sampling, and Priority Sampling use the fine-tuned model. Random Sampling selects 100 random flags for each sample. Priority Sampling outperforms all previous methods, including autotuner, which was used for labeling [8].

This is an astonishing result since the model was trained to mimic the autotuner's behavior, not to outperform it. Since the autotuner operates on the complex set of LLVM optimizations tied to the input program's structure, our model seems to generalize from these patterns and combines them in a novel way on the unseen programs, resulting in higher performance.

6.4 Additional Experiments

For ablation (Table 6.1), we show how the performance of Priority Sampling changes if 1) we don't use regular expression filtering, 2) use the geometric mean of probabilities of previously generated tokens as the metric for the priority queue, and 3) constrain the expansion for each node to 3 and 5.

If we don't enforce regular expression generation, the generated sampling tree will have higher probabilities, but generated samples could be invalid. To address this, we apply an additional pass that removes all invalid optimization passes and defaults to -Oz if all passes are illegal. This technique is beneficial for 1 and 100 samples while enforcing regular expressions outperforms slightly non-constrained versions otherwise.

Next, we evaluate using the geometric mean as the metric for the priority_queue. This could be an interesting idea since the probability of the next token is highly biased with few previous tokens. For example, prefix *-mem2* will put high probability to token *reg*, independently if *-mem2reg* is a good optimization to apply. On the other hand, calculating the geometric mean of previous token probabilities doubles memory requirements since we need to store probability with each generated token. We found that this doesn't significantly influence the final performance.

Finally, we evaluate the method's performance when the branching size constrains. This idea focuses on increasing sample diversity and avoiding the generation of many

	Improvement over -Oz [%]					
Method	Sample 1	Sample 3	Sample 5	Sample 10	Sample 30	Sample 100
Random Sampling	-12.56%	1.15%	1.60%	1.97%	2.36%	2.76%
Temp0	2.87%	-	-	-	-	-
Temp1.2	1.80%	3.74%	4.05%	4.31%	4.61%	4.80%
Temp1.4	-1.19%	3.52%	3.99%	4.28%	4.63%	4.86%
Temp1.6	-10.06%	0.75%	2.65%	3.81%	4.46%	4.82%
Autotuner	4.98%					
Priority Sampling (PS)	2.69%	**4.23%**	4.55%	4.82%	**5.00%**	5.09%
PS (no regex)	**3.17%**	4.18%	4.41%	4.64%	4.93%	**5.12%**
PS (max_branch 3)	2.62%	4.22%	4.56%	4.83%	4.99%	5.09%
PS (max_branch 5)	2.62%	4.22%	**4.61%**	**4.85%**	4.99%	5.09%
PS geometric (PSG)	2.68%	4.17%	4.45%	4.75%	4.96%	5.07%
PSG (max_branch 3)	2.62%	4.17%	4.52%	4.77%	4.98%	5.11%
PSG (max_branch 5)	2.62%	4.17%	4.56%	4.80%	4.98%	**5.12%**

Table 6.1 : Experimental results and ablation experiments of Priority Sampling. Evaluation includes the improvement of Random Sampling, Nucleus Sampling, and Autotuner over the compiler (default -Oz optimization). Ablation evaluates the use of the regular expression, constraining branching factor, and using the geometric mean as the priority metric in Priority Sampling [8].

nodes with the same prefixes. For a given prefix, the first few samples should be enough to finalize the optimization strategy, while we should use other samples to explore the alternatives. Our results suggest that there is some benefit of constraining the branching factor to 5 for our problem, but not significant.

6.5 Related Work

By default, LLMs auto-regressively use greedy decoding to choose the token with the highest probability. Since the inference is deterministic, it generates the same answer for the given prompt. This kind of generation could be inconvenient for open-ended tasks, where multiple ways exist to come to a favorable solution. Even more, the lack of diversity limits the quality of the generation compared to sampling methods since alternative generations often perform better than greedy decoding.

So, how do we sample LLM? In sampling methods, we choose the next token from the probability distribution of all possible tokens instead of the token with the highest probability. In most cases (when the input is similar to the training distribution), the probability mass is concentrated around a few tokens, while the majority of the tokens have a probability close to 0. Many sampling methods focus on reshaping this distribution, preferring certain kinds of tokens or making distribution more uniform.

The simplest way to control the shape of the next token probabilities is to use Temperature Sampling [321,331], which introduces parameter T in the range *(0, inf)*, which divides each probability from distribution. For $T < 1$, the values of probabilities will increase, effectively sharpening the distribution while otherwise becoming more uniform. Finding the right temperature depends on the application and presents a tradeoff between conservative answers that could be repetitive or liberal answers that could become incomprehension.

Top-k Sampling narrows the choice of the next word to the top k most probable tokens [321, 331, 332]. This approach often results in grammatically correct text but fails to maintain fluency for longer answers. Even though language models do generally assign high scores to well-formed text, the highest scores for longer texts are often generic, repetitive, and awkward [9]. Additionally, top k choices often include tokens with a probability close to 0, which could result in an incoherent token. This problem is even more pronounced in Temperature Sampling.

To solve this problem, Nucleus Sampling [9] eliminates the low-probability tail of the distribution and preserves diversity by sampling from tokens whose sum is larger than *top-p=0.95* probability. Rather than relying on fixed top-k choices or using a temperature parameter to control the shape of the distribution, Nucleus Sampling suppresses the unreliable tail of a distribution, expanding and contracting the candidate pool dynamically. As a result, this approach generates more natural human-written text with less repetition.

It is interesting to note that both human-written language and Nucleus Sampling have higher Perplexity (that corresponds to entropy of generation) than Top-k Sampling while being more coherent (Figure 6.4). The authors of [9] explain this by an intrinsic property of the human language to optimize against stating the obvious, discouraging long sequences of predictable words. This property doesn't necessarily hold for LLMs when generating assembly code rather than human language.

Another big problem with sampling is diversity. For each previous model, there is no way to guarantee that each sample is going to be different from the previous. This is particularly a problem with shorter generations, such as a sequence of LLVM flags, in which many samples are the same as greedy decoding.

The first work that addressed this problem is Diverse Beam Search (DBS) [333].

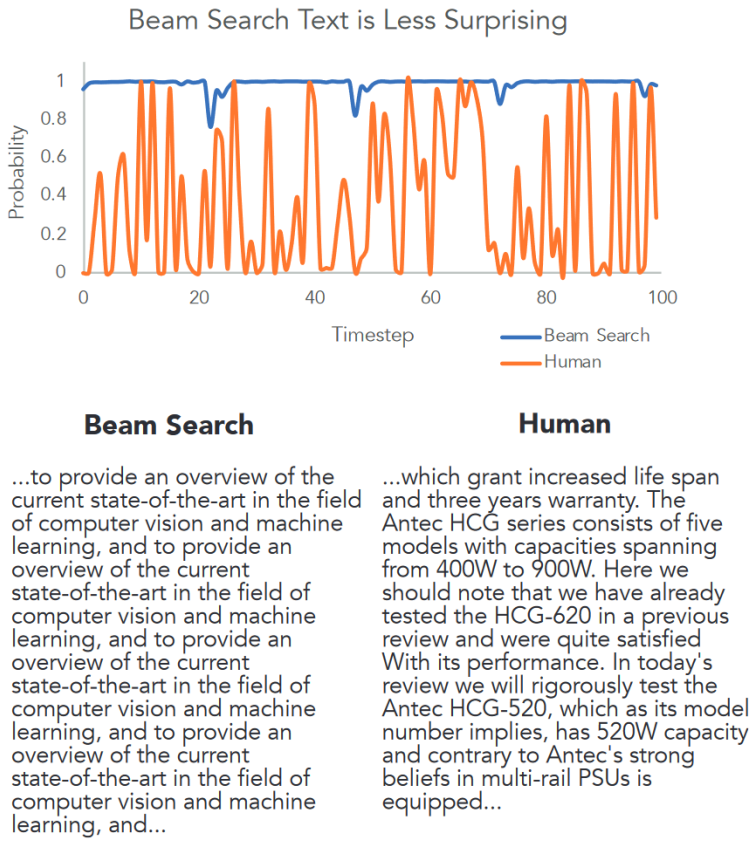


Figure 6.4 : Figure from [9]. The probability assigned to tokens generated by Beam Search and humans, given the same context. Note the increased variance that characterizes human text, in contrast with the endless repetition of text generated by Beam Search.

Similar to Beam Search, DBS decodes diverse lists by dividing the beam budget into groups and enforcing diversity between groups of beams. To enforce diversity, DBS optimizes for both sequence likelihood under the model and a dissimilarity term based on Diverse M-Best MAP inference [334]. With this approach, DBS creates more diversity than traditional beam search without extra computation in inference.

Instead of shaping reward function based on diversity, Stochastic Beam Search

(SBS) [329] uses Gumbel-Max trick [335] to sample top-k tokens with the highest probabilities in differentiable manner. The Gumbel-Max Trick involves adding Gumbel-distributed noise to the logits (likelihood scores), after which we apply the softmax and select top-k candidates. By applying this procedure recursively for each next token, SBS returns a fixed-size batch of samples. However, SBS comes with a few caveats. It generates only fixed-size batches of unique samples and requires $1 + (L - 1)k$ of inferences for the sequence of length L and k samples.

Although beam search and Gumbel top-k sampling guarantee a different output for each beam element, they are not easy to parallelize. Arithmetic Sampling [10] solves this problem as shown in Figure 6.5. First, it samples N numbers from a uniform distribution in the range (0, 1). Next, for each new token, concatenates probability distribution in the range (0, 1) intervals and selects the next tokens whose intervals contain any of the N samples. Finally, it normalizes samples for the next interval and repeats this process until the end-of-sequence token is reached. This method guarantees a diverse set of high-probability samples that are easy to parallelize. Unfortunately, this method may include duplicates.

To solve these problems, Shi et al. propose a Unique Randomizer [11] that incrementally samples sequence models while guaranteeing the uniqueness of each sample. Unique Randomizer constructs a Trie to keep track of probability distribution mass for each token (Figure 6.6). For the first node, the probability distribution mass is set to 1 and is divided recursively on children's nodes proportional to the probability distribution of children's tokens. Once the last token in the sample is generated, it will be added as a leaf node in the trie, while its probability mass will be subtracted from each of its predecessors and itself. This will cause its probability mass to become 0. When the next sample is generated, the next token will be sampled from

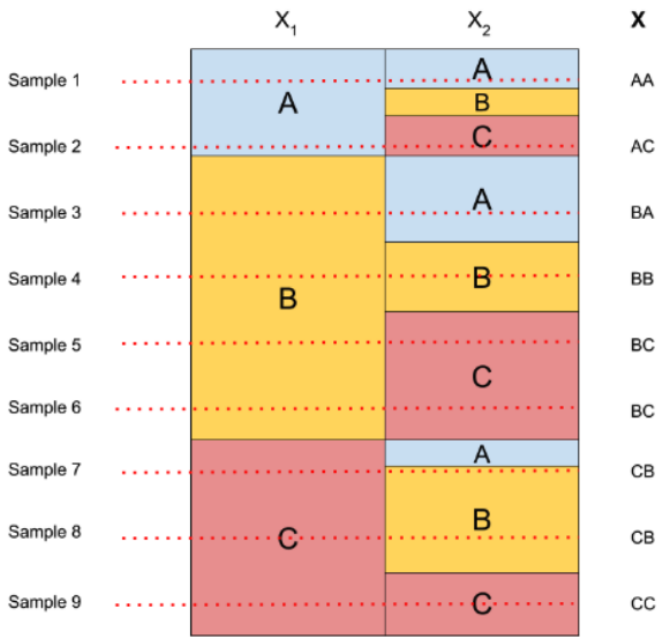


Figure 6.5 : Figure from [10]. Each sample takes a number from the range (0,1) and iteratively finds the next tokens whose probability captures that interval.

the probability distribution of the children nodes, which means that nodes with a probability mass of 0 will never be selected again.

The advantage of this approach is that it draws new unique samples incrementally, enabling us to sample until a timeout is reached, after we get enough coverage, or until we reach enough diversity in samples. Additionally, a Unique Randomizer requires the same number of inferences as non-leaf nodes in the trie, saving the inference for all samples that have the same prefix.

In our work, we further extend the idea of a Unique Randomizer with few key differences. First, we implicitly construct the trie of generated tokens but keep the probabilities of each token in the priority queue with its prefix. This enables us to quickly and deterministically find the node in the trie that needs to be expanded next

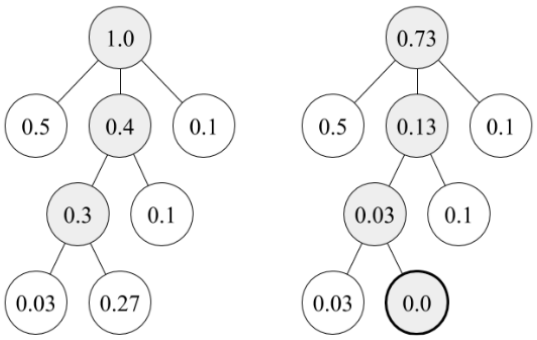


Figure 6.6 : Figure from [11]. Left: trie after the partial generation of a sample [1, 0], immediately before making the third random choice. Right: updated trie after P terminates for the full trace [1, 0, 1].

while avoiding inferences for the prefix tokens. Additionally, we constrain children's expansion to nodes that, together with prefixes, satisfy the regular expression we provide, keeping the size of the trie minimal.

Expanding only nodes with the highest probabilities results in lower Perplexity even than Top-k Sampling, which might not be the right way to process human language but might be beneficial for generating code and learning how to manipulate other discrete optimization tasks. This is because, unlike human language programming languages, assembly and other tasks with discrete grammar didn't evolve to increase information but to remove ambiguity, for which Perplexity should be the right measure for evaluation.

6.6 Discussion

Priority Sampling is a simple inference technique that provides a deterministic and controllable way of generating unique samples for which LLM is the most confident.

Priority Sampling is much more sample-efficient than widely used Nucleus Sampling, reaching 91% of the autotuner improvement over -Oz and even outperforming autotuner for more than 30 samples.

This is a surprising result since the model was trained to replicate the performance of the autotuner and not to surpass it. This finding supports the argument that LLMs store a large amount of knowledge accessible with the clever expansion of the search tree. Additionally, Priority Sampling includes support for regular expression generation that provides a controllable and structured exploration process.

6.6.1 Sequential Implementation

Priority Sampling, however, comes with a few limitations. The current implementation is inherently sequential. We need to construct an augmented search tree to decide what branch needs to be expanded next. One way to parallelize Priority Sampling would be to treat the priority queue as a task generator, from which threads take the next branching position whenever they are idle. Second, Priority Sampling needs to find the top N next tokens that match the regular expression, which is more time-consuming than sampling methods such as Unique Randomizer [11] and Arithmetic Sampling [10]. This is, however, a necessary step for generating samples in order.

Chapter 7

Conclusions and Future Work

Compiler optimization is essential for elevating the efficiency and performance of software on modern processor architectures. The intricate nature of cutting-edge computer architectures, ever-evolving software frameworks, and escalating computation demands have rendered traditional manual optimization techniques more challenging and time-intensive. In response to these challenges, the integration of machine learning (ML) became an instrumental part of the modern compilers. ML adeptly discerns intricate patterns, facilitating the automatic customization of code generation and optimization strategies tailored to specific hardware configurations.

This thesis extends previous work and presents novel findings about machine learning based techniques for improving compiler heuristics.

First, we use reinforcement learning to optimize the domain-specific tensor compiler LoopNest. LoopTune trains a policy network that reorders, and tiles loop nests and applies hardware-specific optimization using LoopNest to tailor the loop nest to the underlying hardware. To map this problem to reinforcement learning, LoopTune introduces a novel action space, graph-based state representation, and reward signal. LoopTune achieves an order of magnitude better results than the optimized implementation of TVM, which uses the same optimizations such as blocking, loop permutation, and vectorization. Additionally, LoopTune outperforms MetaSchedule and AutoTVM by 2.8x and 1.08x on average, generating code again in 1 second, while

MetaSchedule and AutoTVM require 33 seconds and 62 seconds, respectively. This makes real-time auto-tuning possible.

When it comes to limitations, LoopTune supports only loop nests with constant loop sizes. For most of the machine learning computation, this is defined by design. Another limitation is the size of the computation workload. Since the training time of the policy network is proportional to execution time, training on larger problems would require training of the cost model which would predict the performance in constant time.

LoopTune supports only the CPU backend, which could be easily extended by writing support for novel devices in LoopNest and retraining the model with LoopTune. Finally, extending the LoopTune representation with features that describe inner-loop computation would be beneficial in optimizing compute-bound computation.

Second, we evaluate the capability of the 7B-parameter transformer model to predict performant LLVM optimization flags and generate optimized IR itself. The model takes an unoptimized assembly as input and outputs a list of compiler options to optimize the program best. Crucially, during training, we ask the model to predict instruction counts before and after optimization and the optimized code itself. These auxiliary learning tasks significantly improve the model's optimization performance and depth of understanding.

Our approach achieves a 3.0% improvement in reducing instruction counts over the compiler, outperforming two state-of-the-art baselines that require thousands of compilations. Furthermore, the model shows surprisingly strong code reasoning abilities, generating compilable code 91% of the time and perfectly emulating the output of the compiler 70% of the time.

The main limitation of our approach is the short context size and the weak math reasoning abilities of the model. The current context size of 2k tokens allows only function-level optimizations for functions that fit in the context. Novel research ideas such as Retrieval Augmented Generation (RAG), Rotary Position Embeddings(RoPE), and recent length extrapolation techniques significantly lift context size limitations. Additionally, chain-of-thought prompting and tool-enabled LLMs could effectively handle math tasks.

Third, we explored the idea of feedback-directed large language models that use compiler feedback to improve their generation. After generation is finalized, we compile the input with generated optimization passes and evaluate if the predicted instruction count is correct, generated IR is compilable, and corresponds to the compiled code. We provide this feedback back to LLM and give it another chance to optimize the code. We explore three formats of feedback with various degrees of information. All feedback forms outperform the original model on temperature 0 by 0.11%, 0.4%, and 0.53%. We further explore the sampling properties of the Fast Feedback and compare it to the original model. The Fast Feedback model fails to improve the performance of the original model when 10 or more samples are given. From this, we conclude that sampling is an irreplaceable technique for getting high performance and a promising direction to explore.

Finally, we developed a Priority Sampling simple deterministic sampling technique that produces unique samples ordered by the model's confidence and guarantees each sample will satisfy the given regular expression. Each new sample expands the unexpanded token with the highest probability in the constructed search tree. We evaluate Priority Sampling on the task of reducing code size by selecting LLVM optimization passes. Priority Sampling outperforms Nucleus Sampling for any number of samples,

boosting the performance of the original model from 2.87% to 5% improvement over -Oz. Additionally, it outperforms the autotuner used for the generation of labels used for the training of the original model in just 30 samples. This unexpected result supports the argument that LLMs store a large amount of knowledge accessible through the clever expansion of the search tree.

The current implementation of Priority Sampling is inherently sequential. We need to construct an augmented search tree to decide what branch needs to be expanded next. One way to parallelize Priority Sampling would be to treat the priority queue as a task generator, from which threads take the next branching position when they are idle. Additionally, sorting the next token probabilities for each token introduces extra overhead compared to other sampling-based approaches. This is, however, a necessary step for generating samples in order.

Future Work

Using machine learning in compiler design holds great promise for advancing the field and addressing complex optimization challenges.

Reinforcement Learning. With its ability to autonomously learn and adapt based on feedback, reinforcement learning offers an avenue to optimize compiler decisions dynamically. Exploring more sophisticated reinforcement learning algorithms and incorporating them into various stages of the compilation process could lead to more efficient and adaptive code generation. Moreover, the exploration of meta reinforcement learning, where compilers learn to learn from a variety of tasks, holds the potential for creating more generalized and versatile optimization strategies. As the field progresses, the synergy between reinforcement learning and compiler design has potential to redefine the boundaries of code generation, fostering the development

of compilers that optimize for performance and exhibit a nuanced understanding of diverse programming paradigms and architectural configurations.

Novel Code Representations. The development of novel representations, such as graph-based structures and advanced state representations, opens new possibilities for capturing intricate relationships within code. Future research may evolve to enhance these representations to accommodate diverse programming paradigms and languages. The synergy between reinforcement learning and innovative representations can pave the way for compilers that not only optimize for traditional performance metrics but also consider broader aspects such as energy efficiency, security, and specialized hardware architectures. As the field evolves, these advancements have the potential to revolutionize compiler design and play a pivotal role in shaping the next generation of high-performance and adaptable computing systems.

LLMs for code generation. Large language models present a transformative technology for compiler design and code generation. As LLMs evolve and scale, they will be commonly used in software development as code assistants that implement, summarize, and write documentation. To be able to work with large real-world code bases, LLMs will need to extend their context size, implement some kind of hierarchical reasoning, and implement advanced retrieval algorithms to focus on specific pieces of the code. By understanding contextual information and syntactic structures, LLMs can assist developers in producing more efficient and error-resistant code.

LLMs for unit testing. Additionally, LLMs will need to develop mechanisms to guarantee the correctness of the generated code. One possible direction to address this problem is the automatic generation of unit tests that will scale up the development methodology used for human developers. This involves not only generating test cases that cover various code paths but also ensuring that these tests are comprehensive

enough to validate the correctness of complex program logic. The challenge lies in developing intelligent algorithms within LLMs that can autonomously discern potential edge cases and generate relevant tests to cover these scenarios, thereby reinforcing the reliability of the generated code.

Formal verification. Formal verification methods, including mathematical proofs, offer a rigorous approach to ensuring the correctness of software. Integrating formal verification techniques into the LLM-based code generation process involves establishing a mathematical foundation for the generated code's correctness and adherence to specified requirements. This approach holds potential for critical applications where correctness is paramount, such as safety-critical systems and mission-critical software. As LLMs advance, incorporating these mechanisms not only enhances the trustworthiness of the generated code but also fosters the integration of LLMs into domains with stringent correctness and safety standards. The synergy of intelligent testing methodologies and formal verification represents a critical step toward realizing the full potential of LLMs in compiler design and code generation.

LLMs for compilers. Another notable direction involves using LLMs for automating generation of domain-specific languages (DSLs) and compiler heuristics. LLMs' proficiency in understanding natural language specifications could accelerate defining DSLs tailored to specific application domains. This approach not only facilitates the development of more expressive programming languages but also empowers developers who may not possess expertise in low-level language intricacies. Moreover, employing LLMs to learn and optimize compiler heuristics based on vast code repositories could lead to more adaptive and efficient code generation strategies. As LLMs continue to advance, their integration into compiler design will revolutionize how code is written, optimized, and adapted to ever-evolving computing architectures.

Bibliography

[1] Leather, Hugh and Cummins, Chris, "Machine learning in compilers: Past, present and future," in *2020 Forum for Specification and Design Languages (FDL)*, pp. 1–8, IEEE, 2020.

[2] Cummins, Christopher Edward, *Deep learning for compilers*. PhD thesis, University of Edinburgh, UK, 2020.

[3] Sakthivel T., "Neural Network Image." https://www.linkedin.com/pulse/analyzing-3-types-neural-networks-deep-learning-sakthivel-t, 2022 (accessed March 5, 2024).

[4] Grubisic, Dejan and Wasti, Bram and Cummins, Chris and Mellor-Crummey, John and Zlateski, Aleksandar, "LoopTune: Optimizing Tensor Computations with Reinforcement Learning," *arXiv preprint arXiv:2309.01825*, 2023.

[5] Wasti, Bram and Grubisic, Dejan and Steiner, Benoit and Zlateski, Aleksandar, "LoopStack: ML-friendly ML Compiler Stack," in *NeurIPS Workshops, ML For Systems*, no. 22 in NIPS'22, 2022.

[6] Cummins, Chris and Seeker, Volker and Grubisic, Dejan and Elhoushi, Mostafa and Liang, Youwei and Roziere, Baptiste and Gehring, Jonas and Gloeckle, Fabian and Hazelwood, Kim and Synnaeve, Gabriel and others, "Large Language Models for Compiler Optimization," *arXiv preprint arXiv:2309.07062*, 2023.

[7] D. Grubisic, C. Cummins, V. Seeker, and H. Leather, "Compiler generated feedback for large language models," *arXiv preprint arXiv:2403.14714*, 2024.

[8] D. Grubisic, C. Cummins, V. Seeker, and H. Leather, "Priority sampling of large language models for compilers," *arXiv preprint arXiv:2402.18734*, 2024.

[9] Holtzman, Ari and Buys, Jan and Du, Li and Forbes, Maxwell and Choi, Yejin, "The curious case of neural text degeneration," *arXiv preprint arXiv:1904.09751*, 2019.

[10] Vilnis, Luke and Zemlyanskiy, Yury and Murray, Patrick and Passos, Alexandre Tachard and Sanghai, Sumit, "Arithmetic sampling: parallel diverse decoding for large language models," in *Proceedings on Machine Learning Research*, pp. 35120–35136, PMLR, 2023.

[11] Shi, Kensen and Bieber, David and Sutton, Charles, "Incremental sampling without replacement for sequence models," in *Proceedings on Machine Learning Research*, pp. 8785–8795, PMLR, 2020.

[12] Lattner, Chris and Adve, Vikram, "LLVM: A compilation framework for lifelong program analysis & transformation," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pp. 75–86, IEEE, 2004.

[13] Cooper, Keith D and Torczon, Linda, *Engineering a compiler*. Elsevier, 2011.

[14] Franke, Björn and Cummins, Chris and Leather, Hugh and Hazelwood, Kim and Cole, Murray and Seeker, Volker, "Revealing Compiler Heuristics through Automated Discovery and Optimization," in *International Symposium on Code Generation and Optimization*, 2023.

[15] Chen, Yang and Huang, Yuanjie and Eeckhout, Lieven and Fursin, Grigori and Peng, Liang and Temam, Olivier and Wu, Chengyong, "Evaluating iterative optimization across 1000 datasets," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 448–459, 2010.

[16] Desai, Sujay B and Madhvapathy, Surabhi R and Sachid, Angada B and Llinas, Juan Pablo and Wang, Qingxiao and Ahn, Geun Ho and Pitner, Gregory and Kim, Moon J and Bokor, Jeffrey and Hu, Chenming and others, "MOS2 transistors with 1-nanometer gate lengths," *Science*, vol. 354, no. 6308, pp. 99–102, 2016.

[17] Knuth, Donald E and Pardo, Luis Trabb, "The early development of programming languages," *A history of computing in the twentieth century*, pp. 197–273, 1980.

[18] Backus, John Warner and Heising, William P, "Fortran," *IEEE Transactions on Electronic Computers*, no. 4, pp. 382–385, 1964.

[19] Cunningham, Joseph F, "COBOL," *Communications of the ACM*, vol. 6, no. 3, pp. 79–82, 1963.

[20] Kennedy, Ken, *A survey of data flow analysis techniques.* IBM Thomas J. Watson Research Division, 1979.

[21] McKeeman, William M, "Peephole optimization," *Communications of the ACM*, vol. 8, no. 7, pp. 443–444, 1965.

[22] Gillespie, Daniel T, "A general method for numerically simulating the stochastic

time evolution of coupled chemical reactions," *Journal of computational physics*, vol. 22, no. 4, pp. 403–434, 1976.

[23] Orszag, Steven A and Israeli, Moshe, "Numerical simulation of viscous incompressible flows," *Annual Review of Fluid Mechanics*, vol. 6, no. 1, pp. 281–318, 1974.

[24] Fox, Douglas G and Lilly, Douglas K, "Numerical simulation of turbulent flows," *Reviews of Geophysics*, vol. 10, no. 1, pp. 51–72, 1972.

[25] Dongarra, Jack J and Hinds, A_R, "Unrolling loops in Fortran," *Software: Practice and Experience*, vol. 9, no. 3, pp. 219–226, 1979.

[26] Fabri, Janet, "Automatic storage optimization," in *Proceedings of the SIGPLAN Symposium on Compiler Construction*, pp. 83–91, 1979.

[27] Bornat, Richard, "Code Optimisation," in *Understanding and Writing Compilers*, pp. 153–175, Springer, 1979.

[28] Russell, Richard M, "The Cray-1 computer system," *Communications of the ACM*, vol. 21, no. 1, pp. 63–72, 1978.

[29] Goldstein, Ira P and Bobrow, Daniel G, "Extending object oriented programming in Smalltalk," in *Proceedings of the ACM Conference on LISP and Functional Programming*, pp. 75–81, 1980.

[30] Nygaard, Kristen and Dahl, Ole-Johan, "The development of the SIMULA languages," in *History of programming languages*, pp. 439–480, Association for Computing Machinery, 1978.

[31] Stroustrup, Bjarne, "The C++ programming language: reference manual," tech. rep., Bell Lab., 1984.

[32] Caron, John, "Java: Status Report and Language Overview," *CSCI 5535 Project, Dec. 1995, University of Colorado at Boulder*, 1995.

[33] Ellis, Margaret A and Stroustrup, Bjarne, *The annotated C++ reference manual*. Addison-Wesley Longman Publishing Co., Inc., 1990.

[34] Treleaven, Philip C. and Hopkins, Richard P. and Rautenbach, Paul W., "Combining data flow and control flow computing," *The Computer Journal*, vol. 25, no. 2, pp. 207–217, 1982.

[35] Davidson, Jack W and Fraser, Christopher W, "Eliminating redundant object code," in *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 128–132, 1982.

[36] Davidson, Jack W and Holler, Anne M, "A study of a C function inliner," *Software: Practice and Experience*, vol. 18, no. 8, pp. 775–790, 1988.

[37] Lam, Monica, "Software pipelining: An effective scheduling technique for VLIW machines," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 318–328, 1988.

[38] Chow, Frederick and Hennessy, John, "Register allocation by priority-based coloring," in *Proceedings of the SIGPLAN Symposium on Compiler Construction*, pp. 222–232, 1984.

[39] Cooper, Keith D and Kennedy, Ken and Torczon, Linda, "Interprocedural optimization: Eliminating unnecessary recompilation," in *Proceedings of the SIG-*

PLAN Symposium on Compiler Construction, pp. 58–67, 1986.

[40] Pasko, Robert and Schaumont, Patrick and Derudder, Veerle and Vernalde, Serge and Durackova, Daniela, "A new algorithm for elimination of common subexpressions," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 18, no. 1, pp. 58–68, 1999.

[41] Mayrand and Leblanc and Merlo, "Experiment on the automatic detection of function clones in a software system using metrics," in *Proceedings of International Conference on Software Maintenance*, pp. 244–253, IEEE, 1996.

[42] Banerjee, Utpal and Eigenmann, Rudolf and Nicolau, Alexandru and Padua, David A, "Automatic program parallelization," *Proceedings of the IEEE*, vol. 81, no. 2, pp. 211–243, 1993.

[43] Lam, Monica D and Rothberg, Edward E and Wolf, Michael E, "The cache performance and optimizations of blocked algorithms," *ACM SIGOPS Operating Systems Review*, vol. 25, no. Special Issue, pp. 63–74, 1991.

[44] Quilleré, Fabien and Rajopadhye, Sanjay, "Optimizing memory usage in the polyhedral model," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 22, no. 5, pp. 773–815, 2000.

[45] Rawlings, Rosamund, "Objective-C: an object-oriented language for pragmatists," in *IEE Colloquium on Applications of Object-Oriented Programming*, pp. 1–2, IET, 1989.

[46] Sanner, Michel F and others, "Python: a programming language for software integration and development," *J Mol Graph Model*, vol. 17, no. 1, pp. 57–61, 1999.

[47] Ihaka, Ross and Gentleman, Robert, "R: a language for data analysis and graphics," *Journal of computational and graphical statistics*, vol. 5, no. 3, pp. 299–314, 1996.

[48] Jones, Simon Peyton and Hughes, John and Augustsson, Lennart and Barton, Dave and Boutel, Brian and Burton, Warren and Fasel, Joseph and Hammond, Kevin and Hinze, Ralf and Hudak, Paul and others, "Haskell 98," 1999.

[49] Engel, Joshua, *Programming for the Java virtual machine*. Addison-Wesley Professional, 1999.

[50] Cramer, Timothy and Friedman, Richard and Miller, Terrence and Seberger, David and Wilson, Robert and Wolczko, Mario, "Compiling Java just in time," *Ieee micro*, vol. 17, no. 3, pp. 36–43, 1997.

[51] Eich, Brendan and McKinney, C Rand, "JavaScript Language Specification," *Techn. Ber. Netscape Communications, Nov*, pp. 96–002, 1996.

[52] Richards, Robert and Richards, Robert, "Representational state transfer (rest)," *Pro PHP XML and web services*, pp. 633–672, 2006.

[53] Lerdorf, Rasmus and Tatroe, Kevin, *Programming PHP*. O'Reilly Media, Inc., 2002.

[54] Forcier, Jeff and Bissex, Paul and Chun, Wesley J, *Python web development with Django*. Addison-Wesley Professional, 2008.

[55] Hansson, David Heinemeier and Team, Rails Core, "Ruby on rails," *Development*, vol. 4, p. 1, 2009.

[56] Hejlsberg, Anders and Wiltamuth, Scott and Golde, Peter, *C# language specification*. Addison-Wesley Longman Publishing Co., Inc., 2003.

[57] Bhattacharyya, Siddhartha, "DotNet," *Reason-A Technical Journal (Formerly Reason-A Technical Magazine)*, vol. 5, pp. 1–5, 2004.

[58] Virding, Robert and Wikström, Claes and Williams, Mike, *Concurrent programming in ERLANG*. Prentice Hall International (UK) Ltd., 1996.

[59] Clark, Keith L and McCabe, Francis G, "Go!—A multi-paradigm programming language for implementing multi-threaded agents," *Annals of Mathematics and Artificial Intelligence*, vol. 41, pp. 171–206, 2004.

[60] Lattner, Chris, "LLVM and Clang: Next generation compiler technology," in *The BSD conference*, vol. 5, pp. 1–20, 2008.

[61] Marty, Michael R, *Cache coherence techniques for multicore processors*. PhD thesis, University of Wisconsin–Madison, USA, 2008.

[62] Xu, Chi and Chen, Xi and Dick, Robert P and Mao, Zhuoqing Morley, "Cache contention and application performance prediction for multi-core systems," in *2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*, pp. 76–86, IEEE, 2010.

[63] Blagodurov, Sergey and Zhuravlev, Sergey and Fedorova, Alexandra, "Contention-aware scheduling on multicore systems," *ACM Transactions on Computer Systems (TOCS)*, vol. 28, no. 4, pp. 1–45, 2010.

[64] Arnold, Matthew and Fink, Stephen and Grove, David and Hind, Michael and Sweeney, Peter F, "Adaptive optimization in the Jalapeno JVM," in *Proceedings*

of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp. 47–65, 2000.

[65] Arnold, Matthew and Hind, Michael and Ryder, Barbara G, "Online feedback-directed optimization of Java," *ACM SIGPLAN Notices*, vol. 37, no. 11, pp. 111–129, 2002.

[66] Luebke, David, "CUDA: Scalable parallel programming for high-performance scientific computing," in *2008 5th IEEE international symposium on biomedical imaging: from nano to macro*, pp. 836–838, IEEE, 2008.

[67] AMD Corporation, "AMD Stream Computing: Software Stack," *AMD white paper*, 2008.

[68] Krizhevsky, Alex and Sutskever, Ilya and Hinton, Geoffrey E., "ImageNet classification with deep convolutional neural networks," in *Proceedings of the International Conference on Neural Information Processing Systems*, vol. 1 of *NIPS'12*, (Red Hook, NY, USA), p. 1097–1105, Curran Associates Inc., 2012.

[69] Vaswani, Ashish and Shazeer, Noam and Parmar, Niki and Uszkoreit, Jakob and Jones, Llion and Gomez, Aidan N. and Kaiser, Łukasz and Polosukhin, Illia, "Attention is all you need," in *Proceedings of the International Conference on Neural Information Processing Systems*, NIPS'17, (Red Hook, NY, USA), p. 6000–6010, Curran Associates Inc., 2017.

[70] Paszke, Adam and Gross, Sam and Massa, Francisco and Lerer, Adam and Bradbury, James and Chanan, Gregory and Killeen, Trevor and Lin, Zeming and Gimelshein, Natalia and Antiga, Luca and Desmaison, Alban and Kopf,

Andreas and Yang, Edward and DeVito, Zachary and Raison, Martin and Tejani, Alykhan and Chilamkurthy, Sasank and Steiner, Benoit and Fang, Lu and Bai, Junjie and Chintala, Soumith, "PyTorch: An Imperative Style, High-Performance Deep Learning Library," in *Advances in Neural Information Processing Systems 32* (H. Wallach and H. Larochelle and A. Beygelzimer and F. d'Alché-Buc and E. Fox and R. Garnett, ed.), pp. 8024–8035, Curran Associates, Inc., 2019.

[71] Abadi, Martín and Barham, Paul and Chen, Jianmin and Chen, Zhifeng and Davis, Andy and Dean, Jeffrey and Devin, Matthieu and Ghemawat, Sanjay and Irving, Geoffrey and Isard, Michael and others, "Tensorflow: A system for large-scale machine learning," in *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, pp. 265–283, 2016.

[72] Bezanson, Jeff and Karpinski, Stefan and Shah, Viral B and Edelman, Alan, "Julia: A fast dynamic language for technical computing," *arXiv preprint arXiv:1209.5145*, 2012.

[73] Ashari, Arash and Tatikonda, Shirish and Boehm, Matthias and Reinwald, Berthold and Campbell, Keith and Keenleyside, John and Sadayappan, P, "On optimizing machine learning workloads via kernel fusion," *ACM SIGPLAN Notices*, vol. 50, no. 8, pp. 173–182, 2015.

[74] Fey, Matthias and Lenssen, Jan Eric, "Fast graph representation learning with PyTorch Geometric," *arXiv preprint arXiv:1903.02428*, 2019.

[75] Brown, Tom and Mann, Benjamin and Ryder, Nick and Subbiah, Melanie and Kaplan, Jared D and Dhariwal, Prafulla and Neelakantan, Arvind and Shyam,

Pranav and Sastry, Girish and Askell, Amanda and others, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[76] Li, Shen and Zhao, Yanli and Varma, Rohan and Salpekar, Omkar and Noordhuis, Pieter and Li, Teng and Paszke, Adam and Smith, Jeff and Vaughan, Brian and Damania, Pritam and Chintala, Soumith, "PyTorch distributed: experiences on accelerating data parallel training," *Proceedings of the VLDB Endowment*, vol. 13, p. 3005–3018, aug 2020.

[77] Moritz, Philipp and Nishihara, Robert and Wang, Stephanie and Tumanov, Alexey and Liaw, Richard and Liang, Eric and Elibol, Melih and Yang, Zongheng and Paul, William and Jordan, Michael I and others, "Ray: A distributed framework for emerging AI applications," in *13th USENIX symposium on operating systems design and implementation (OSDI 18)*, pp. 561–577, 2018.

[78] Sergeev, Alexander and Del Balso, Mike, "Horovod: fast and easy distributed deep learning in TensorFlow," *arXiv preprint arXiv:1802.05799*, 2018.

[79] Rasley, Jeff and Rajbhandari, Samyam and Ruwase, Olatunji and He, Yuxiong, "Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters," in *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 3505–3506, 2020.

[80] Frostig, Roy and Johnson, Matthew James and Leary, Chris, "Compiling machine learning programs via high-level tracing," *Systems for Machine Learning*, vol. 4, no. 9, 2018.

[81] Cito, Jürgen and Gall, Harald C., "Using Docker containers to improve reproducibility in software engineering research," in *Proceedings of the International Conference on Software Engineering Companion*, ICSE '16, (New York, NY, USA), p. 906–907, Association for Computing Machinery, 2016.

[82] Saito, Hideto and Lee, Hui-Chuan Chloe and Hsu, Ke-Jou Carol, *Kubernetes Cookbook*. Packt Publishing, 2016.

[83] Meng, Xiangrui and Bradley, Joseph and Yavuz, Burak and Sparks, Evan and Venkataraman, Shivaram and Liu, Davies and Freeman, Jeremy and Tsai, DB and Amde, Manish and Owen, Sean and others, "Mllib: Machine learning in Apache Spark," *The journal of machine learning research*, vol. 17, no. 1, pp. 1235–1241, 2016.

[84] White, Tom, *Hadoop: The definitive guide*. O'Reilly Media, Inc., 2012.

[85] Garg, Nishant, *Apache Kafka*. Packt Publishing Birmingham, UK, 2013.

[86] Rocklin, Matthew and others, "Dask: Parallel computation with blocked algorithms and task scheduling," in *Proceedings of the Python in Science Conference*, vol. 130, p. 136, SciPy Austin, TX, 2015.

[87] LaValle, Steven M, *Virtual reality*. Cambridge university press, 2023.

[88] Chen, Yunqiang and Wang, Qing and Chen, Hong and Song, Xiaoyu and Tang, Hui and Tian, Mengxiao, "An overview of augmented reality technology," in *Journal of Physics: Conference Series*, vol. 1237, p. 022082, IOP Publishing, 2019.

[89] Badue, Claudine and Guidolini, Rânik and Carneiro, Raphael Vivacqua and Azevedo, Pedro and Cardoso, Vinicius B and Forechi, Avelino and Jesus, Luan and Berriel, Rodrigo and Paixao, Thiago M and Mutz, Filipe and others, "Self-driving cars: A survey," *Expert Systems with Applications*, vol. 165, p. 113816, 2021.

[90] Stallman, Richard M and others, *Using and porting the GNU compiler collection*, vol. 86. Free Software Foundation Boston, MA, USA, 1999.

[91] Bodin, François and Kisuki, Toru and Knijnenburg, Peter and O'Boyle, Mike and Rohou, Erven, "Iterative compilation in a non-linear optimisation space," in *Workshop on profile and feedback-directed compilation*, 1998.

[92] Pan, Zhelong and Eigenmann, Rudolf, "Fast and Effective Orchestration of Compiler Optimizations for Automatic Performance Tuning," in *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '06, (USA), p. 319–332, IEEE Computer Society, 2006.

[93] Martins, Luiz GA and Nobre, Ricardo and Cardoso, Joao MP and Delbem, Alexandre CB and Marques, Eduardo, "Clustering-based selection for the exploration of compiler optimization sequences," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 13, no. 1, pp. 1–28, 2016.

[94] Nobre, Ricardo and Martins, Luiz GA and Cardoso, Joao MP, "Use of previously acquired positioning of optimizations for phase ordering exploration," in *Proceedings of the International Workshop on Software and Compilers for Embedded Systems*, pp. 58–67, 2015.

[95] Ashouri, Amir Hossein, "Design space exploration methodology for compiler parameters in VLIW processors," *Archive of Politecnico di Milano*, 2012.

[96] Almagor, Lelac and Cooper, Keith D and Grosul, Alexander and Harvey, Timothy J and Reeves, Steven W and Subramanian, Devika and Torczon, Linda and Waterman, Todd, "Finding effective compilation sequences," *ACM SIGPLAN Notices*, vol. 39, no. 7, pp. 231–239, 2004.

[97] Cooper, Keith D and Schielke, Philip J and Subramanian, Devika, "Optimizing for reduced code space using genetic algorithms," in *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, pp. 1–9, 1999.

[98] Beaty, Steven J, "Genetic algorithms and instruction scheduling," in *Proceedings of the Annual International Symposium on Microarchitecture*, pp. 206–211, 1991.

[99] White, David R and Arcuri, Andrea and Clark, John A, "Evolutionary improvement of programs," *IEEE Transactions on Evolutionary Computation*, vol. 15, no. 4, pp. 515–538, 2011.

[100] Kukunas, James and Cupper, Robert D and Kapfhammer, Gregory M, "A genetic algorithm to improve Linux kernel performance on resource-constrained devices," in *Proceedings of the Annual Conference Companion on Genetic and Evolutionary Computation*, pp. 2095–2096, 2010.

[101] Harman, Mark and Langdon, William B and Jia, Yue and White, David R and Arcuri, Andrea and Clark, John A, "The GISMOE challenge: Constructing the pareto program surface using genetic programming to find better programs

(keynote paper)," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, pp. 1–14, 2012.

[102] Cooper, Keith D and Grosul, Alexander and Harvey, Timothy J and Reeves, Steven and Subramanian, Devika and Torczon, Linda and Waterman, Todd, "ACME: adaptive compilation made efficient," *ACM SIGPLAN Notices*, vol. 40, no. 7, pp. 69–77, 2005.

[103] Ashouri, Amir H and Bignoli, Andrea and Palermo, Gianluca and Silvano, Cristina and Kulkarni, Sameer and Cavazos, John, "Micomp: Mitigating the compiler phase-ordering problem using optimization sub-sequences and machine learning," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 14, no. 3, pp. 1–28, 2017.

[104] Cooper, Keith D and Subramanian, Devika and Torczon, Linda, "Adaptive optimizing compilers for the 21st century," *The Journal of Supercomputing*, vol. 23, pp. 7–22, 2002.

[105] Pouchet, Louis-Noël and Bastoul, Cédric and Cohen, Albert and Cavazos, John, "Iterative optimization in the polyhedral model: Part II, multidimensional time," *ACM SIGPLAN Notices*, vol. 43, no. 6, pp. 90–100, 2008.

[106] Whaley, R Clinton and Dongarra, Jack J, "Automatically tuned linear algebra software," in *SC'98: Proceedings of the ACM/IEEE Conference on Supercomputing*, pp. 38–38, IEEE, 1998.

[107] Puschel, Markus and Moura, José MF and Johnson, Jeremy R and Padua, David and Veloso, Manuela M and Singer, Bryan W and Xiong, Jianxin and Franchetti, Franz and Gacic, Aca and Voronenko, Yevgen and others, "SPIRAL:

Code generation for DSP transforms," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 232–275, 2005.

[108] Ansel, Jason and Chan, Cy and Wong, Yee Lok and Olszewski, Marek and Zhao, Qin and Edelman, Alan and Amarasinghe, Saman, "PetaBricks: A language and compiler for algorithmic choice," *ACM Sigplan Notices*, vol. 44, no. 6, pp. 38–49, 2009.

[109] Steuwer, Michel and Remmelg, Toomas and Dubach, Christophe, "Lift: a functional data-parallel IR for high-performance GPU code generation," in *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pp. 74–85, IEEE, 2017.

[110] Lee, Kyong-Ha and Lee, Yoon-Joon and Choi, Hyunsik and Chung, Yon Dohn and Moon, Bongki, "Parallel data processing with MapReduce: a survey," *AcM sIGMoD record*, vol. 40, no. 4, pp. 11–20, 2012.

[111] Holewinski, Justin and Pouchet, Louis-Noël and Sadayappan, Ponnuswamy, "High-performance code generation for stencil computations on GPU architectures," in *Proceedings of the ACM international Conference on Supercomputing*, pp. 311–320, 2012.

[112] Cavazos, John and Dubach, Christophe and Agakov, Felix and Bonilla, Edwin and O'Boyle, Michael FP and Fursin, Grigori and Temam, Olivier, "Automatic performance model construction for the fast software exploration of new hardware designs," in *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, pp. 24–34, 2006.

[113] Cavazos, John and O'Boyle, Michael FP, "Method-specific dynamic compilation

using logistic regression," *ACM SIGPLAN Notices*, vol. 41, no. 10, pp. 229–240, 2006.

[114] Agakov, F. and Bonilla, E. and Cavazos, J. and Franke, B. and Fursin, G. and O'Boyle, M. F. P. and Thomson, J. and Toussaint, M. and Williams, C. K. I., "Using Machine Learning to Focus Iterative Optimization," in *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '06, (USA), p. 295–305, IEEE Computer Society, 2006.

[115] Huang, Qijing and Haj-Ali, Ameer and Moses, William and Xiang, John and Stoica, Ion and Asanovic, Krste and Wawrzynek, John, "Autophase: Compiler phase-ordering for High Level Synthesis with Deep Reinforcement Learning," in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 308–308, IEEE, 2019.

[116] Ferrante, Jeanne and Ottenstein, Karl J and Warren, Joe D, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 9, no. 3, pp. 319–349, 1987.

[117] Ottenstein, Karl Joseph, *Data-flow graphs as an intermediate program form.* PhD thesis, Purdue University, USA, 1978.

[118] Park, Eunjung and Cavazos, John and Alvarez, Marco A, "Using graph-based program characterization for predictive modeling," in *Proceedings of the International Symposium on Code Generation and Optimization*, pp. 196–206, 2012.

[119] Fursin, Grigori and Kashnikov, Yuriy and Memon, Abdul Wahid and Chamski, Zbigniew and Temam, Olivier and Namolaru, Mircea and Yom-Tov, Elad and Mendelson, Bilha and Zaks, Ayal and Courtois, Eric and others, "Milepost

GCC: Machine learning enabled self-tuning compiler," *International journal of parallel programming*, vol. 39, pp. 296–327, 2011.

[120] Cavazos, John and Fursin, Grigori and Agakov, Felix and Bonilla, Edwin and O'Boyle, Michael FP and Temam, Olivier, "Rapidly selecting good compiler optimizations using performance counters," in *International Symposium on Code Generation and Optimization (CGO'07)*, pp. 185–197, IEEE, 2007.

[121] Mucci, Philip J and Browne, Shirley and Deane, Christine and Ho, George, "PAPI: A portable interface to hardware performance counters," in *Proceedings of the Department of Defense HPCMP Users Group Conference*, vol. 710, 1999.

[122] Adhianto, Laksono and Banerjee, Sinchan and Fagan, Mike and Krentel, Mark and Marin, Gabriel and Mellor-Crummey, John and Tallent, Nathan R, "HPCToolkit: Tools for performance analysis of optimized parallel programs," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, pp. 685–701, 2010.

[123] Graham, Susan L and Kessler, Peter B and McKusick, Marshall K, "Gprof: A call graph execution profiler," *ACM SIGPLAN Notices*, vol. 17, no. 6, pp. 120–126, 1982.

[124] Ashouri, Amir Hossein and Mariani, Giovanni and Palermo, Gianluca and Park, Eunjung and Cavazos, John and Silvano, Cristina, "Cobayn: Compiler autotuning framework using bayesian networks," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 13, no. 2, pp. 1–25, 2016.

[125] Cox, David Roxbee and Snell, E Joyce, *Analysis of binary data*, vol. 32. CRC press, 1989.

[126] Vapnik, Vladimir N and Chervonenkis, A Ya, "On the uniform convergence of relative frequencies of events to their probabilities," in *Measures of complexity*, pp. 11–30, Springer, 2015.

[127] Fisher, Ronald A, "The use of multiple measurements in taxonomic problems," *Annals of eugenics*, vol. 7, no. 2, pp. 179–188, 1936.

[128] Breiman, Leo, "Random forests," *Machine learning*, vol. 45, pp. 5–32, 2001.

[129] Rumelhart, David E and Hinton, Geoffrey E and Williams, Ronald J, "Learning representations by back-propagating errors," *nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[130] Zhou, Jie and Cui, Ganqu and Hu, Shengding and Zhang, Zhengyan and Yang, Cheng and Liu, Zhiyuan and Wang, Lifeng and Li, Changcheng and Sun, Maosong, "Graph neural networks: A review of methods and applications," *AI open*, vol. 1, pp. 57–81, 2020.

[131] Pearl, Judea, *Probabilistic reasoning in intelligent systems: networks of plausible inference*. Morgan Kaufmann, 1988.

[132] Kingma, Diederik P and Welling, Max, "Auto-encoding variational bayes," *arXiv preprint arXiv:1312.6114*, 2013.

[133] Ho, Jonathan and Jain, Ajay and Abbeel, Pieter, "Denoising diffusion probabilistic models," *Advances in neural information processing systems*, vol. 33, pp. 6840–6851, 2020.

[134] Goodfellow, Ian and Pouget-Abadie, Jean and Mirza, Mehdi and Xu, Bing and Warde-Farley, David and Ozair, Sherjil and Courville, Aaron and Ben-

gio, Yoshua, "Generative adversarial networks," *Communications of the ACM*, vol. 63, no. 11, pp. 139–144, 2020.

[135] Schölkopf, Bernhard, "The kernel trick for distances," *Advances in neural information processing systems*, vol. 13, 2000.

[136] Sanchez, Ricardo Nabinger and Amaral, Jose Nelson and Szafron, Duane and Pirvu, Marius and Stoodley, Mark, "Using machines to learn method-specific compilation strategies," in *International Symposium on Code Generation and Optimization (CGO 2011)*, pp. 257–266, IEEE, 2011.

[137] Stephenson, Mark and Amarasinghe, Saman, "Predicting unroll factors using supervised classification," in *International symposium on code generation and optimization*, pp. 123–134, IEEE, 2005.

[138] Park, Eunjung and Kulkarni, Sameer and Cavazos, John, "An evaluation of different modeling techniques for iterative compilation," in *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, pp. 65–74, 2011.

[139] Fraser, Christopher W, "Automatic inference of models for statistical code compression," *ACM SIGPLAN Notices*, vol. 34, no. 5, pp. 242–246, 1999.

[140] Monsifrot, Antoine and Bodin, François and Quiniou, Rene, "A machine learning approach to automatic production of compiler heuristics," in *Proceedings of the International Conference on Artificial Intelligence: Methodology, Systems, and Applications*, pp. 41–50, Springer, 2002.

[141] Herrera, Victor M and Khoshgoftaar, Taghi M and Villanustre, Flavio and Furht, Borko, "Random forest implementation and optimization for Big Data

analytics on LexisNexis's high performance computing cluster platform," *Journal of Big Data*, vol. 6, no. 1, pp. 1–36, 2019.

[142] Sharma, Sagar and Sharma, Simone and Athaiya, Anidhya, "Activation functions in neural networks," *Towards Data Sci*, vol. 6, no. 12, pp. 310–316, 2017.

[143] Kingma, Diederik P and Ba, Jimmy, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[144] Tieleman, Tijmen and Hinton, G, "RMSprop: Divide the gradient by a running average of its recent magnitude," *Notes from Lecture, slide 27*, 2017.

[145] Zeiler, Matthew D, "Adadelta: an adaptive learning rate method," *arXiv preprint arXiv:1212.5701*, 2012.

[146] Ioffe, Sergey and Szegedy, Christian, "Batch Normalization: Accelerating deep network training by reducing internal covariate shift," in *Proceedings on Machine Learning Research*, pp. 448–456, PMLR, 2015.

[147] Yang, Shouguo and Shi, Zhiqiang and Zhang, Guodong and Li, Mingxuan and Ma, Yuan and Sun, Limin, "Understand code style: Efficient cnn-based compiler optimization recognition system," in *ICC 2019-2019 IEEE International Conference on Communications (ICC)*, pp. 1–6, IEEE, 2019.

[148] Sharma, Tushar and Efstathiou, Vasiliki and Louridas, Panos and Spinellis, Diomidis, "On the feasibility of transfer-learning code smells using deep learning," *arXiv preprint arXiv:1904.03031*, 2019.

[149] Cummins, Chris and Fisches, Zacharias V and Ben-Nun, Tal and Hoefler, Torsten and O'Boyle, Michael FP and Leather, Hugh, "Programl: A graph-

based program representation for data flow analysis and compiler optimizations," in *Proceedings on Machine Learning Research*, pp. 2244–2253, PMLR, 2021.

[150] Cyphers, Scott and Bansal, Arjun K and Bhiwandiwalla, Anahita and Bobba, Jayaram and Brookhart, Matthew and Chakraborty, Avijit and Constable, Will and Convey, Christian and Cook, Leona and Kanawi, Omar and others, "Intel nGraph: An intermediate representation, compiler, and executor for deep learning," *arXiv preprint arXiv:1801.08058*, 2018.

[151] Amit Sabne, "XLA : Compiling Machine Learning for Peak Performance." https://github.com/openxla/xla, 2020. GitHub Repository.

[152] Allamanis, Miltiadis and Barr, Earl T and Devanbu, Premkumar and Sutton, Charles, "A survey of machine learning for big code and naturalness," *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, pp. 1–37, 2018.

[153] Brauckmann, Alexander and Goens, Andrés and Ertel, Sebastian and Castrillon, Jeronimo, "Compiler-based graph representations for deep learning models of code," in *Proceedings of the International Conference on Compiler Construction*, pp. 201–211, 2020.

[154] Dam, Hoa Khanh and Pham, Trang and Ng, Shien Wee and Tran, Truyen and Grundy, John and Ghose, Aditya and Kim, Taeksu and Kim, Chul-Joo, "Lessons learned from using a deep tree-based model for software defect prediction in practice," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pp. 46–57, IEEE, 2019.

[155] Mou, Lili and Li, Ge and Zhang, Lu and Wang, Tao and Jin, Zhi, "Convolutional neural networks over tree structures for programming language processing," in *Proceedings of the AAAI Conference on Artificial Intelligence*, AAAI'16, p. 1287–1293, AAAI Press, 2016.

[156] Mou, Lili and Li, Ge and Zhang, Lu and Wang, Tao and Jin, Zhi, "Convolutional neural networks over tree structures for programming language processing," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 30, 2016.

[157] Li, Yujia and Gu, Chenjie and Dullien, Thomas and Vinyals, Oriol and Kohli, Pushmeet, "Graph matching networks for learning the similarity of graph structured objects," in *Proceedings on Machine Learning Research*, pp. 3835–3845, PMLR, 2019.

[158] Yang, Junhan and Liu, Zheng and Xiao, Shitao and Li, Chaozhuo and Lian, Defu and Agrawal, Sanjay and Singh, Amit and Sun, Guangzhong and Xie, Xing, "GraphFormers: GNN-nested transformers for representation learning on textual graph," *Advances in Neural Information Processing Systems*, vol. 34, pp. 28798–28810, 2021.

[159] Haj-Ali, Ameer and Ahmed, Nesreen K and Willke, Ted and Shao, Yakun Sophia and Asanovic, Krste and Stoica, Ion, "Neurovectorizer: End-to-end vectorization with deep reinforcement learning," in *Proceedings of the ACM/IEEE International Symposium on Code Generation and Optimization*, pp. 242–255, 2020.

[160] Ahn, Byung Hoon and Pilligundla, Prannoy and Yazdanbakhsh, Amir and Esmaeilzadeh, Hadi, "Chameleon: Adaptive code optimization for expedited deep

neural network compilation," *arXiv preprint arXiv:2001.08743*, 2020.

[161] Mircea Trofin and Yundi Qian and Eugene Brevdo and Zinan Lin and Krzysztof Choromanski and David Li, "MLGO: a Machine Learning Guided Compiler Optimizations Framework," *arXiv:2101.04808*, 2021.

[162] Brauckmann, Alexander and Goens, Andrés and Castrillon, Jeronimo, "A reinforcement learning environment for polyhedral optimizations," *arXiv preprint arXiv:2104.13732*, 2021.

[163] Cummins, Chris and Wasti, Bram and Guo, Jiadong and Cui, Brandon and Ansel, Jason and Gomez, Sahir and Jain, Somya and Liu, Jia and Teytaud, Olivier and Steiner, Benoit and Tian, Yuandong and Leather, Hugh, "CompilerGym: Robust, Performant Compiler Optimization Environments for AI Research," in *CGO*, 2022.

[164] Neelakantan, Arvind and Le, Quoc V and Sutskever, Ilya, "Neural programmer: Inducing latent programs with gradient descent," *arXiv preprint arXiv:1511.04834*, 2015.

[165] Liang, Chen and Berant, Jonathan and Le, Quoc and Forbus, Kenneth D and Lao, Ni, "Neural symbolic machines: Learning semantic parsers on freebase with weak supervision," *arXiv preprint arXiv:1611.00020*, 2016.

[166] Ye, Wei and Xie, Rui and Zhang, Jinglei and Hu, Tianxiang and Wang, Xiaoyin and Zhang, Shikun, "Leveraging Code Generation to Improve Code Retrieval and Summarization via Dual Learning," in *Proceedings of The Web Conference 2020*, WWW '20, (New York, NY, USA), p. 2309–2319, Association for Computing Machinery, 2020.

[167] Shido, Yusuke and Kobayashi, Yasuaki and Yamamoto, Akihiro and Miyamoto, Atsushi and Matsumura, Tadayuki, "Automatic source code summarization with extended tree-lstm," in *2019 International Joint Conference on Neural Networks (IJCNN)*, pp. 1–8, IEEE, 2019.

[168] Li, Yujia and Choi, David and Chung, Junyoung and Kushman, Nate and Schrittwieser, Julian and Leblond, Rémi and Eccles, Tom and Keeling, James and Gimeno, Felix and Lago, Agustin Dal and Hubert, Thomas and Choy, Peter and d'Autume, Cyprien de Masson and Babuschkin, Igor and Chen, Xinyun and Huang, Po-Sen and Welbl, Johannes and Gowal, Sven and Cherepanov, Alexey and Molloy, James and Mankowitz, Daniel J. and Robson, Esme Sutherland and Kohli, Pushmeet and de Freitas, Nando and Kavukcuoglu, Koray and Vinyals, Oriol, "Competition-Level Code Generation with AlphaCode," *Science*, vol. 378, no. 6624, 2022.

[169] Baptiste Rozière and Jonas Gehring and Fabian Gloeckle and Sten Sootla and Itai Gat and Xiaoqing Ellen Tan and Yossi Adi and Jingyu Liu and Tal Remez and Jérémy Rapin and Artyom Kozhevnikov and Ivan Evtimov and Joanna Bitton and Manish Bhatt and Cristian Canton Ferrer and Aaron Grattafiori and Wenhan Xiong and Alexandre Défossez and Jade Copet and Faisal Azhar and Hugo Touvron and Louis Martin and Nicolas Usunier and Thomas Scialom and Gabriel Synnaeve, "Code Llama: Open Foundation Models for Code," *arXiv:2308.12950*, 2023.

[170] OpenAI, "ChatGPT." https://chat.openai.com/.

[171] Feng, Zhangyin and Guo, Daya and Tang, Duyu and Duan, Nan and Feng, Xiaocheng and Gong, Ming and Shou, Linjun and Qin, Bing and Liu, Ting and

Jiang, Daxin and others, "CodeBERT: A Pre-trained Model for Programming and Natural Languages," *arXiv:2002.08155*, 2020.

[172] Daya Guo and Shuo Ren and Shuai Lu and Zhangyin Feng and Duyu Tang and Shujie Liu and Long Zhou and Nan Duan and Alexey Svyatkovskiy and Shengyu Fu and Michele Tufano and Shao Kun Deng and Colin Clement and Dawn Drain and Neel Sundaresan and Jian Yin and Daxin Jiang and Ming Zhou, "GraphCodeBERT: Pre-training Code Representations with Data Flow," *arXiv:2009.08366*, 2021.

[173] Wang, Yue and Wang, Weishi and Joty, Shafiq and Hoi, Steven CH, "CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation," *arXiv:2109.00859*, 2021.

[174] Li, Raymond and Allal, Loubna Ben and Zi, Yangtian and Muennighoff, Niklas and Kocetkov, Denis and Mou, Chenghao and Marone, Marc and Akiki, Christopher and Li, Jia and Chim, Jenny and Liu, Qian and Zheltonozhskii, Evgenii and Zhuo, Terry Yue and Wang, Thomas and Dehaene, Olivier and Davaadorj, Mishig and Lamy-Poirier, Joel and Monteiro, João and Shliazhko, Oleh and Gontier, Nicolas and Meade, Nicholas and Zebaze, Armel and Yee, Ming-Ho and Umapathi, Logesh Kumar and Zhu, Jian and Lipkin, Benjamin and Oblokulov, Muhtasham and Wang, Zhiruo and Murthy, Rudra and Stillerman, Jason and Patel, Siva Sankalp and Abulkhanov, Dmitry and Zocca, Marco and Dey, Manan and Zhang, Zhihan and Fahmy, Nour and Bhattacharyya, Urvashi and Yu, Wenhao and Singh, Swayam and Luccioni, Sasha and Villegas, Paulo and Kunakov, Maxim and Zhdanov, Fedor and Romero, Manuel and Lee, Tony and Timor, Nadav and Ding, Jennifer and Schlesinger, Claire and

Schoelkopf, Hailey and Ebert, Jan and Dao, Tri and Mishra, Mayank and Gu, Alex and Robinson, Jennifer and Anderson, Carolyn Jane and Dolan-Gavitt, Brendan and Contractor, Danish and Reddy, Siva and Fried, Daniel and Bahdanau, Dzmitry and Jernite, Yacine and Ferrandis, Carlos Muñoz and Hughes, Sean and Wolf, Thomas and Guha, Arjun and von Werra, Leandro and de Vries, Harm, "StarCoder: may the source be with you!," *arXiv:2305.06161*, 2023.

[175] Loubna Ben Allal and Raymond Li and Denis Kocetkov and Chenghao Mou and Christopher Akiki and Carlos Munoz Ferrandis and Niklas Muennighoff and Mayank Mishra and Alex Gu and Manan Dey and Logesh Kumar Umapathi and Carolyn Jane Anderson and Yangtian Zi and Joel Lamy Poirier and Hailey Schoelkopf and Sergey Troshin and Dmitry Abulkhanov and Manuel Romero and Michael Lappert and Francesco De Toni and Bernardo García del Río and Qian Liu and Shamik Bose and Urvashi Bhattacharyya and Terry Yue Zhuo and Ian Yu and Paulo Villegas and Marco Zocca and Sourab Mangrulkar and David Lansky and Huu Nguyen and Danish Contractor and Luis Villa and Jia Li and Dzmitry Bahdanau and Yacine Jernite and Sean Hughes and Daniel Fried and Arjun Guha and Harm de Vries and Leandro von Werra, "SantaCoder: don't reach for the stars!," *arXiv:2301.03988*, 2023.

[176] DeepSeek, "DeepSeek Coder: Let the Code Write Itself." https://github.com/deepseek-ai/DeepSeek-Coder, 2023.

[177] Ye, Guixin and Tang, Zhanyong and Tan, Shin Hwei and Huang, Songfang and Fang, Dingyi and Sun, Xiaoyang and Bian, Lizhong and Wang, Haibo and Wang, Zheng, "Automated conformance testing for JavaScript engines via deep compiler fuzzing," in *Proceedings of the ACM SIGPLAN International*

Conference on Programming Language Design and Implementation, PLDI 2021, (New York, NY, USA), p. 435–450, Association for Computing Machinery, 2021.

[178] Deng, Yinlin and Xia, Chunqiu Steven and Peng, Haoran and Yang, Chenyuan and Zhang, Lingming, "Large Language Models Are Zero-Shot Fuzzers: Fuzzing Deep-Learning Libraries via Large Language Models," in *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2023, (New York, NY, USA), p. 423–435, Association for Computing Machinery, 2023.

[179] Chunqiu Steven Xia and Matteo Paltenghi and Jia Le Tian and Michael Pradel and Lingming Zhang, "Universal Fuzzing via Large Language Models," *arXiv:2308.04748*, 2023.

[180] Max Schäfer and Sarah Nadi and Aryaz Eghbali and Frank Tip, "Adaptive Test Generation Using a Large Language Model," *arXiv:2302.06527*, 2023.

[181] Szafraniec, Marc and Roziere, Baptiste and Charton, Francois and Leather, Hugh and Labatut, Patrick and Synnaeve, Gabriel, "Code Translation with Compiler Representations," *arXiv:2207.03578*, 2022.

[182] Gallagher, Shannon K and Klieber, William E and Svoboda, David, "LLVM Intermediate Representation for Code Weakness Identification," 2022.

[183] Xia, Chunqiu Steven and Zhang, Lingming, "Less training, more repairing please: revisiting automated program repair via zero-shot learning," in *Proceedings of the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 959–971, 2022.

[184] Chunqiu Steven Xia and Lingming Zhang, "Keep the Conversation Going: Fixing 162 out of 337 bugs for $0.42 each using ChatGPT," *arXiv:2304.00385*, 2023.

[185] Chen, Xue-Wen and Lin, Xiaotong, "Big data deep learning: challenges and perspectives," *IEEE access*, vol. 2, pp. 514–525, 2014.

[186] Torrey, Lisa and Shavlik, Jude, "Transfer learning," in *Handbook of research on machine learning applications and trends: algorithms, methods, and techniques*, pp. 242–264, IGI global, 2010.

[187] Hadi, Muhammad Usman and Qureshi, Rizwan and Shah, Abbas and Irfan, Muhammad and Zafar, Anas and Shaikh, Muhammad Bilal and Akhtar, Naveed and Wu, Jia and Mirjalili, Seyedali and others, "A survey on large language models: Applications, challenges, limitations, and practical usage," *Authorea Preprints*, 2023.

[188] Yang, Xuejun and Chen, Yang and Eide, Eric and Regehr, John, "Finding and understanding bugs in C compilers," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 283–294, 2011.

[189] Tsimpourlas, Foivos and Petoumenos, Pavlos and Xu, Min and Cummins, Chris and Hazelwood, Kim and Rajan, Ajitha and Leather, Hugh, "Benchpress: A deep active benchmark generator," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pp. 505–516, 2022.

[190] Cummins, Chris and Petoumenos, Pavlos and Wang, Zheng and Leather, Hugh, "Synthesizing benchmarks for predictive modeling," in *2017 IEEE/ACM Inter-*

national Symposium on Code Generation and Optimization (CGO), pp. 86–99, IEEE, 2017.

[191] Goens, Andrés and Brauckmann, Alexander and Ertel, Sebastian and Cummins, Chris and Leather, Hugh and Castrillon, Jeronimo, "A case study on machine learning for synthesizing benchmarks," in *Proceedings of the ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pp. 38–46, 2019.

[192] Tsimpourlas, Foivos and Petoumenos, Pavlos and Xu, Min and Cummins, Chris and Hazelwood, Kim and Rajan, Ajitha and Leather, Hugh, "BenchDirect: A Directed Language Model for Compiler Benchmarks," *arXiv preprint arXiv:2303.01557*, 2023.

[193] Markidis, Stefano and Der Chien, Steven Wei and Laure, Erwin and Peng, Ivy Bo and Vetter, Jeffrey S, "Nvidia tensor core programmability, performance & precision," in *2018 IEEE international parallel and distributed processing symposium workshops (IPDPSW)* (Pat Langley, ed.), (Stanford, CA), pp. 522–531, IEEE, 2018.

[194] Choquette, Jack and Gandhi, Wishwesh and Giroux, Olivier and Stam, Nick and Krashinsky, Ronny, "NVIDIA A100 tensor core GPU: Performance and innovation," *IEEE Micro*, vol. 41, no. 2, pp. 29–35, 2021.

[195] Lomont, Chris, "Introduction to intel advanced vector extensions," *Intel white paper*, vol. 23, pp. 1–21, 2011.

[196] Jeong, Hwancheol and Kim, Sunghoon and Lee, Weonjong and Myung, Seok-Ho, "Performance of SSE and AVX instruction sets," *arXiv preprint*

arXiv:1211.0820, 2012.

[197] Wittmann, Markus and Zeiser, Thomas and Hager, Georg and Wellein, Gerhard, "Short note on costs of floating point operations on current x86-64 architectures: Denormals, overflow, underflow, and division by zero," *arXiv preprint arXiv:1506.03997*, 2015.

[198] Tekin, A and Tuncer Durak, A and Piechurski, C and Kaliszan, D and Aylin Sungur, F and Robertsén, F and Gschwandtner, P, "State-of-the-art and trends for computing and interconnect network solutions for HPC and AI," *Partnership for Advanced Computing in Europe, Available online at www.praceri.eu*, 2021.

[199] Jouppi, Norman P and Young, Cliff and Patil, Nishant and Patterson, David and Agrawal, Gaurav and Bajwa, Raminder and Bates, Sarah and Bhatia, Suresh and Boden, Nan and Borchers, Al and others, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the Annual International Symposium on Computer Architecture*, pp. 1–12, 2017.

[200] Jia, Zhe and Tillman, Blake and Maggioni, Marco and Scarpazza, Daniele Paolo, "Dissecting the Graphcore IPU architecture via microbenchmarking," *arXiv preprint arXiv:1912.03413*, 2019.

[201] Rocki, Kamil and Van Essendelft, Dirk and Sharapov, Ilya and Schreiber, Robert and Morrison, Michael and Kibardin, Vladimir and Portnoy, Andrey and Dietiker, Jean Francois and Syamlal, Madhava and James, Michael, "Fast stencil-code computation on a wafer-scale processor," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–14, IEEE, 2020.

[202] Chetlur, Sharan and Woolley, Cliff and Vandermersch, Philippe and Cohen, Jonathan and Tran, John and Catanzaro, Bryan and Shelhamer, Evan, "cudnn: Efficient primitives for deep learning," *arXiv preprint arXiv:1410.0759*, 2014.

[203] Intel, "OneDNN." https://github.com/oneapi-src/oneDNN, 2020.

[204] Google, "XNNPACK." https://github.com/google/XNNPACK, 2020.

[205] Ragan-Kelley, Jonathan and Barnes, Connelly and Adams, Andrew and Paris, Sylvain and Durand, Frédo and Amarasinghe, Saman, "Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," *Acm SIGPLAN Notices*, vol. 48, no. 6, pp. 519–530, 2013.

[206] Tianqi Chen and Thierry Moreau and Ziheng Jiang and Lianmin Zheng and Eddie Yan and Haichen Shen and Meghan Cowan and Leyuan Wang and Yuwei Hu and Luis Ceze and Carlos Guestrin and Arvind Krishnamurthy, "TVM: An Automated End-to-End Optimizing Compiler for Deep Learning," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, (Carlsbad, CA), pp. 578–594, USENIX Association, Oct. 2018.

[207] Mnih, Volodymyr and Kavukcuoglu, Koray and Silver, David and Graves, Alex and Antonoglou, Ioannis and Wierstra, Daan and Riedmiller, Martin, "Playing Atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[208] Silver, David and Huang, Aja and Maddison, Chris J and Guez, Arthur and Sifre, Laurent and Van Den Driessche, George and Schrittwieser, Julian and Antonoglou, Ioannis and Panneershelvam, Veda and Lanctot, Marc and others, "Mastering the game of Go with deep neural networks and tree search," *nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[209] Brauckmann, Alexander and Goens, Andrés and Castrillon, Jeronimo, "A Reinforcement Learning Environment for Polyhedral Optimizations," *arXiv preprint arXiv:2104.13732*, 2021.

[210] Wang, Huanting and Tang, Zhanyong and Zhang, Cheng and Zhao, Jiaqi and Cummins, Chris and Leather, Hugh and Wang, Zheng, "Automating reinforcement learning architecture design for code optimization," in *Proceedings of the ACM SIGPLAN International Conference on Compiler Construction*, pp. 129–143, 2022.

[211] Matthews, Devin A, "High-performance tensor contraction without transposition," *SIAM Journal on Scientific Computing*, vol. 40, no. 1, pp. C1–C24, 2018.

[212] Abdi, Hervé and Williams, Lynne J, "Principal component analysis," *Wiley interdisciplinary reviews: computational statistics*, vol. 2, no. 4, pp. 433–459, 2010.

[213] Radu, Valentin and Tong, Catherine and Bhattacharya, Sourav and Lane, Nicholas D and Mascolo, Cecilia and Marina, Mahesh K and Kawsar, Fahim, "Multimodal deep learning for activity and context recognition," *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, vol. 1, no. 4, pp. 1–27, 2018.

[214] Albooyeh, Marjan and Bertolini, Daniele and Ravanbakhsh, Siamak, "Incidence networks for geometric deep learning," *arXiv preprint arXiv:1905.11460*, 2019.

[215] Grosser, Tobias and Groesslinger, Armin and Lengauer, Christian,

"Polly—performing polyhedral optimizations on a low-level intermediate representation," *Parallel Processing Letters*, vol. 22, no. 04, 2012.

[216] Di Napoli, Edoardo and Fabregat-Traver, Diego and Quintana-Ortí, Gregorio and Bientinesi, Paolo, "Towards an efficient use of the BLAS library for multilinear tensor contractions," *Applied Mathematics and Computation*, vol. 235, pp. 454–468, 2014.

[217] Stallman, Richard M and others, *Using and porting the GNU compiler collection*. Free Software Foundation, 1999.

[218] Goto, Kazushige and Geijn, Robert A van de, "Anatomy of high-performance matrix multiplication," *ACM Transactions on Mathematical Software (TOMS)*, vol. 34, no. 3, pp. 1–25, 2008.

[219] Park, Neungsoo and Liu, Wenheng and Prasanna, Viktor K and Raghavendra, Cauligi, "Efficient matrix multiplication using cache conscious data layouts," in *HPCMO User Group Conference*, 2000.

[220] Smith, Tyler M and Van De Geijn, Robert and Smelyanskiy, Mikhail and Hammond, Jeff R and Van Zee, Field G, "Anatomy of high-performance many-threaded matrix multiplication," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pp. 1049–1059, IEEE, 2014.

[221] Gunnels, John A and Henry, Greg M and Van De Geijn, Robert A, "A family of high-performance matrix multiplication algorithms," in *International Conference on Computational Science*, pp. 51–60, Springer, 2001.

[222] Jia, Zhen and Zlateski, Aleksandar and Durand, Fredo and Li, Kai, "Optimizing N-dimensional, winograd-based convolution for manycore CPUs," in *Proceed-*

ings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 109–123, 2018.

[223] Zlateski, Aleksandar and Jia, Zhen and Li, Kai and Durand, Fredo, "The anatomy of efficient FFT and winograd convolutions on modern CPUs," in *Proceedings of the ACM International Conference on Supercomputing*, pp. 414–424, 2019.

[224] Zlateski, Aleksandar and Seung, H Sebastian, "Compile-time optimized and statically scheduled ND ConvNet primitives for multi-core and many-core (Xeon Phi) CPUs," in *Proceedings of the International Conference on Supercomputing*, pp. 1–10, 2017.

[225] Grosser, Tobias and Zheng, Hongbin and Aloor, Raghesh and Simbürger, Andreas and Größlinger, Armin and Pouchet, Louis-Noël, "Polly-Polyhedral optimization in LLVM," in *Proceedings of the International Workshop on Polyhedral Compilation Techniques (IMPACT)*, vol. 2011, p. 1, 2011.

[226] Leben, Jakob and Tzanetakis, George, "Polyhedral compilation for multi-dimensional stream processing," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 16, no. 3, pp. 1–26, 2019.

[227] Adams, Andrew and Ma, Karima and Anderson, Luke and Baghdadi, Riyadh and Li, Tzu-Mao and Gharbi, Michaël and Steiner, Benoit and Johnson, Steven and Fatahalian, Kayvon and Durand, Frédo and others, "Learning to optimize halide with tree search and random programs," *ACM Transactions on Graphics (TOG)*, vol. 38, no. 4, pp. 1–12, 2019.

[228] Zheng, Lianmin and Jia, Chengfan and Sun, Minmin and Wu, Zhao and

Yu, Cody Hao and Haj-Ali, Ameer and Wang, Yida and Yang, Jun and Zhuo, Danyang and Sen, Koushik and others, "Ansor: Generating {High-Performance} tensor programs for deep learning," in *14th USENIX symposium on operating systems design and implementation (OSDI 20)*, pp. 863–879, 2020.

[229] Steiner, Benoit and Cummins, Chris and He, Horace and Leather, Hugh, "Value learning for throughput optimization of deep learning workloads," *Proceedings of Machine Learning and Systems*, vol. 3, pp. 323–334, 2021.

[230] ARM, "Cortex-A57 Software Optimization Guide," *ARM Corporation*, 2016.

[231] Intel, "Intel 64 and ia-32 architectures optimization reference manual," *Intel Corporation*, 2014.

[232] Rastello, Fabrice and Ponnuswany, Sadayappan and van Amstel, Duco, *A Tiling Perspective for Register Optimization*. PhD thesis, Inria, 2014.

[233] Van Zee, Field G and Van De Geijn, Robert A, "BLIS: A framework for rapidly instantiating BLAS functionality," *ACM Transactions on Mathematical Software (TOMS)*, vol. 41, no. 3, pp. 1–33, 2015.

[234] Barrachina, Sergio and Castillo, Maribel and Igual, Francisco D and Mayo, Rafael and Quintana-Orti, Enrique S, "Evaluation and tuning of the level 3 CUBLAS for graphics processors," in *2008 IEEE International Symposium on Parallel and Distributed Processing*, pp. 1–8, IEEE, 2008.

[235] Wang, Endong and Zhang, Qing and Shen, Bo and Zhang, Guangyong and Lu, Xiaowei and Wu, Qing and Wang, Yajuan, "Intel math kernel library," in *High-Performance Computing on the Intel® Xeon Phi™*, pp. 167–188, Springer, 2014.

[236] Heinecke, Alexander and Henry, Greg and Hutchinson, Maxwell and Pabst, Hans, "LIBXSMM: accelerating small matrix multiplications by runtime code generation," in *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 981–991, IEEE, 2016.

[237] Heinecke, Alexander and Pabst, Hans and Henry, Greg, "Libxsmm: A high performance library for small matrix multiplications," *Poster and Extended Abstract Presented at SC*, 2015.

[238] Khudia, Daya and Huang, Jianyu and Basu, Protonu and Deng, Summer and Liu, Haixin and Park, Jongsoo and Smelyanskiy, Mikhail, "FBGEMM: Enabling High-Performance Low-Precision Deep Learning Inference," *arXiv preprint arXiv:2101.05615*, 2021.

[239] Zlateski, Aleksandar and Jia, Zhen and Li, Kai and Durand, Fredo, "The anatomy of efficient FFT and winograd convolutions on modern CPUs," in *Proceedings of the ACM International Conference on Supercomputing*, ICS '19, (New York, NY, USA), p. 414–424, Association for Computing Machinery, 2019.

[240] Elsen, Erich and Dukhan, Marat and Gale, Trevor and Simonyan, Karen, "Fast sparse convnets," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 14629–14638, 2020.

[241] Heinecke, Alexander and Breuer, Alexander and Bader, Michael and Dubey, Pradeep, "High order seismic simulations on the Intel Xeon Phi processor (Knights Landing)," in *International Conference on High Performance Computing*, pp. 343–362, Springer, 2016.

[242] Einstein, Albert, "Die grundlage der allgemeinen relativitätstheorie," in *Das Relativitätsprinzip*, pp. 81–124, Springer, 1923.

[243] Alexander Rush, "Tensor Considered Harmful." http://nlp.seas.harvard.edu/NamedTensor, 2018 (accessed August 26, 2020).

[244] Oliphant, Travis E, *A guide to NumPy*, vol. 1. Trelgol Publishing USA, 2006.

[245] A. Qasem, G. Jin, and J. Mellor-Crummey, "Improving performance with integrated program transformations," 2004.

[246] Y. Zhao and K. Kennedy, "Scalarization using loop alignment and loop skewing," *The Journal of Supercomputing*, vol. 31, pp. 5–46, 2005.

[247] Cummins, Chris and Wasti, Bram and Guo, Jiadong and Cui, Brandon and Ansel, Jason and Gomez, Sahir and Jain, Somya and Liu, Jia and Teytaud, Olivier and Steiner, Benoit and others, "CompilerGym: robust, performant compiler optimization environments for AI research," in *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pp. 92–105, IEEE, 2022.

[248] Liang, Eric and Liaw, Richard and Nishihara, Robert and Moritz, Philipp and Fox, Roy and Goldberg, Ken and Gonzalez, Joseph and Jordan, Michael and Stoica, Ion, "RLlib: Abstractions for distributed reinforcement learning," in *Proceedings on Machine Learning Research*, pp. 3053–3062, PMLR, 2018.

[249] Kanervisto, Anssi and Scheller, Christian and Hautamäki, Ville, "Action space shaping in deep reinforcement learning," in *2020 IEEE Conference on Games (CoG)*, pp. 479–486, IEEE, 2020.

[250] Mnih, Volodymyr and Kavukcuoglu, Koray and Silver, David and Graves, Alex and Antonoglou, Ioannis and Wierstra, Daan and Riedmiller, Martin, "Playing Atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[251] Horgan, Dan and Quan, John and Budden, David and Barth-Maron, Gabriel and Hessel, Matteo and Van Hasselt, Hado and Silver, David, "Distributed prioritized experience replay," *arXiv preprint arXiv:1803.00933*, 2018.

[252] Schulman, John and Wolski, Filip and Dhariwal, Prafulla and Radford, Alec and Klimov, Oleg, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[253] Mnih, Volodymyr and Badia, Adria Puigdomenech and Mirza, Mehdi and Graves, Alex and Lillicrap, Timothy and Harley, Tim and Silver, David and Kavukcuoglu, Koray, "Asynchronous methods for deep reinforcement learning," in *Proceedings on Machine Learning Research*, pp. 1928–1937, PMLR, 2016.

[254] Espeholt, Lasse and Soyer, Hubert and Munos, Remi and Simonyan, Karen and Mnih, Vlad and Ward, Tom and Doron, Yotam and Firoiu, Vlad and Harley, Tim and Dunning, Iain and others, "Impala: Scalable distributed deep-RL with importance weighted actor-learner architectures," in *Proceedings on Machine Learning Research*, pp. 1407–1416, PMLR, 2018.

[255] Ashouri, Amir H and Killian, William and Cavazos, John and Palermo, Gianluca and Silvano, Cristina, "A survey on compiler autotuning using machine learning," *ACM Computing Surveys (CSUR)*, vol. 51, no. 5, pp. 1–42, 2018.

[256] Dolan, Elizabeth D and Moré, Jorge J, "Benchmarking optimization software with performance profiles," *Mathematical programming*, vol. 91, pp. 201–213,

2002.

[257] TVM documentation version(0.11.dev0), "How to optimize GEMM on CPU." https://tvm.apache.org/docs/how$_to$/$optimize_operators$/opt_gemm.html.[$Online$; $accessed$28 -

[258] Zheng, Lianmin and Liu, Ruochen and Shao, Junru and Chen, Tianqi and Gonzalez, Joseph E and Stoica, Ion and Ali, Ameer Haj, "Tenset: A large-scale program performance dataset for learned tensor compilers," in *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*, 2021.

[259] Zheng, Size and Liang, Yun and Wang, Shuo and Chen, Renze and Sheng, Kaiwen, "Flextensor: An automatic schedule exploration and optimization framework for tensor computation on heterogeneous system," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 859–873, 2020.

[260] Abrams, Philip Samuel, "An APL machine.," tech. rep., Stanford Linear Accelerator Center, California, 1970.

[261] Van Der Walt, Stefan and Colbert, S Chris and Varoquaux, Gael, "The NumPy array: a structure for efficient numerical computation," *Computing in science & engineering*, vol. 13, no. 2, pp. 22–30, 2011.

[262] Bader, Brett W and Kolda, Tamara G, "Algorithm 862: MATLAB tensor classes for fast algorithm prototyping," *ACM Transactions on Mathematical Software (TOMS)*, vol. 32, no. 4, pp. 635–653, 2006.

[263] Intel, "MKL Developer Reference." https://software.intel.com/content/www/us/en/develop/c
developer-reference-c/top.html, 2020.

[264] Epifanovsky, Evgeny and Wormit, Michael and Kuś, Tomasz and Landau, Arie and Zuev, Dmitry and Khistyaev, Kirill and Manohar, Prashant and Kaliman, Ilya and Dreuw, Andreas and Krylov, Anna I, "New implementation of high-level correlated methods using a general block tensor library for high-performance electronic structure calculations," 2013.

[265] Solomonik, Edgar and Matthews, Devin and Hammond, Jeff R and Stanton, John F and Demmel, James, "A massively parallel tensor contraction framework for coupled-cluster computations," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3176–3190, 2014.

[266] Frigo, Matteo and Johnson, Steven G, "FFTW: An adaptive software architecture for the FFT," in *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, vol. 3, pp. 1381–1384, IEEE, 1998.

[267] Ansel, Jason and Kamil, Shoaib and Veeramachaneni, Kalyan and Ragan-Kelley, Jonathan and Bosboom, Jeffrey and O'Reilly, Una-May and Amarasinghe, Saman, "Opentuner: An extensible framework for program autotuning," in *Proceedings of the International Conference on Parallel Architectures and Compilation*, pp. 303–316, 2014.

[268] Rotem, Nadav and Fix, Jordan and Abdulrasool, Saleem and Catron, Garret and Deng, Summer and Dzhabarov, Roman and Gibson, Nick and Hegeman, James and Lele, Meghan and Levenstein, Roman and others, "Glow: Graph lowering compiler techniques for neural networks," *arXiv preprint arXiv:1805.00907*, 2018.

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

[270] Chen, Tianqi and Zheng, Lianmin and Yan, Eddie and Jiang, Ziheng and Moreau, Thierry and Ceze, Luis and Guestrin, Carlos and Krishnamurthy, Arvind, "Learning to optimize tensor programs," in *Advances in Neural Information Processing Systems*, pp. 3389–3400, 2018.

[271] Verdoolaege, Sven, "isl: An integer set library for the polyhedral model," in *International Congress on Mathematical Software*, pp. 299–302, Springer, 2010.

[272] Bagnara, Roberto and Hill, Patricia M and Zaffanella, Enea, "The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems," *Science of Computer Programming*, vol. 72, no. 1-2, pp. 3–21, 2008.

[273] Vasilache, Nicolas and Zinenko, Oleksandr and Theodoridis, Theodoros and Goyal, Priya and DeVito, Zachary and Moses, William S and Verdoolaege, Sven and Adams, Andrew and Cohen, Albert, "Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions," *arXiv preprint arXiv:1802.04730*, 2018.

[274] Trofin, Mircea and Qian, Yundi and Brevdo, Eugene and Lin, Zinan and Choromanski, Krzysztof and Li, David, "Mlgo: a machine learning guided compiler

optimizations framework," *arXiv preprint arXiv:2101.04808*, 2021.

[275] B. Wasti, J. P. Cambronero, B. Steiner, H. Leather, and A. Zlateski, "Loopstack: a lightweight tensor algebra compiler stack," *arXiv preprint arXiv:2205.00618*, 2022.

[276] OpenAI, "GPT-4 Technical Report," *arXiv:2303.08774*, 2023.

[277] Chowdhery, Aakanksha and Narang, Sharan and Devlin, Jacob and Bosma, Maarten and Mishra, Gaurav and Roberts, Adam and Barham, Paul and Chung, Hyung Won and Sutton, Charles and Gehrmann, Sebastian and others, "Palm: Scaling language modeling with pathways," *Journal of Machine Learning Research*, vol. 24, no. 240, pp. 1–113, 2023.

[278] Fried, Daniel and Aghajanyan, Armen and Lin, Jessy and Wang, Sida and Wallace, Eric and Shi, Freda and Zhong, Ruiqi and Yih, Wen-tau and Zettlemoyer, Luke and Lewis, Mike, "InCoder: A Generative Model for Code Infilling and Synthesis," *arXiv:2204.05999*, 2023.

[279] Suriya Gunasekar and Yi Zhang and Jyoti Aneja and Caio César Teodoro Mendes and Allie Del Giorno and Sivakanth Gopi and Mojan Javaheripi and Piero Kauffmann and Gustavo de Rosa and Olli Saarikivi and Adil Salim and Shital Shah and Harkirat Singh Behl and Xin Wang and Sébastien Bubeck and Ronen Eldan and Adam Tauman Kalai and Yin Tat Lee and Yuanzhi Li, "Textbooks Are All You Need," *arXiv:2306.11644*, 2023.

[280] Chen, Mark and Tworek, Jerry and Jun, Heewoo and Yuan, Qiming and Pinto, Henrique Ponde de Oliveira and Kaplan, Jared and Edwards, Harri and Burda, Yuri and Joseph, Nicholas and Brockman, Greg and Ray, Alex and Puri, Raul

and Krueger, Gretchen and Petrov, Michael and Khlaaf, Heidy and Sastry, Girish and Mishkin, Pamela and Chan, Brooke and Gray, Scott and Ryder, Nick and Pavlov, Mikhail and Power, Alethea and Kaiser, Lukasz and Bavarian, Mohammad and Winter, Clemens and Tillet, Philippe and Such, Felipe Petroski and Cummings, Dave and Plappert, Matthias and Chantzis, Fotios and Barnes, Elizabeth and Herbert-Voss, Ariel and Guss, William Hebgen and Nichol, Alex and Paino, Alex and Tezak, Nikolas and Tang, Jie and Babuschkin, Igor and Balaji, Suchir and Jain, Shantanu and Saunders, William and Hesse, Christopher and Carr, Andrew N. and Leike, Jan and Achiam, Josh and Misra, Vedant and Morikawa, Evan and Radford, Alec and Knight, Matthew and Brundage, Miles and Murati, Mira and Mayer, Katie and Welinder, Peter and McGrew, Bob and Amodei, Dario and McCandlish, Sam and Sutskever, Ilya and Zaremba, Wojciech, "Evaluating Large Language Models Trained on Code," *arXiv:2107.03374*, 2021.

[281] Roziere, Baptiste and Lachaux, Marie-Anne and Chanussot, Lowik and Lample, Guillaume, "Unsupervised translation of programming languages," *Advances in Neural Information Processing Systems*, vol. 33, pp. 20601–20611, 2020.

[282] GitHub, "Copilot." https://copilot.github.com/.

[283] Zheng Wang and Michael O'Boyle, "Machine Learning in Compiler Optimisation," *arXiv:1805.03441*, 2018.

[284] Liang, Youwei and Stone, Kevin and Shameli, Ali and Cummins, Chris and Elhoushi, Mostafa and Guo, Jiadong and Steiner, Benoit and Yang, Xiaomeng and Xie, Pengtao and Leather, Hugh and Tian, Yuandong, "Learning Compiler

Pass Orders using Coreset and Normalized Value Prediction," in *Proceedings of International Conference on Machine Learning (ICML)*, 2023.

[285] Nicholas Asher and Swarnadeep Bhar and Akshay Chaturvedi and Julie Hunter and Soumya Paul, "Limits for Learning with Language Models," *arXiv:2306.12213*, 2023.

[286] Jing Qian and Hong Wang and Zekun Li and Shiyang Li and Xifeng Yan, "Limitations of Language Models in Arithmetic and Symbolic Induction," *arXiv:2208.05051*, 2022.

[287] Touvron, Hugo and Martin, Louis and Stone, Kevin and Albert, Peter and Almahairi, Amjad and Babaei, Yasmine and Bashlykov, Nikolay and Batra, Soumya and Bhargava, Prajjwal and Bhosale, Shruti and others, "Llama 2: Open Foundation and Fine-Tuned Chat Models," *arXiv:2307.09288*, 2023.

[288] Fursin, GG and O'Boyle, Michael FP and Knijnenburg, Peter MW, "Evaluating iterative compilation," in *Languages and Compilers for Parallel Computing: 15th Workshop, LCPC 2002, College Park, MD, USA, July 25-27, 2002*, pp. 362–376, Springer, 2005.

[289] Kocetkov, Denis and Li, Raymond and Allal, Loubna Ben and Li, Jia and Mou, Chenghao and Ferrandis, Carlos Muñoz and Jernite, Yacine and Mitchell, Margaret and Hughes, Sean and Wolf, Thomas and others, "The Stack: 3TB of Permissively Licensed Source Code," *arXiv:2211.15533*, 2022.

[290] Gao, Leo and Biderman, Stella and Black, Sid and Golding, Laurence and Hoppe, Travis and Foster, Charles and Phang, Jason and He, Horace and Thite,

Anish and Nabeshima, Noa and others, "The Pile: An 800GB Dataset of Diverse Text for Language Modeling," *arXiv:2101.00027*, 2020.

[291] Husain, Hamel and Wu, Ho-Hsiang and Gazit, Tiferet and Allamanis, Miltiadis and Brockschmidt, Marc, "CodeSearchNet Challenge: Evaluating the State of Semantic Code Search," *arXiv:1909.09436*, 2019.

[292] Armengol-Estapé, Jordi and Woodruff, Jackson and Cummins, Chris and O'Boyle, Michael FP, "SLaDe: A Portable Small Language Model Decompiler for Optimized Assembler," *arXiv:2305.12520*, 2023.

[293] Armengol-Estapé, Jordi and O'Boyle, Michael FP, "Learning C to x86 Translation: An Experiment in Neural Compilation," *arXiv:2108.07639*, 2021.

[294] Gage, Philip, "A New Algorithm for Data Compression," *C Users Journal*, vol. 12, no. 2, 1994.

[295] Loshchilov, Ilya and Hutter, Frank, "Decoupled Weight Decay Regularization," *arXiv:1711.05101*, 2017.

[296] Papineni, Kishore and Roukos, Salim and Ward, Todd and Zhu, Wei-Jing, "Bleu: a method for automatic evaluation of machine translation," in *Proceedings of the Annual Meeting of the Association for Computational Linguistics*, pp. 311–318, 2002.

[297] Fadel, Ali and Musleh, Husam and Tuffaha, Ibraheem and Al-Ayyoub, Mahmoud and Jararweh, Yaser and Benkhelifa, Elhadj and Rosso, Paolo, "Overview of the PAN@ FIRE 2020 task on the authorship identification of SOurce COde," in *Proceedings of the Annual Meeting of the Forum for Information Retrieval Evaluation*, pp. 4–8, 2020.

[298] Mou, Lili and Li, Ge and Zhang, Lu and Wang, Tao and Jin, Zhi, "Convolutional neural networks over tree structures for programming language processing," in *Proceedings of the AAAI conference on Artificial Intelligence*, vol. 30, 2016.

[299] Livinskii, Vsevolod and Babokin, Dmitry and Regehr, John, "Random testing for C and C++ compilers with YARPGen," *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 1–25, 2020.

[300] Armengol-Estapé, Jordi and Woodruff, Jackson and Brauckmann, Alexander and Magalhães, José Wesley de Souza and O'Boyle, Michael FP, "ExeBench: an ML-scale dataset of executable C functions," in *Proceedings of the ACM SIGPLAN International Symposium on Machine Programming*, pp. 50–59, 2022.

[301] Haj-Ali, Ameer and Huang, Qijing Jenny and Xiang, John and Moses, William and Asanovic, Krste and Wawrzynek, John and Stoica, Ion, "Autophase: Juggling hls phase orderings in random forests with deep reinforcement learning," *Proceedings of Machine Learning and Systems*, vol. 2, pp. 70–81, 2020.

[302] Michal Paszkowski, "LLVM Canon." https://github.com/michalpaszkowski/LLVM-Canon.

[303] McKeeman, William M, "Differential Testing for Software," *Digital Technical Journal*, vol. 10, no. 1, 1998.

[304] Kisuki, Toru and Knijnenburg, Peter MW and O'Boyle, Michael FP, "Combined selection of tile sizes and unroll factors using iterative compilation," in *Proceedings International Conference on Parallel Architectures and Compilation Techniques*, pp. 237–246, IEEE, 2000.

[305] Ogilvie, William F and Petoumenos, Pavlos and Wang, Zheng and Leather, Hugh, "Minimizing the cost of iterative compilation with active learning," in *International Symposium on Code Generation and Optimization (CGO)*, pp. 245–256, IEEE, 2017.

[306] Ashouri, Amir H and Elhoushi, Mostafa and Hua, Yuzhe and Wang, Xiang and Manzoor, Muhammad Asif and Chan, Bryan and Gao, Yaoqing, "MLGOPerf: An ML Guided Inliner to Optimize Performance," *arXiv:2207.08389*, 2022.

[307] Cummins, Chris and Petoumenos, Pavlos and Wang, Zheng and Leather, Hugh, "End-to-end deep learning of optimization heuristics," in *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pp. 219–232, IEEE, 2017.

[308] Phothilimthana, Phitchaya Mangpo and Sabne, Amit and Sarda, Nikhil and Murthy, Karthik Srinivasa and Zhou, Yanqi and Angermueller, Christof and Burrows, Mike and Roy, Sudip and Mandke, Ketan and Farahani, Rezsa and others, "A flexible approach to autotuning multi-pass machine learning compilers," in *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pp. 1–16, IEEE, 2021.

[309] Hosseini, Iman and Dolan-Gavitt, Brendan, "Beyond the C: Retargetable Decompilation using Neural Machine Translation," *arXiv:2212.08950*, 2022.

[310] Xia, Chunqiu Steven and Wei, Yuxiang and Zhang, Lingming, "Automated Program Repair in the Era of Large Pre-Trained Language Models," in *Proceedings of the International Conference on Software Engineering*, ICSE '23, p. 1482–1494, IEEE Press, 2023.

[311] Touvron, Hugo and Lavril, Thibaut and Izacard, Gautier and Martinet, Xavier and Lachaux, Marie-Anne and Lacroix, Timothée and Rozière, Baptiste and Goyal, Naman and Hambro, Eric and Azhar, Faisal and others, "Llama: Open and efficient foundation language models," *arXiv preprint arXiv:2302.13971*, 2023.

[312] Ding, Jiayu and Ma, Shuming and Dong, Li and Zhang, Xingxing and Huang, Shaohan and Wang, Wenhui and Wei, Furu, "LongNet: Scaling Transformers to 1,000,000,000 Tokens," *arXiv:2307.02486*, 2023.

[313] Chen, Shouyuan and Wong, Sherman and Chen, Liangjian and Tian, Yuandong, "Extending Context Window of Large Language Models via Positional Interpolation," *arXiv:2306.15595*, 2023.

[314] Sun, Yutao and Dong, Li and Patra, Barun and Ma, Shuming and Huang, Shaohan and Benhaim, Alon and Chaudhary, Vishrav and Song, Xia and Wei, Furu, "A Length-Extrapolatable Transformer," *arXiv:2212.10554*, 2022.

[315] Wei, Jason and Wang, Xuezhi and Schuurmans, Dale and Bosma, Maarten and Xia, Fei and Chi, Ed and Le, Quoc V and Zhou, Denny and others, "Chain-of-thought prompting elicits reasoning in large language models," *Advances in Neural Information Processing Systems*, vol. 35, pp. 24824–24837, 2022.

[316] Gao, Luyu and Madaan, Aman and Zhou, Shuyan and Alon, Uri and Liu, Pengfei and Yang, Yiming and Callan, Jamie and Neubig, Graham, "Pal: Program-aided language models," in *Proceedings on Machine Learning Research*, pp. 10764–10799, PMLR, 2023.

[317] Cobbe, Karl and Kosaraju, Vineet and Bavarian, Mohammad and Chen, Mark

and Jun, Heewoo and Kaiser, Lukasz and Plappert, Matthias and Tworek, Jerry and Hilton, Jacob and Nakano, Reiichiro and others, "Training Verifiers to Solve Math Word Problems," *arXiv:2110.14168*, 2021.

[318] Xiao, Guangxuan and Lin, Ji and Seznec, Mickael and Wu, Hao and Demouth, Julien and Han, Song, "Smoothquant: Accurate and efficient post-training quantization for large language models," in *Proceedings on Machine Learning Research*, pp. 38087–38099, PMLR, 2023.

[319] Ackley, David H and Hinton, Geoffrey E and Sejnowski, Terrence J, "A learning algorithm for Boltzmann machines," *Cognitive science*, vol. 9, no. 1, pp. 147–169, 1985.

[320] Ficler, Jessica and Goldberg, Yoav, "Controlling linguistic style aspects in neural language generation," *arXiv preprint arXiv:1707.02633*, 2017.

[321] Fan, Angela and Lewis, Mike and Dauphin, Yann, "Hierarchical neural story generation," *arXiv preprint arXiv:1805.04833*, 2018.

[322] Caccia, Massimo and Caccia, Lucas and Fedus, William and Larochelle, Hugo and Pineau, Joelle and Charlin, Laurent, "Language gans falling short," *arXiv preprint arXiv:1811.02549*, 2018.

[323] Ahmad, Baleegh and Thakur, Shailja and Tan, Benjamin and Karri, Ramesh and Pearce, Hammond, "Fixing Hardware Security Bugs with Large Language Models," *arXiv preprint arXiv:2302.01215*, 2023.

[324] Da Silva, Anderson Faustino and Kind, Bruno Conde and de Souza Magalhães, José Wesley and Rocha, Jerônimo Nunes and Guimaraes, Breno Campos Fer-

reira and Pereira, Fernando Magno Quinão, "Anghabench: A suite with one million compilable c benchmarks for code-size reduction," in *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pp. 378–390, IEEE, 2021.

[325] Yang, Chengrun and Wang, Xuezhi and Lu, Yifeng and Liu, Hanxiao and Le, Quoc V and Zhou, Denny and Chen, Xinyun, "Large language models as optimizers," *arXiv preprint arXiv:2309.03409*, 2023.

[326] Ravfogel, Shauli and Goldberg, Yoav and Goldberger, Jacob, "Conformal Nucleus Sampling," *arXiv preprint arXiv:2305.02633*, 2023.

[327] Li, Dong and Jin, Ruoming and Gao, Jing and Liu, Zhi, "On sampling top-k recommendation evaluation," in *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 2114–2124, 2020.

[328] Xie, Yuxi and Kawaguchi, Kenji and Zhao, Yiran and Zhao, Xu and Kan, Min-Yen and He, Junxian and Xie, Qizhe, "Self-Evaluation Guided Beam Search for Reasoning," in *Thirty-seventh Conference on Neural Information Processing Systems*, 2023.

[329] Kool, Wouter and Van Hoof, Herke and Welling, Max, "Stochastic beams and where to find them: The gumbel-top-k trick for sampling sequences without replacement," in *Proceedings on Machine Learning Research*, pp. 3499–3508, PMLR, 2019.

[330] Willard, Brandon T and Louf, Rémi, "Efficient guided generation for large language models," *arXiv e-prints*, 2023.

[331] Radford, Alec and Wu, Jeffrey and Child, Rewon and Luan, David and Amodei, Dario and Sutskever, Ilya and others, "Language models are unsupervised multitask learners," *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.

[332] Holtzman, Ari and Buys, Jan and Forbes, Maxwell and Bosselut, Antoine and Golub, David and Choi, Yejin, "Learning to write with cooperative discriminators," *arXiv preprint arXiv:1805.06087*, 2018.

[333] Vijayakumar, Ashwin K and Cogswell, Michael and Selvaraju, Ramprasath R and Sun, Qing and Lee, Stefan and Crandall, David and Batra, Dhruv, "Diverse beam search: Decoding diverse solutions from neural sequence models," *arXiv preprint arXiv:1610.02424*, 2016.

[334] Batra, Dhruv and Yadollahpour, Payman and Guzman-Rivera, Abner and Shakhnarovich, Gregory, "Diverse m-best solutions in markov random fields," in *Computer Vision–ECCV 2012: 12th European Conference on Computer Vision, Florence, Italy, October 7-13, 2012, Proceedings, Part V 12*, pp. 1–16, Springer, 2012.

[335] Gumbel, Emil Julius, "Statistical theory of extreme valuse and some practical applications," *Nat. Bur. Standards Appl. Math. Ser. 33*, 1954.