LLMs for Programming Tasks

From User Requirements to Program Binaries

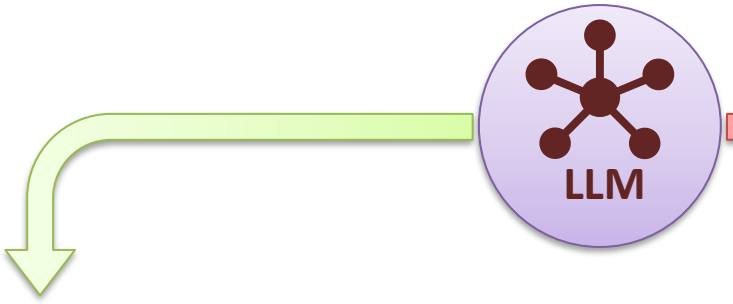
Giuseppe Tagliavini

Department of Computer Science and Engineering (DISI)

Outline

- **Introduction**
- LLMs for code generation
- LLMs for compiler assistance
- Conclusion

The journey from user requirements down to binary code

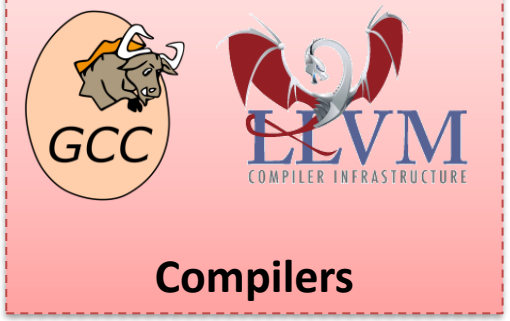


Natural language

Compute the inner product between 2-element vectors

Programming language

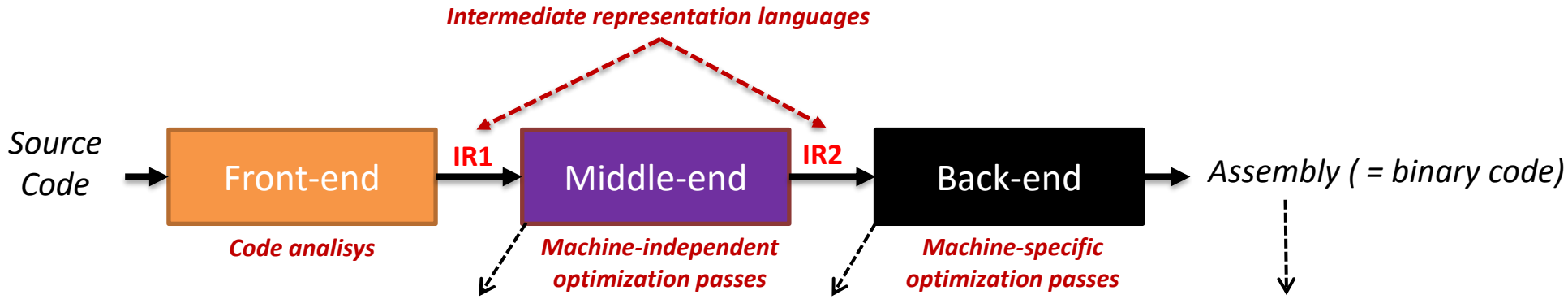

```
int inner_product(int a[2], int b[2]) {
    int32_t result;
    result = a[0] * b[0]+ a[1] * b[1];
    return result;
}
```



Binary code

```
01010100 01101000 01101001 01110011
00100000 01101001 01110011 00100000
01110100 01101000 01100101 00100000
01110100 01110101 01110100 01101111
01110010 01101001 01100001 01101100
00100000 01110100 01101111 00100000
01101100 01100101 01100001 01110010
01101110 00100000 01100010 01101001
01101110 01100001 01110010 01111001
00101110 00100000 01001001 00100000
01101000 01101111 01110000 01100101
00100000 01111001 01101111 01110101
00100000 01100101 01101110 01101010
01101111 01111001 00100000 01101001
01110100 00100001
```

High-level design of modern compilers



- **Int**
  as

```
define i32 @inner_product(i32* %a, i
entry:
  %0 = getelementptr inbounds i32, i
  %1 = load i32, i32* %0, align 4
  %2 = getelementptr inbounds i32, i
  %3 = load i32, i32* %2, align 4
  %mul = mul nsw i32 %1, %3
  %4 = getelementptr inbounds i32, i
  %5 = load i32, i32* %4, align 4
  %6 = getelementptr inbounds i32, i
  %
  %
  %
  ret i32 %add
}
```

```
name: inner_product
body: |
  bb.0:
    %1 = GPR32: $a0
    %2 = GPR32: $a1
    %3 = LW $a3, 0($a0)
    %4 = LW $a4, 0($a1)
    %5 = MUL $a3, $a3, $a4
    %6 = LW $a5, 4($a0)
    %7 = LW $a6, 4($a1)
    %8 = MUL $a6, $a5, $a6
```

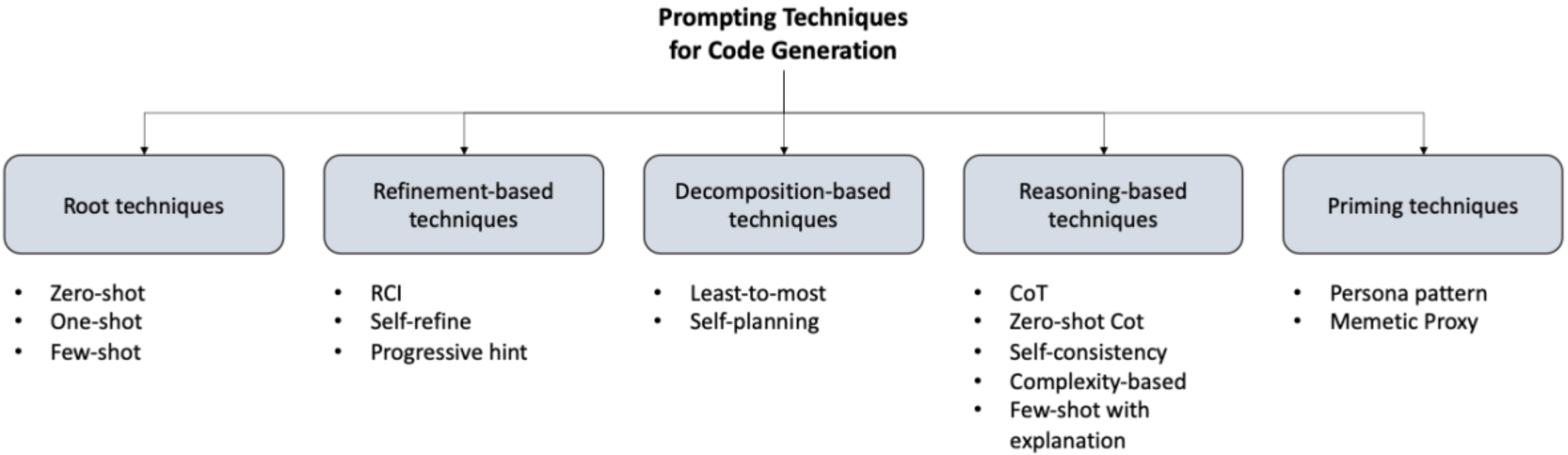
```
inner_product:
    lw a3, 0(a0)
    lw a4, 0(a1)
    mul a3, a3, a4
    lw a5, 4(a0)
    lw a6, 4(a1)
    mul a6, a5, a6
    add a0, a3, a6
    ret
```

What do we really mean for «optimization»?
From a compiler perspective, optimizing means transforming one program representation into another

Outline

- Introduction
- **LLMs for code generation**
- LLMs for compiler assistance
- Conclusion

Classification of prompting techniques for code generation

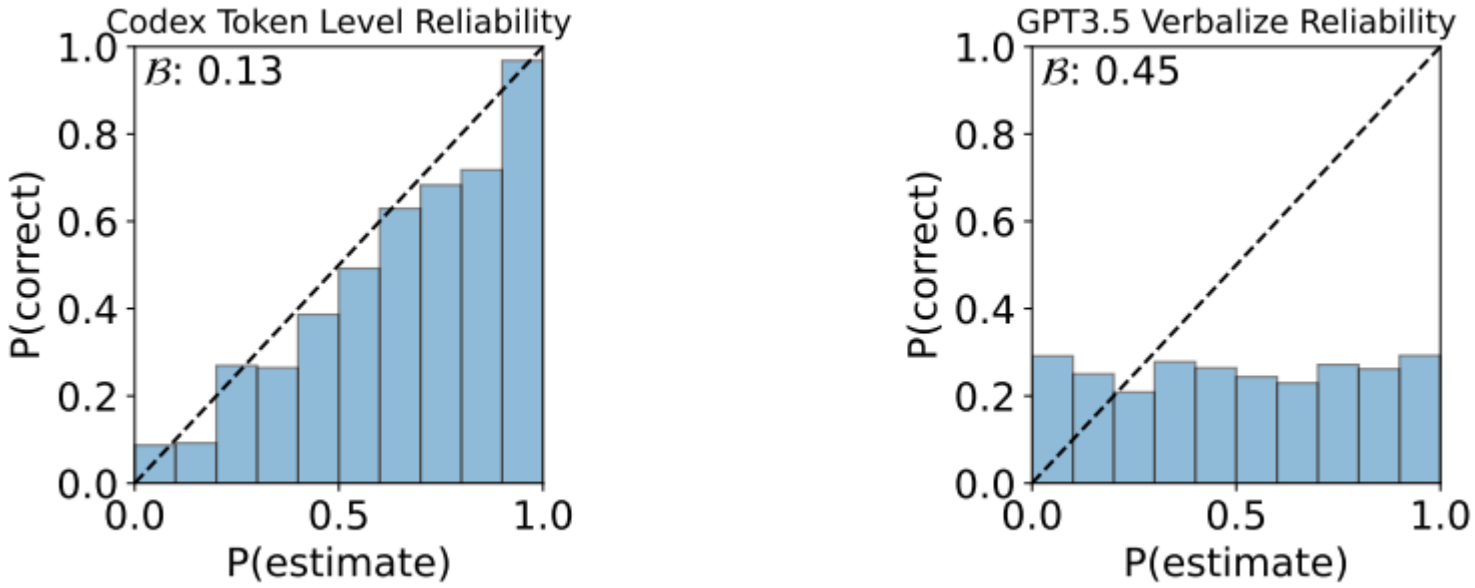


Tony, Catherine, et al. "Prompting Techniques for Secure Code Generation: A Systematic Investigation." *arXiv preprint arXiv:2407.07064* (July 2024).
Link: https://arxiv.org/pdf/2407.07064

Challenges from LLM-generated code (1/4)

Correctness → LLMs may generate code that is syntactically correct but contains bugs

- ❑ We could use the **confidence measure** of GPT models, but it is not reliable by design (e.g., confidence > 90% → 52% test passed)
- ❑ Current research trends → introducing (post-training) techniques to **calibrate the confidence measure** for code generation



Spiess, Claudio, et al. "Calibration and correctness of language models for code." *arXiv preprint arXiv:2402.02047* (August 2024).
Link: https://arxiv.org/abs/2402.02047

Challenges from LLM-generated code (2/4)

Security → LLM-generated code may contain vulnerabilities

- ❑ In recent LLMs, this issue can be solved injecting the domain knowledge using **prompting techniques**
- ❑ *65% of the generated code tagged as insecure by security engineers* → code security close to 100% after **manual guidance of a skilled engineer**

Required prompts to ensure Input Validation								
	ChatGPT		Copilot		CodeWhisperer		CodeLLama	
	JS	Python	JS	Python	JS	Python	JS	Python
SQL Injection	-	-	-	Zero	1-Shot	1-Shot	-	1-Shot
OS Command Injection	-	-	-	-	-	2-Shot	-	-
Input Validation	Zero	Zero	-	-	CoT	1-Shot	-	Zero
Path Traversal	1-Shot	Zero	2-Shot	1-Shot	2-Shot	CoT	1-Shot	1-Shot

Goetz, Stefan, and Andreas Schaad. "You still have to study“ - On the Security of LLM generated code." *arXiv preprint arXiv:2408.07106* (August 2024).
Link: https://arxiv.org/abs/2408.07106

Challenges from LLM-generated code (3/4)

Data privacy →LLM could leak sensitive information if trained on datasets including proprietary or confidential code

- ❑ **Zero-knowledge proofs**
  - Raeini, Mohammad. "Privacy-preserving large language models (PPLLMs)." Available at SSRN 4512071 (2023)
- ❑ **Differential privacy**
  - Sebastian, Glorin. "Privacy and data protection in ChatGPT and other AI Chatbots: strategies for securing user information." International Journal of Security and Privacy in Pervasive Computing (IJSPPC) 15.1 (2023): 1-14.
  - Li, Yansong, Zhixing Tan, and Yang Liu. "Privacy-preserving prompt tuning for large language model services." arXiv preprint arXiv:2305.06212 (2023).
- ❑ **Federated learning**
  - Kuang, Weirui, et al. "FederatedScope-LLM: A comprehensive package for fine-tuning large language models in federated learning." Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining. 2024.
  - Jiang, Jingang, Xiangyang Liu, and Chenyou Fan. "Low-parameter federated learning with large language models." arXiv preprint arXiv:2307.13896 (2023).
  - Fan, Tao, et al. “FATE/LLM: An industrial grade federated learning framework for large language models." arXiv preprint arXiv:2310.10049 (2023).

Challenges from LLM-generated code (4/4)

Accountability → Who is responsible for the LLM-generated code?

Common guidelines:

- ✓ At least **one programmer must take responsibility** for the code generated by the LLM
- ✓ The **model used and its version must be reported**, along with details of how much of the content was automatically generated
- ✓ The portions of code that are LLM-generated must be **annotated**

Outline

- Introduction
- LLMs for code generation
- **LLMs for compiler assistance**
- Conclusion

LLM supporting compilers: Phase Ordering Problem (1/2)

- **Phase ordering problem** → determining the optimal sequence of compiler optimization passes
- LLM-based solution by Meta AI [1]
- Model architecture: Llama 2 7B
- Dataset: 1,000,000 LLVM IR functions (373M tokens)
- Training: for 30,000 steps on 64 V100s for (620 GPU days)

- Comparison with the state-of-the-art:

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

[1] Cummins, Chris, et al. "Large language models for compiler optimization." arXiv preprint arXiv:2309.07062 (2023).
[2] Haj-Ali, Ameer, et al. "AutoPhase: Juggling HLS Phase Orderings in Random Forests with Deep Reinforcement Learning." Proceedings of Machine Learning and Systems 2 (2020): 70-81.
[3] Liang, Youwei, et al. "Learning Compiler Pass Orders using Coreset and Normalized Value Prediction." *International Conference on Machine Learning*. PMLR, 2023.

LLM supporting compilers: Phase Ordering Problem (2/2)

```
define i32 @f1(i8 %0) {
 %2 = alloca i32, align 4
 %3 = alloca i8, align 1
 store i8 %0, i8* %3, align 1
 %4 = load i8, i8* %3, align 1
 %5 = zext i8 %4 to i32
 %6 = icmp sge i32 %5, 65
 br i1 %6, label %7, label %15

7:
 %8 = load i8, i8* %3, align 1
 %9 = zext i8 %8 to i32
 %10 = icmp sle i32 %9, 90
 br i1 %10, label %11, label %15

11:
 %12 = load i8, i8* %3, align 1
 %13 = zext i8 %12 to i32

<snip 21 lines...>

33:
 %34 = load i32, i32* %2, align 4
 ret i32 %34
}
```

Input → 39 instructions

```
define i32 @f1(i8 %0) {
 %2 = zext i8 %0 to i32
 %.off = add i8 %0, 191
 %3 = icmp ult i8 %.off, 26
 br i1 %3, label %4, label %6

4:
 %5 = add nsw i32 %2, 191
 br label %10

6:
 %.reload16.off = add nsw i32 %2, 159
 %7 = icmp ult i32 %.reload16.off, 26
 br i1 %7, label %10, label %8

8:
 %9 = icmp eq i8 %0, 32
 %. = select i1 %9, i32 26, i32 1
 br label %10

10:
 %.0.reg2mem.0 = phi i32 [%5, %4],
  [%., %8], [%.reload16.off, %6]
 ret i32 %.0.reg2mem.0
}
```

Autotuner → 14 instructions

`-reg2mem -instcombine -Os -O1`

Result after testing 26k different pass orderings

```
define i32 @f1(i8 %0) {
 %2 = zext i8 %0 to i32
 %.off = add i8 %0, 191
 %3 = icmp ult i8 %.off, 26
 br i1 %3, label %6, label %._crit_edge

._crit_edge:
 %.off24 = add i8 %0, 159
 %4 = icmp ult i8 %.off24, 26
 br i1 %4, label %6, label %._crit_edge9

._crit_edge9:
 %5 = icmp eq i8 %0, 32
 %spec.select = select i1 %5,
  i32 26, i32 1
 ret i32 %spec.select

6:
 %.sink = phi i32 [191, %1],
  [159, %._crit_edge]
 %7 = add nsw i32 %.sink, %2
 ret i32 %7
}
```

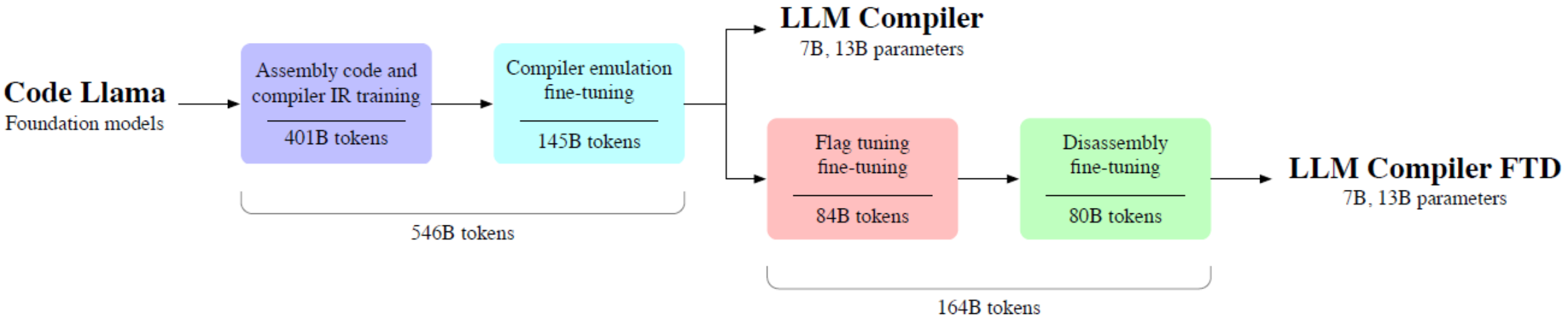
LLM →13 instructions

`-reg2mem -simplifycfg -mem2reg -jump-threading –Os`

This pass list appears 5 times in the training set

LLMCompiler: The Holy Grail (?) of LLM compiler research

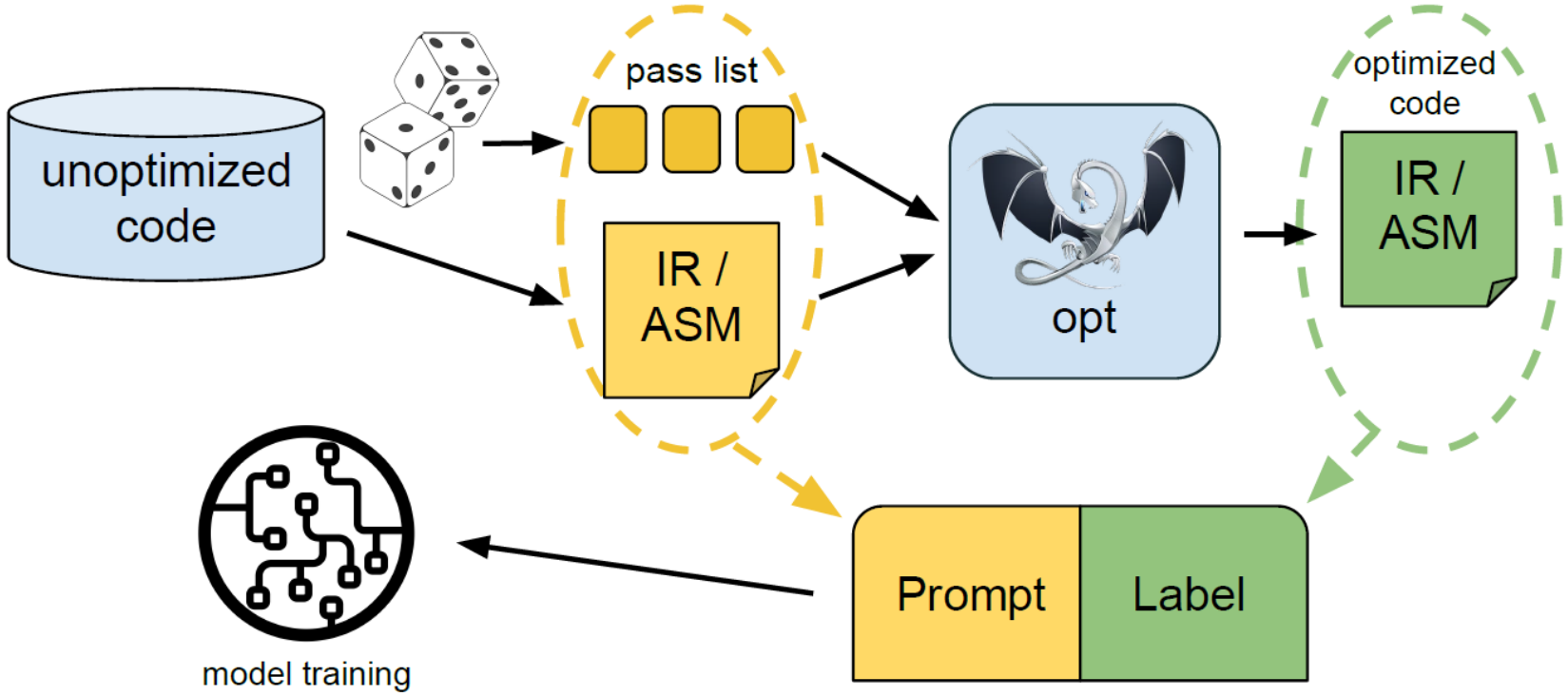
- LLMCompiler is a **family of foundation models** that have already been trained to understand the semantics of compiler IRs and assemblies and to emulate the compiler
- Fine-tuning with minimal data for specific downstream compiler optimization tasks
- Trained on IR and assembly code generated by a full-fledged compiler (LLVM v17.0.6)



Cummins, Chris, et al. "Meta Large Language Model Compiler: Foundation Models of Compiler Optimization." *arXiv preprint arXiv:2407.02524* (June 2024).

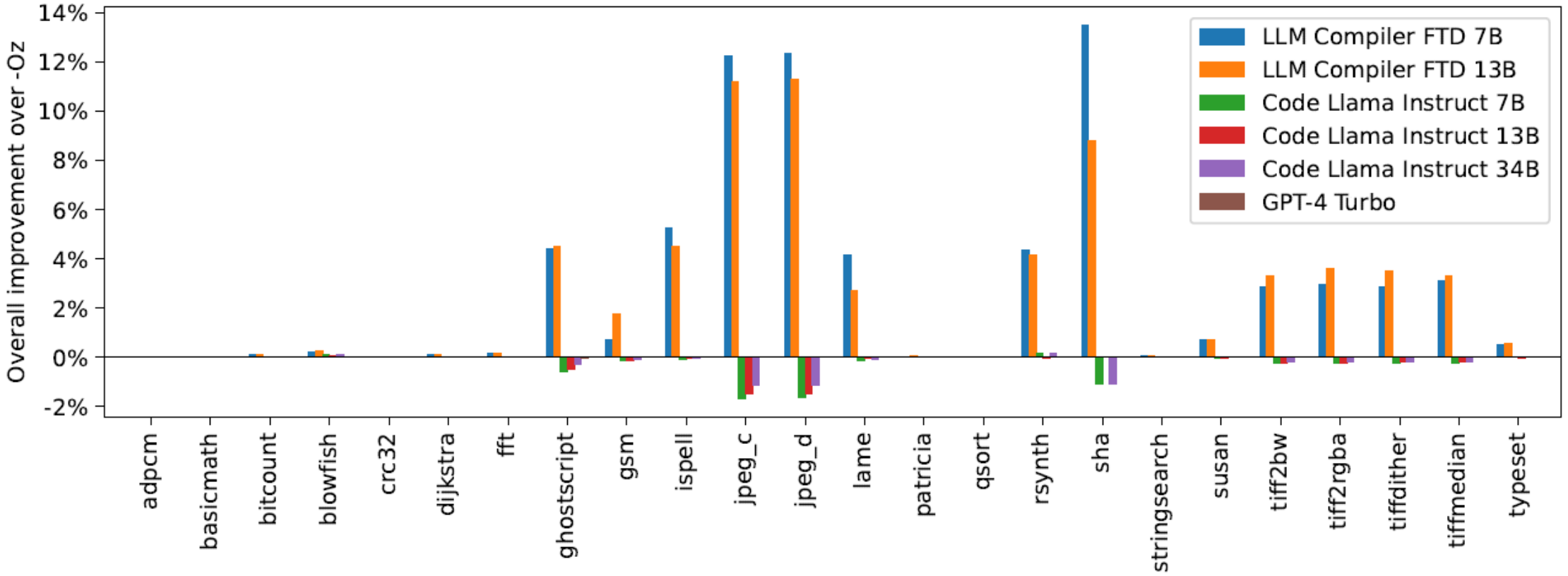
Compiler emulation fine-tuning

- **Compiler emulation dataset** → applying random lists of between 1 and 50 optimization passes to unoptimized programs



Cummins, Chris, et al. "Meta Large Language Model Compiler: Foundation Models of Compiler Optimization." *arXiv preprint arXiv:2407.02524* (June 2024).

Reducing the binary size (baseline –Oz)



Cummins, Chris, et al. "Meta Large Language Model Compiler: Foundation Models of Compiler Optimization." *arXiv preprint arXiv:2407.02524* (June 2024).

Outline

- Introduction
- LLMs for code generation
- LLMs for compiler assistance
- **Conclusion**

Conclusion

- Most of these papers are published on arXiv → Even if the quality is good and the code is usually provided, a peer-review process is missing
- What are the real advantages of using LLMs for the compilation process?
- Can we really trust these technologies?