



AI for Compiler: MLGO和LLM Compiler的介绍及使用

华为 编译器实验室

谢志恒

背景介绍



openEuler 24.03 LTS

openEuler 24.03 LTS 是基于6.6内核的长周期版本，面向服务器、云、边缘计算、AI和嵌入式场景，提供更多新特性和功能，给开发者和用户带来全新的体验，服务更多的领域和更多的用户。

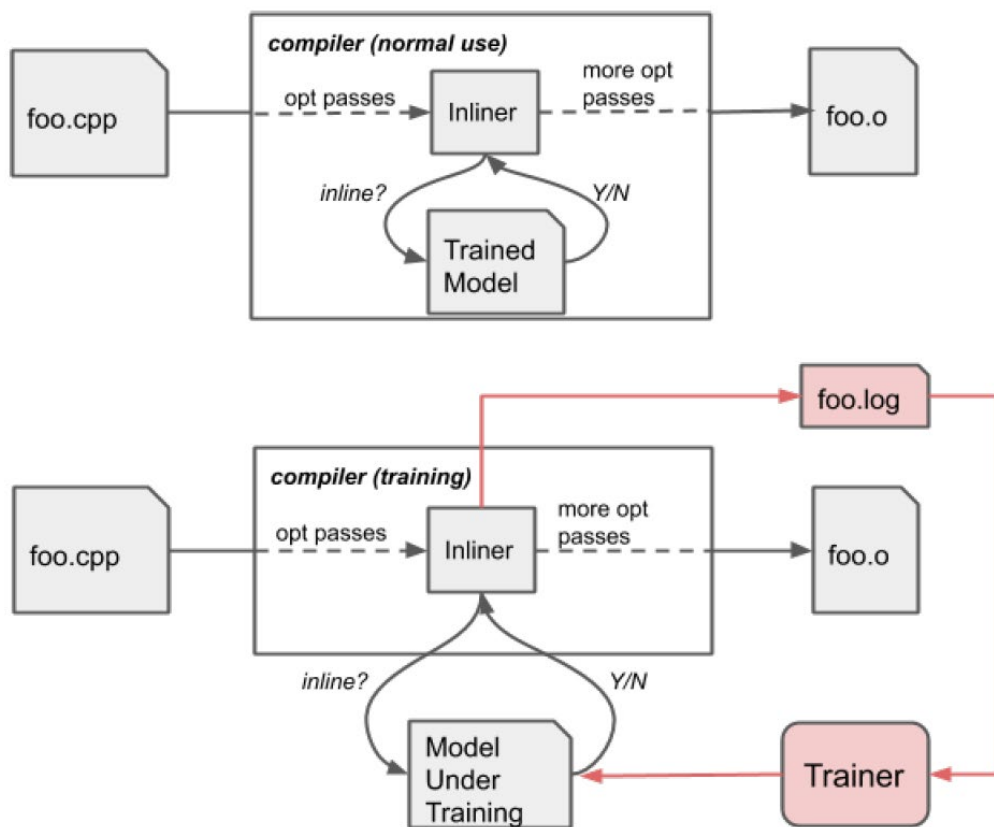
Planned EOL: 2026/05

[发行说明](#) | [安装指南](#) | [白皮书](#) | [生命周期](#)

架构	x86_64	<div>AArch64</div>	ARM32	LoongArch64	RISC-V
场景	服务器	<div>嵌入式</div>	边缘计算	云计算	
qemu-aarch64-6.6- <div>llvm</div>	-	AGDSN (10000Mb/s)	▼	前往下载 →	
qemu-aarch64- <div>llvm</div>	-	AGDSN (10000Mb/s)	▼	前往下载 →	
raspberrypi4-64	-	AGDSN (10000Mb/s)	▼	前往下载 →	
raspberrypi4-64-6.6	-	AGDSN (10000Mb/s)	▼	前往下载 →	
raspberrypi4-64-6.6- <div>llvm</div>	-	AGDSN (10000Mb/s)	▼	前往下载 →	
raspberrypi4-64-6.6-rt-hmi	-	AGDSN (10000Mb/s)	▼	前往下载 →	
raspberrypi4-64- <div>llvm</div>	-	AGDSN (10000Mb/s)	▼	前往下载 →	

- openEuler 24.03 LTS 我们首次正式发布了四个基于LLVM构建的嵌入式版本
- 我们将持续进行特性的引入、性能以及codesize的优化等
- 本次我们将结合AI for Compiler技术，介绍谷歌的MLGO和Meta的LLM Compiler技术及其使用，以及在codesize上的优化效果

什么是MLGO



MLGO Inline Optimization总览

- **MLGO**: a Machine Learning Guided compiler Optimizations framework
- 谷歌首次将ML技术集成到工业编译器LLVM当中 [arXiv:2101.04808](https://arxiv.org/abs/2101.04808)，并同时发布了模型训练框架 [ml-compiler-opt](https://github.com/google/ml-compiler-opt)
- 作为第一步，使用ML技术代替编译器启发式算法进行是否发生内联的决策，实现 codesize 的优化

如何使用MLGO inline优化

- LLVM 13及以上版本支持MLGO框架, [openeuler/llvm-project](https://openeuler.org/en-US/projects/llvm-project/) 主要版本为LLVM 17, 已天然支持
- 首先需要构建一个使能MLGO的LLVM编译器
构建环境: 需要安装tensorflow, pip3 install tensorflow
LLVM cmake构建配置: 需要添加,
 - DTENSORFLOW_AOT_PATH=/path/to/tensorflow
 - DLLVM_INLINER_MODEL_PATH=/path/to/model如果选择直接下载模型, 则需要提供下载链接,
 - DLLVM_INLINER_MODEL_PATH=download
 - DLLVM_INLINER_MODEL_CURRENT_URL=\$URL # 如 [google model url](#)
- 编译器运行环境不需要依赖tensorflow, 使用编译选项
-mllvm -enable-ml-inliner=release 使能优化

如何训练MLGO inline模型

➤ 谷歌的 [inlining-demo](#) 提供了使用MLGO框架训练模型的案例

1. 环境依赖：使用pipenv的 [ml-compiler-opt/Pipfile](#) 配置文件创建虚拟环境，并安装 mlgo-utils
2. 依赖包安装：使用脚本 [ml-compiler-opt/buildbot/build_tflite.sh](#) 安装依赖包，将生成 tflite.cmake 配置文件
3. 构建ML开发模式的LLVM编译器用于模型训练：需要添加cmake构建配置，-C /path/to/tflite.cmake
4. 提取语料库
 - a) 在构建目标应用时添加 **-Xclang=-fembed-bitcode=all** 选项，在obj文件中将生成.llvmbc和.llvmcmd段记录LLVM IR以及编译命令
 - b) 使用mlgo-utils工具指定目录提取语料库，它能够遍历目录下所有的obj文件，提取其中.llvmbc和.llvmcmd段的内容

[Nr]	Name	Type
	Size	EntSize
[0]	0000000000000000	NULL
[1]	.strtab 0000000000000069	STRTAB 0000000000000000
[2]	.text 0000000000000004	PROGBITS 0000000000000000
[3]	.llvmbc 0000000000000938	PROGBITS 0000000000000000
[4]	.llvmcmd 000000000000040e	PROGBITS 0000000000000000
[5]	.comment 000000000000006e	PROGBITS 0000000000000001

obj文件中.llvmbc和.llvmcmd段

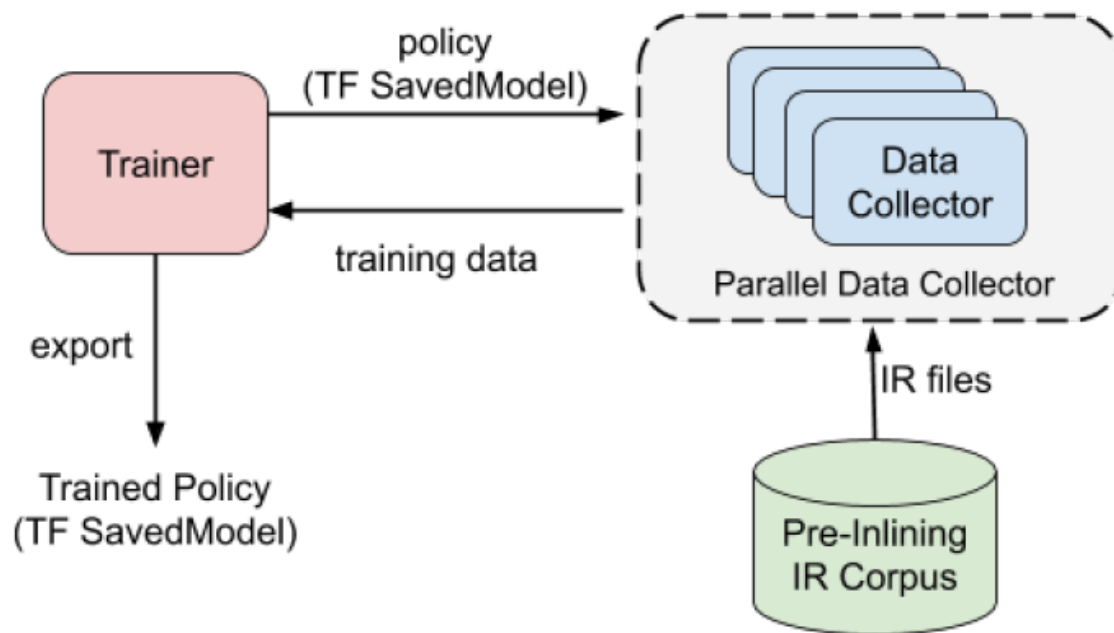
如何训练MLGO inline模型

➤ 谷歌的 [inlining-demo](#) 提供了使用MLGO框架训练模型的案例

4. 训练模型

- a) 基于LLVM inline的启发式算法训练一个基础的行为克隆模型，避免从零开始训练，可以加速模型的训练
- b) MLGO框架目前可以使用 Proximal Policy Optimization (PPO) 算法进行模型的训练

5. MLGO框架训练模型完成后会导出tensorflow格式的模型结果，参考上一章使用新的模型构建LLVM编译器



模型训练流程总览

SPEC CPU codesize优化效果

- 使用谷歌发布的 [inlining-for-size 模型](#) 验证SPEC CPU 2006和SPEC CPU 2017的codesize优化效果
- 相比C语言子项，C++语言子项的codesize优化效果更好一些

架构: x86				
编译器: openeuler/llvm-project				
ML模型版本: google/ml-compiler-opt/inlining-Oz-v1.1				
SPEC2006子项	子项类型	-Oz base	MLGO	codesize下降比例
401.bzip2	C	73880	73456	0.58%
403.gcc	C	3421080	3437024	-0.46%
429.mcf	C	23024	23024	0.00%
433.milc	C	135904	135904	0.00%
445.gobmk	C	4416792	4423696	-0.16%
456.hmmer	C	287480	291464	-1.37%
458.sjeng	C	144496	148304	-2.57%
462.libquantum	C	55608	55568	0.07%
464.h264ref	C	545968	549936	-0.72%
470.lbm	C	26456	26384	0.27%
482.sphinx3	C	193960	193776	0.09%
平均值				-0.39%
444.namd	C++	273232	272464	0.28%
447.dealII	C++	3643872	3449136	5.65%
450.soplex	C++	397232	397008	0.06%
453.povray	C++	1017864	1037352	-1.88%
471.omnetpp	C++	851144	851184	0.00%
473.astar	C++	53640	48432	10.75%
483.xalancbmk	C++	6114672	5807520	5.29%
平均值				2.88%

SPEC CPU 2006 codesize优化

架构: x86				
编译器: openeuler/llvm-project				
ML模型版本: google/ml-compiler-opt/inlining-Oz-v1.1				
SPEC2017子项	子项类型	-Oz	MLGO	codesize下降比例
500.perlbench_r	C	2117160	2124848	-0.36%
502.gcc_r	C	9241272	9257280	-0.17%
505.mcf_r	C	36096	36064	0.09%
511.povray_r	C++,C	1041320	1056920	-1.48%
519.lbm_r	C	26512	30536	-13.18%
525.x264_r	C	475904	473840	0.44%
538.imagick_r	C	1831688	1832136	-0.02%
544.nab_r	C	184712	188272	-1.89%
557.xz_r	C	201776	204272	-1.22%
平均值				-1.98%
508.namd_r	C++	631504	630072	0.23%
510.parest_r	C++	1.1E+07	9840872	9.06%
520.omnetpp_r	C++	2646680	2575384	2.77%
523.xalancbmk_r	C++	6906064	6559648	5.28%
531.deepsjeng_r	C++	94080	93968	0.12%
541.leela_r	C++	202440	177384	14.13%
平均值				5.26%

SPEC CPU 2017 codesize优化

openEuler Embedded LLVM版本codesize优化



- 使用谷歌发布的 inlining-for-size 模型和用openEuler Embedded版本提取的语料库进行训练的模型进行codesize优化

编译器：openeuler/llvm-project		
openEuler嵌入式版本codesize	openeuler-image-qemu-aarch64.rootfs.cpio.gz	codesize下降比例
default base	65447866	
MLGO google/ml-compiler- opt/inlining-Oz-v1.1	63134800	3.66%
MLGO 自训练模型	62850271	4.13%

使用两种模型构建openEuler Embedded版本效果

- 相关的自训练模型在完善后会计划发布到 [openEuler AI4C](#) 仓

MLGOPerf — inlining-for-performance

- 同时，inline优化也可以调整用于性能优化，华为加拿大编译器实验室首次实现了MLGOPerf框架训练 inlining-for-performance 模型 [arXiv:2207.08389](https://arxiv.org/abs/2207.08389)
- 使用MLGOPerf在SPEC CPU 2006和Cbench上验证获得1.8%和2.2%的性能收益
- 使用ML技术优化性能相比codesize会有更多的困难
 - a) 性能相比codesize更难测量，并且存在波动
 - b) 不同机器架构、硬件配置、环境参数等对于性能的影响很大
 - c) 更难构造理想的特性维度对软件进行刻画

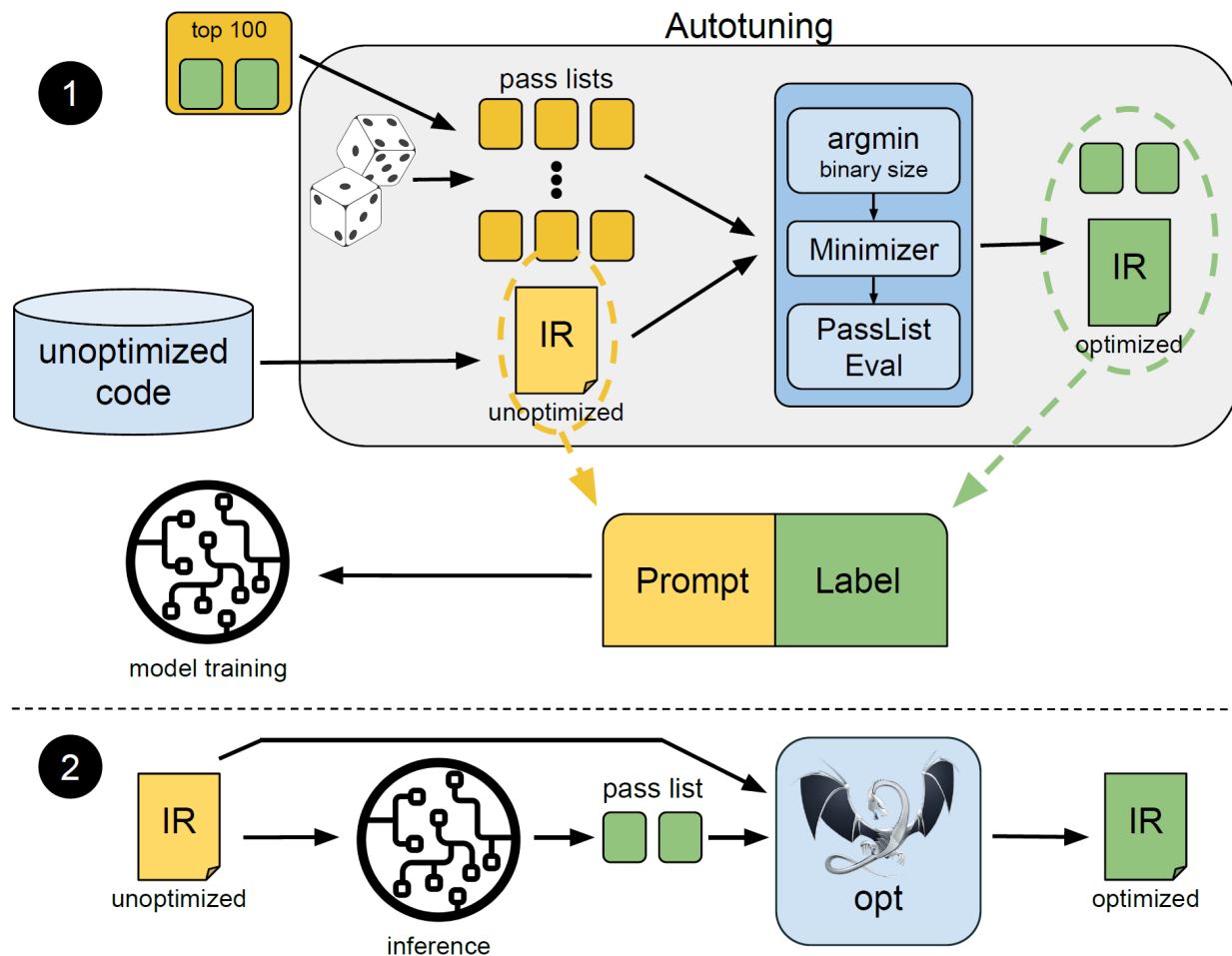
Benchmark	Speedup wrt O3
401.bzip2	1.052
403.gcc	1.022
429.mcf	1.009
445.gobmk	1.030
456.hmmmer	0.997
458.sjeng	1.003
462.libquantum	1.040
464.h264ref	1.068
471.omnetpp	1.004
433.milc	1.021
444.namd	0.992
453.povray	0.997
470.lbm	1.020
482.sphinx3	0.993
Geomean	1.018

SPEC CPU 2006性能优化

Benchmark	Speedup wrt O3
automotive_bitcount	1.002
automotive_qsort1	1.000
automotive_susan_c	1.020
automotive_susan_e	1.015
automotive_susan_s	1.002
bzip2d	1.025
bzip2e	1.014
consumer_jpeg_c	1.002
consumer_jpeg_d	1.041
consumer_lame	1.006
consumer_tiff2bw	1.027
consumer_tiff2rgba	1.051
consumer_tiffdither	1.006
consumer_tiffmedian	1.012
network_dijkstra	1.005
network_patricia	1.000
office_stringsearch1	0.996
security_blowfish_d	1.005
security_blowfish_e	1.059
security_rijndael_d	1.042
security_rijndael_e	1.037
security_sha	1.134
telecom_adpcm_c	1.001
telecom_adpcm_d	1.001
telecom_CRC32	1.065
Geomean	1.022

Cbench性能优化

Meta Large Language Model Compiler



LLM Compiler 训练和推理流程

- LLMs在代码生成、代码翻译、代码测试等任务上，业界已经有很多不错的成果（StarCoder、Code Llama、GPT-4）
- Meta首次发布了大模型编译器LLM Compiler [arXiv:2407.02524](https://arxiv.org/abs/2407.02524)，它在Meta Code Llama的基础上训练增加了代码优化和代码编译的能力，同时也能够进行 codesize 的优化
- 训练过程中Prompt为未优化的LLVM IR代码，Label为优化遍列表、二进制体积以及优化后的代码

LLM Compiler代码编译能力



- 输入LLVM IR和进行编译的指导语句，LLM Compiler能够模拟编译器优化，输出优化后的LLVM IR/汇编，但是目前正确率无法达到100%

Prompt:

[INST] Give the assembly for the following code when optimized using opt -p 'module(default<0z>)':

```
<code>; ModuleID = '<stdin>'
source_filename = "-"
target_datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8:16:32:64-S128"
target_triple = "x86_64-unknown-linux-gnu"

; Function Attrs: minsize nounwind optsize uwtable
define dso_local i32 @add_two(i32 noundef %0, i32 noundef %1) #0 {
    %3 = alloca i32, align 4
    %4 = alloca i32, align 4
    store i32 %0, ptr %3, align 4, !tbaa !5
    store i32 %1, ptr %4, align 4, !tbaa !5
    %5 = load i32, ptr %3, align 4, !tbaa !5
    %6 = load i32, ptr %4, align 4, !tbaa !5
    %7 = add nsw i32 %5, %6
    ret i32 %7
}

attributes #0 = { minsize nounwind optsize uwtable "min-legal-vector-width"="0" "no-trapping-math"="true"
  ↳ "stack-protector-buffer-size"="8" "target-cpu"="x86-64" "target-features"="+cmov,+cx8,+fxsr,+mmx,+sse
  ↳ ,+sse2,+x87" "tune-cpu"="generic" }

!llvm.module.flags = !{!0, !1, !2, !3}
!llvm.ident = !{!4}

!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{i32 8, !"PIC Level", i32 2}
!2 = !{i32 7, !"PIE Level", i32 2}
!3 = !{i32 7, !"uwtable", i32 2}
!4 = !{"clang version 17.0.6 (git@github.com:fairinternal/CodeGen.git
  ↳ b05db9bbf7a92019267416c1bb9996fe6134e3f1)"}
!5 = !{!6, !6, i64 0}
!6 = !{"int", !7, i64 0}
!7 = !{"omnipotent char", !8, i64 0}
!8 = !{"Simple C/C++ TBAA"}
</code>
```

The input code has instruction count 8 and binary size 65 bytes. [/INST]

Answer:

The assembly will have instruction count 2 and binary size 53 bytes:

```
<code> .text
.file "-"
.globl add_two                                # -- Begin function add_two
.type add_two,@function
add_two:                                      # @add_two
.cfi_startproc
# %bb.0:
movl %esi, %eax
addl %edi, %eax
retq
.Lfunc_end0:
.size add_two, .Lfunc_end0-add_two
.cfi_endproc

                                # -- End function
.ident "clang version 17.0.6 (git@github.com:fairinternal/CodeGen.git
  ↳ b05db9bbf7a92019267416c1bb9996fe6134e3f1)"
.section ".note.GNU-stack","",@progbits
.addrsig
</code>
```

LLM Compiler codesize优化能力



- 输入LLVM IR和进行codesize优化的指导语句，LLM Compiler能够输出优化后的结果，并提供需要opt运行的优化遍列表

Prompt:

[INST] Tell me how to optimize this LLVM-IR for object file size:

```
<code>; ModuleID = '<stdin>'
source_filename = "-"
target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-unknown-linux-gnu"
```

```
; Function Attrs: minsize nounwind optsize uwtable
define dso_local i32 @add_two(i32 noundef %0, i32 noundef %1) #0 {
    %3 = alloca i32, align 4
    %4 = alloca i32, align 4
    store i32 %0, ptr %3, align 4, !tbaa !5
    store i32 %1, ptr %4, align 4, !tbaa !5
    %5 = load i32, ptr %3, align 4, !tbaa !5
    %6 = load i32, ptr %4, align 4, !tbaa !5
    %7 = add nsw i32 %5, %6
    ret i32 %7
}
```

```
attributes #0 = { minsize nounwind optsize uwtable "min-legal-vector-width"="0" "no-trapping-math"="true"
    ↪ "stack-protector-buffer-size"="8" "target-cpu"="x86-64" "target-features"="+cmov,+cx8,+fxsr,+mmx,+sse
    ↪ ,+sse2,+x87" "tune-cpu"="generic" }
```

```
!llvm.module.flags = !{!0, !1, !2, !3}
!llvm.ident = !{!4}
```

```
!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{i32 8, !"PIC Level", i32 2}
!2 = !{i32 7, !"PIE Level", i32 2}
!3 = !{i32 7, !"uwtable", i32 2}
!4 = !{!"clang version 17.0.6 (git@github.com:fairinternal/CodeGen.git
    ↪ b05db9bbf7a92019267416c1bb9996fe6134e3f1)"}
!5 = !{!6, !6, i64 0}
!6 = !{!"int", i7, i64 0}
!7 = !{!"omnipotent char", !8, i64 0}
!8 = !{!"Simple C/C++ TBAA"}</code> [/INST]
```

Answer:

The code has object file size of 65 which can be reduced to 53 by running:

```
'opt -p 'module(default<0z>)' ' '
```

This will produce code:

```
<code>; ModuleID = '<stdin>'
source_filename = "-"
target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-unknown-linux-gnu"
```

```
; Function Attrs: minsize mustprogress nofree norecurse nosync nounwind optsize willreturn memory(none)
    ↪ uwtable
define dso_local i32 @add_two(i32 noundef %0, i32 noundef %1) local_unnamed_addr #0 {
    %3 = add nsw i32 %1, %0
    ret i32 %3
}
```

```
attributes #0 = { minsize mustprogress nofree norecurse nosync nounwind optsize willreturn memory(none)
    ↪ uwtable "min-legal-vector-width"="0" "no-trapping-math"="true" "stack-protector-buffer-size"="8" "
    ↪ target-cpu"="x86-64" "target-features"="+cmov,+cx8,+fxsr,+mmx,+sse,+sse2,+x87" "tune-cpu"="generic" }
```

```
!llvm.module.flags = !{!0, !1, !2, !3}
!llvm.ident = !{!4}
```

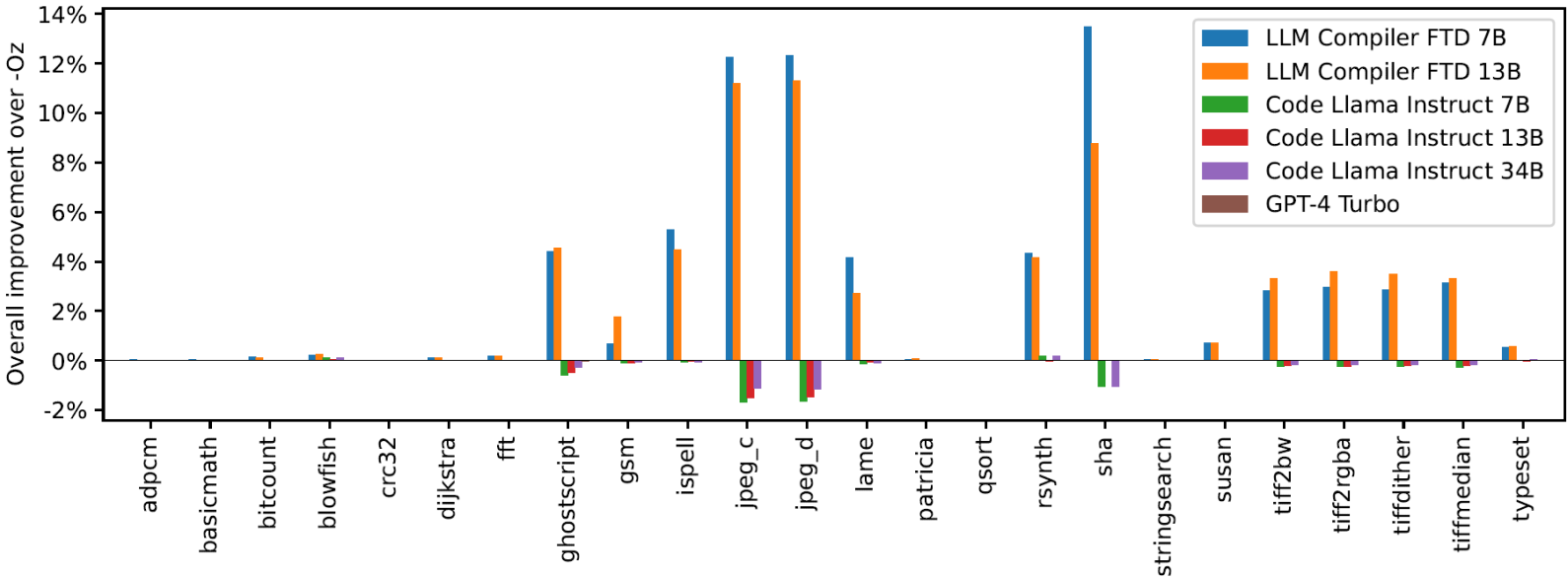
```
!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{i32 8, !"PIC Level", i32 2}
!2 = !{i32 7, !"PIE Level", i32 2}
!3 = !{i32 7, !"uwtable", i32 2}
!4 = !{!"clang version 17.0.6 (git@github.com:fairinternal/CodeGen.git
    ↪ b05db9bbf7a92019267416c1bb9996fe6134e3f1)"}</code>
```

LLM Compiler codesize优化效果

- 使用LLM Compiler优化MiBench codesize的效果如右图所示
- 在-Oz的基础上，LLM Compiler可以整体再额外获得4.88%的codesize下降

	Size	Improved	Regressed	Overall improvement over -Oz	
				zero-shot	-Oz backup
LLM COMPILER FTD	7B	1,465	302	4.77%	5.24%
	13B	1,466	299	4.88%	5.26%
CODE LLAMA - INSTRUCT	7B	379	892	-0.49%	0.23%
	13B	319	764	-0.42%	0.18%
	34B	230	493	-0.27%	0.15%
GPT-4 Turbo (2024-04-09)	-	13	24	-0.01%	0.03%

MiBench整体codesize优化效果



MiBench各子项codesize优化效果

LLM Compiler现在存在的问题

- 输入长度的限制：LLM Compiler目前支持的最大窗口长度为16K token，而实际应用中将会远超这个限制
- 正确性问题：LLM Compiler目前无法确保输出的结果完全正确
- 编译耗时长：LLM Compiler编译30行的LLVM IR（相当于C源码约10行左右），在x86机器并行度64线程下需要消耗40G+内存，编译180s，对比使用LLVM编译器耗时0.02s
- 编译接收的语言主要是LLVM IR：LLM Compiler训练使用的语料库主要由LLVM IR组成，目前验证对于C/C++源码文件支持程度较低
- 适配应用构建工程的问题：LLM Compiler和传统GCC/LLVM编译器的使用方式不同，在实际构建工程中使用仍需再封装一层接口



Thank You.

Compiler SIG 专注于编译器领域技术交流探讨和分享，包括 GCC/LLVM/OpenJDK 以及其他的程序优化技术，聚集编译技术领域的学者、专家、学术等同行，共同推进编译相关技术的发展。



毕昇编译公众号



Compiler 交流群小助手