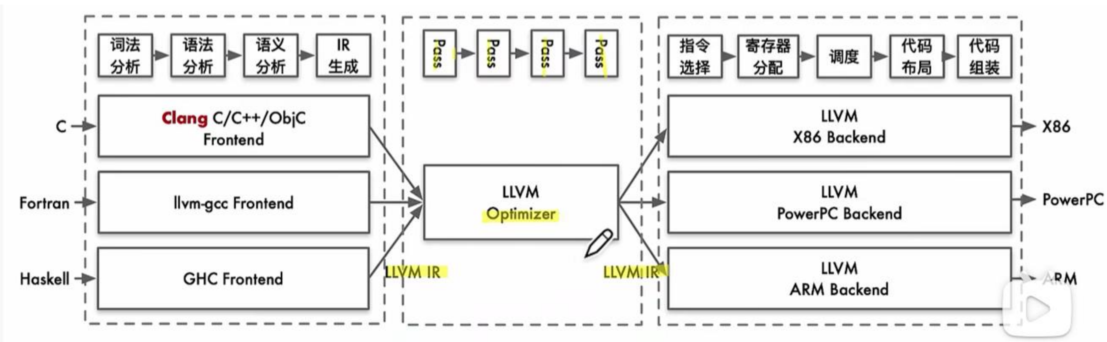
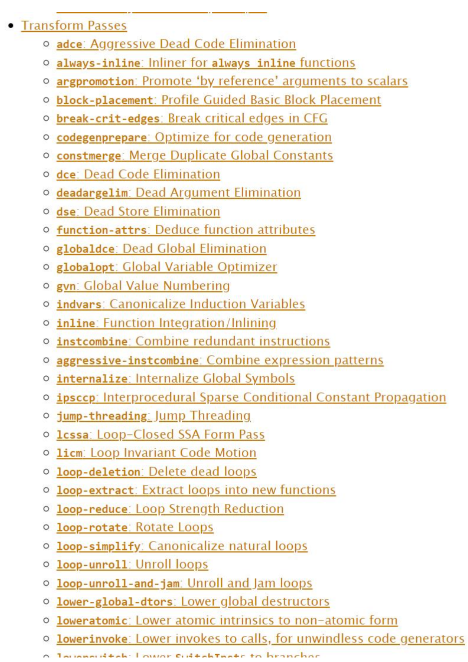
Large Language Models for Compiler Optimization

一. 摘要

本文探讨了将大语言模型在代码优化中的应用。我们提出了一个从头开始训练的 7B 参数的模型，以优化 LLVM 汇编的代码大小。该模型将未经优化的汇编作为输入并输出编译器选项列表，以便对程序进行最佳优化。在训练过程中，我们要求模型预测优化前后的指令数，以及优化后的代码本身。这些辅助学习任务显著提高模型优化性能，并提高模型的理解深度。

我们对大量测试程序进行了评估。我们的方法在减少指令数方面比编译器提高了 3.0%，超过了两个需要数千次编译的 baseline。该模型在 91% 生成了可编译代码，并在 70% 的情况下完美模拟了编译器的输出。





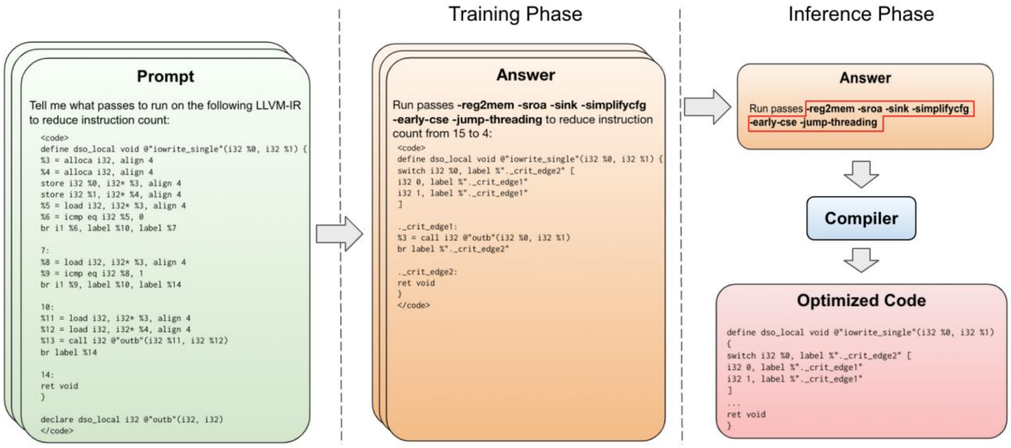
- Transform Passes
  - **adce**: Aggressive Dead Code Elimination
  - **always-inline**: Inliner for **always_inline** functions
  - **argpromotion**: Promote 'by reference' arguments to scalars
  - **block-placement**: Profile Guided Basic Block Placement
  - **break-crit-edges**: Break critical edges in CFG
  - **codegenprepare**: Optimize for code generation
  - **constmerge**: Merge Duplicate Global Constants
  - **dce**: Dead Code Elimination
  - **deadargelim**: Dead Argument Elimination
  - **dse**: Dead Store Elimination
  - **function-attrs**: Deduce function attributes
  - **globaldce**: Dead Global Elimination
  - **globalopt**: Global Variable Optimizer
  - **gvn**: Global Value Numbering
  - **indvars**: Canonicalize Induction Variables
  - **inline**: Function Integration/Inlining
  - **instcombine**: Combine redundant instructions
  - **aggressive-instcombine**: Combine expression patterns
  - **internalize**: Internalize Global Symbols
  - **ipsccp**: Interprocedural Sparse Conditional Constant Propagation
  - **jump-threading**: Jump Threading
  - **lcssa**: Loop-Closed SSA Form Pass
  - **licm**: Loop Invariant Code Motion
  - **loop-deletion**: Delete dead loops
  - **loop-extract**: Extract loops into new functions
  - **loop-reduce**: Loop Strength Reduction
  - **loop-rotate**: Rotate Loops
  - **loop-simplify**: Canonicalize natural loops
  - **loop-unroll**: Unroll loops
  - **loop-unroll-and-jam**: Unroll and Jam loops
  - **lower-global-dtors**: Lower global destructors
  - **loweratomic**: Lower atomic intrinsics to non-atomic form
  - **lowerinvoke**: Lower invokes to calls, for unwindless code generators

PASS ORDERING WITH LLMS

在这项工作中，我们的目标是编译器 pass order。pass order 任务就是从编译器中可用的 optimizing transformation passes 中，选择能为特定输入代码产生最佳结果的 pass list。操纵 pass order 已被证明对运行时性能和代码大小都有相当大的影响。

我们的目标是优化代码大小。在先前的工作中使用 IR 指令数作为二进制大小的代理。

预计在将来以运行时性能为目标。



输入 IR Normalization

我们还包括两项辅助任务

i） 生成应用优化前后的代码指令数

ii）生成输出 IR

部署不需要辅助任务

Model Architecture

我们使用与 Llama 2 相同的模型架构和字节对编码（BPE），从头开始训练模型。使用 Llama 2 中最小的配置： 32 attention heads，4096 hidden dimensions, and 32 layers， 7B parameters.

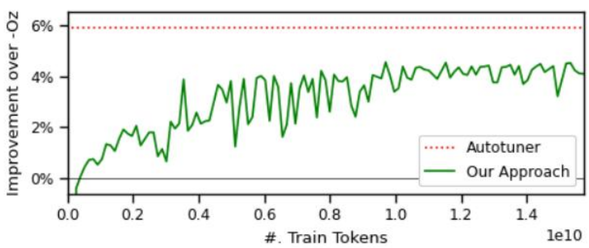
Training

	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

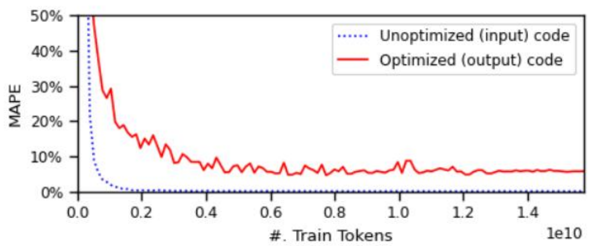
Autotuner：

对于每个函数，我们运行随机搜索固定的时间(780 秒)，然后通过迭代地删除单个随机选择的传递来最小化最佳传递列表，以查看它们是否对指令计数有贡献。如果没有，则将其丢弃。在对每个函数执行此操作后，我们聚合唯一最佳通过列表集，并将它们传播到所有其他函数。因此，如果发现一个通过列表在一个函数上运行良好，那么它将在所有其他函数上进行尝试。

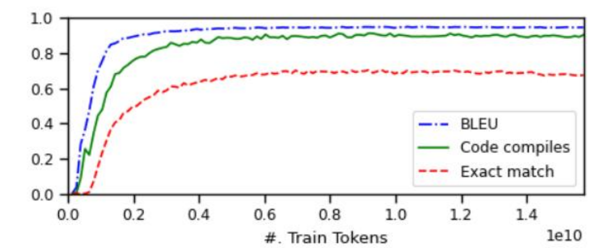
平均编译每个程序 37424 次，计算成本很大



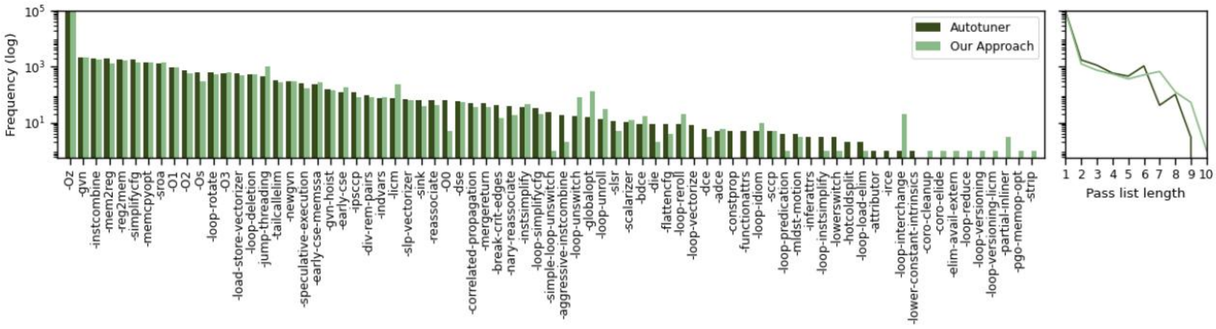
(a) Performance of generated pass lists.



(b) Accuracy at predicting instruction counts.



	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%




```
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
}
```

(a) Input code (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
}
```

(b) Autotuned code (14 instructions) using passes: -reg2mem -instcombine -Os -O1.

```
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
}
```

(c) Model-optimized code (13 instructions) and pass list: -reg2mem -simplifycfg -mem2reg -jump-threading -Os.

在 90.3% 的情况下，模型生成的优化 IR 可以编译。

在 68.4% 的情况下，输出的 IR 与编译器生成的一致。

在 9.7% 的情况下，生成的 IR 无法编译。

挑战一：文本精度指标（如 BLEU 分数）对代码中的差异（如变量名和交换操作数顺序）很敏感。

挑战二：评估语义等同性。