ML Guided Optimizations ('MLGO')

Mircea Trofin (pronounced "*Meercha*")
Compiler Optimization, Google

- you can use ML in clang/llvm (as of 2020)
- open source, in “trunk”
- support for: embedding ML models, extracting training corpus, etc.

At Google:

- Size (ML makes per-callsite inlining decisions):
  - Chrome on Android (since mid-2022)
  - Fuchsia (an OS, runs on Google Assistant devices, ~2020)
  - internal cloud infra (fixed size boot partition. 2022)
- Performance:
  - a register allocation policy for e.g. Search, Big Table, etc. (2022)
  - ML makes Live Range conflict resolution choices

This talk...

- Scope (C/C++ at Google)
- Why ML
- How (high-level)

- Biggest challenges (research opportunities)

Google + LLVM

Clang/LLVM

- github.com/llvm/llvm-project
- LLVM: a library
  - IR
  - extensible pass (“phase”) model
  - state of the art optimizations
  - lowering to Machine IR (MIR) (x86, arm, etc)
- clang - a frontend, lowers C/C++ to IR
- other frontends: - rust, swift…
- also: lld (linker), lldb (debugger)...

C/C++ at Google

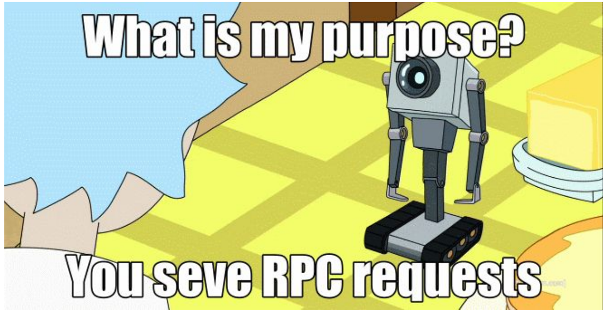
- compiler (performance) engineering matters quite directly to the “bottom line”
    - even small performance improvements (0.5% - 1%) matter a lot
        - less hardware
        - lower power utilization
        - better user experience
        - ...
    - continuous improvement
        - 1% today, another 1% tomorrow…
- we use C/C++ for all the large scale, performance-critical services
- at Google, “C/C++” == Clang / LLVM

When we talk about performance, we *assume*...

- performance profiles (FDO aka PGO)
  - frequency of function calls
  - CFG edge probability
  - loop average trip count

- ThinLTO
  - Link Time Optimization (LTO): reason about the whole program
  - …but because or binaries are statically linked and large -> ThinLTO

What do our binaries usually do



- serve RPC requests
- a request passes through lots of code
    - reusable libraries (both 1st and 3rd party)

- reusable code tries to be ***generic*** wrt context of use
- performance is about being as ***specific*** as possible to the usage context

…our binaries *continued…*

- data won't fit in the cache
- …nor would the hot/warm instruction set for any particular RPC

- interprocedural optimizations (IPO: e.g. inlining) *generally* have big impact (for us)
  - this produces large functions

Why ML

- IPO (inlining), regalloc… reasoning about large data (graphs)
  - local decisions have far-reaching effects
  - complex (i.e. hard to reason about by humans)
  - without ML: *heuristics*:
    - no known perfect optimization algo
    - so we use rules of thumb… tendency for local optima
    - empirically do OK, sometimes they don’t, that’s OK
- ***Reinforcement Learning*** (and similar techniques) **scale well with large data and problems like these**

Why *not* ML

- compilers (at least systems-oriented ones) must be:
  - deterministic (same compiler | same flags | same input => bit-identical output)
  - timely
  - correct
- these are antithetic to ML

What’s the hope / promise?

- break through complexity limitations
- focus on feature extraction **rather than** manually fine-tuning knobs
- automated periodic re-tuning **rather than** “run benchmark suite, see what breaks”

What’s the ***full*** *engineering* problem?

- keep the compiler deterministic, correct, and timely
- have a systematic approach to training
- scalable to compiler community

RL + LLVM

What's a *ML* policy?

- It's a neural network
- …which is a function
  - takes inputs ("tensors": buffers of scalars)
  - executes
  - produces a result
  - (up to user to interpret result)
- ***'executes'***:
  - can be compiled to native code ("AOT")
  - …or interpreted

Guidelines

- ask ML to decide among correctness-preserving alternatives
  - *if* a call site is legally inlinable… should it?
  - *if* some Live Ranges conflict, and a subset of them can be split… which should that be?
- train offline… rarely
- “use” treats ML as an implementation detail
  - so the compiler looks and feels “as usual”
  - …just that some decisions are made differently… but that’s implementation detail

Training: Very High Level

---

0 start with a *policy*

1 ***observe it in action***

2 compare with baseline => ***reward***

3 use reward (and observations) to **produce** ***another policy***

4 goto 1

github.com/google/ml-compiler-opt (non-prescriptive; what ***we*** use)

“Observe it in action”

- compiler:
  - extract **features**
  - execute policy
  - get **result**
  - act on it
- for training, we may additionally want to get a trace of feature values (“observations”) and decisions

LLVM support: MLModelRunner

```
#include "llvm/Analysis/MLModelRunner.h"

// switch to index-based parameter lookup                    index 0                            index 1    ...
MLModelRunner *Runner = factory_method({{"foo", int64_t, (1, 10)},  {"bar", float,...}})

// direct access to parameters' backing buffers to avoid imposing memcpy-ing
Runner->getTensor<int64_t>(foo_index/*==0*/)[0] = Callee.getBasicBlockList().size();
Runner->getTensor<float>(bar_index/*==1*/)[0] = Module.getFunctionList().size();

// execute the model and interpret the result
bool ShouldInline = Runner->evaluate<bool>();
```

examples:

- lib/Analysis/MLInlineAdvisor.cpp
- lib/CodeGen/MLRegAllocEvictAdvisor.cpp

“Reward”

- improvement / regression from a **baseline**
- for size, it’s easy: # of bytes (ls -l)

- what about performance?

Reward for performance

- benchmarks
  - easy for toy problems, hard to scale
  - representativeness (feature value distribution should match deployment)
  - isolated hardware
  - time consuming build and execution
- but we have profile information!
  - can we pre-compute a value at compile time?

This is the main challenge

The main challenges

- profile information is not precise enough
  - **absolutely** better than nothing
  - but lossy during certain optimizations (like inlining)
  - current direction: contextual profiles

- latency prediction
  - pipeline effects
  - cache effects
  - but: we don’t care about accurate *absolute* prediction. Just accurate *trend*

Getting involved (with LLVM, or MLGO)

---

https://discourse.llvm.org/ (tag: #mlgo)

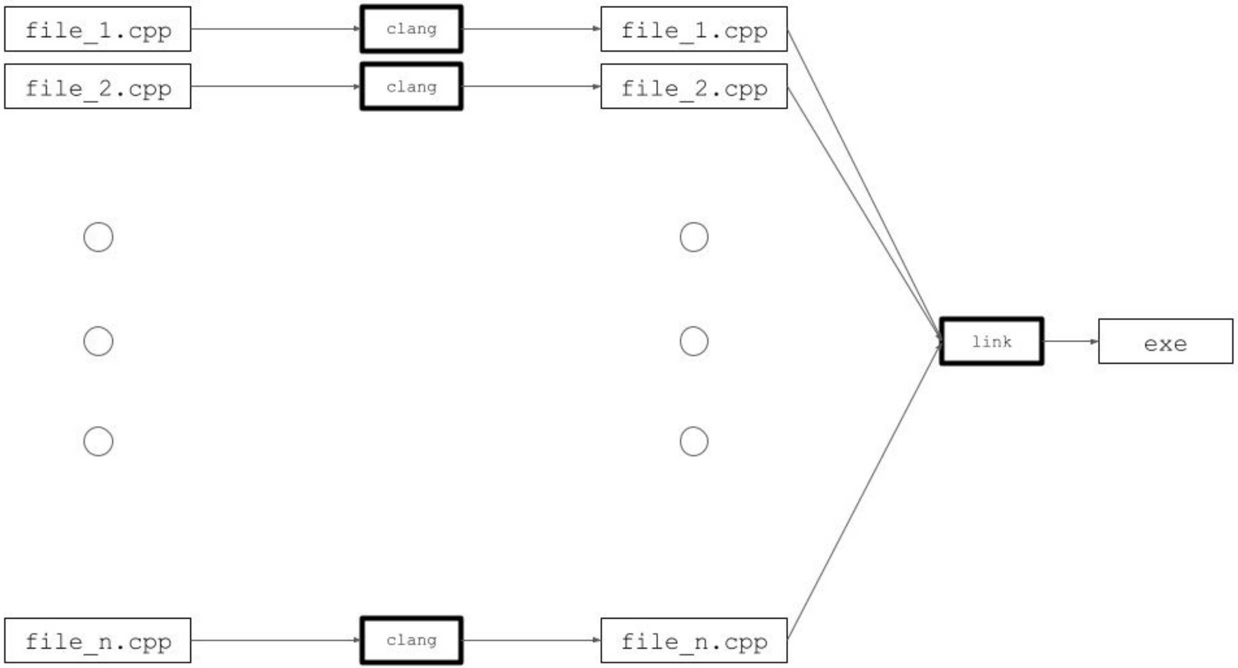
Google Summer of Code, LLVM project

Internships @ Google

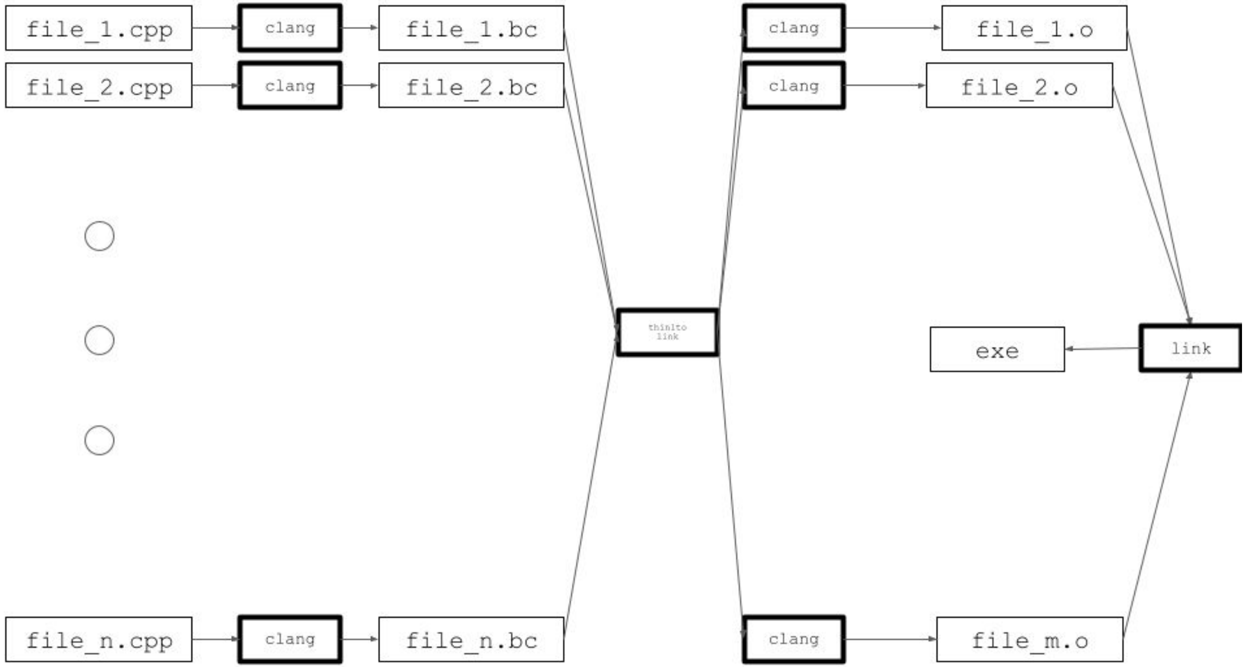
Student Researcher Program

Appendix

“vanilla” (non-ThinLTO)



ThinLTO



file_1.cpp
clang
file_1.bc
file_2.cpp
clang
file_2.bc
file_n.cpp
clang
file_n.bc
thinlto
link
clang
file_1.o
clang
file_2.o
clang
file_m.o
link
exe

How to talk to ML models

ML Models

- essentially, a function written in a DSL
  - Tensorflow: “Saved Model”
- The DSL needs an interpreter / compiler
  - abstraction: `llvm/Analysis/MLModelRunner.h` (`llvm::MLModelRunner`)
- Arguments & return: “Tensors”
  - `llvm/Analysis/TensorSpec.h` (`llvm::TensorSpec`)
    - **name**: name-based binding
    - **type**: scalar type (int32, float..)
    - **shape**: e.g. {3, 2, 5} (but we really care it’s 3*2*5*sizeof(int32) = 120 bytes).

MLModelRunner high-level

use ML **only** for performance, not correctness, decisions

```
#include "llvm/Analysis/MLModelRunner.h"

// switch to index-based parameter lookup                 index 0                          index 1    ...
MLModelRunner *Runner = factory_method({{"foo", int64_t, (1, 10)},  {"bar", float,...}})

// direct access to parameters' backing buffers to avoid imposing memcpy-ing
Runner->getTensor<int64_t>(foo_index/*==0*/)[0] = Callee.getBasicBlockList().size();
Runner->getTensor<float>(bar_index/*==1*/)[0] = Module.getFunctionList().size();

// execute the model and interpret the result
bool ShouldInline = Runner->evaluate<bool>();
```

examples:

- lib/Analysis/MLInlineAdvisor.cpp
- lib/CodeGen/MLRegAllocEvictAdvisor.cpp

Contract with implementers

```
{"name1", float, {1, 3}},  →  InputBuffers[0]
{"name2", uint32_t, {10}}, →  InputBuffers[1]
...
```

```
getTensor(size_t I) { return InputBuffers[I]; }
```

- tensor buffer lifetime == MLModelRunner's lifetime
- row-major order flattening

- "→" is the implementer's ctor responsibility
    - because it may have preferences / internal optimizations
- if implementer doesn't know a tensor, we'll allocate a buffer for it (for versioning / evolution)

llvm::ReleaseModeModelRunner - embed compiled model

- `llvm/Analysis/ReleaseModeModelRunner.h`
- see `llvm/cmake/modules/TensorFlowCompile.cmake:` **`tf_find_and_compile`**
- **must:**

```
$ pip install tensorflow
$ cmake <..> -DTENSORFLOW_AOT_PATH=<...>/site-packages/tensorflow
     (+ flags to specific models)
```

```
using CompiledModelType = RegAllocEvictModel; // <- generated

Runner = std::make_unique<ReleaseModeModelRunner<CompiledModelType>>(
            MF.getFunction().getContext(),            // just for Ctx.emit
            InputFeatures,                         // std::vector<TensorSpec>
            DecisionName);                         // just the tensor name of the output
```

- examples in `lib/{Analysis|CodeGen}/CMakeLists.txt`
- test model generators `lib/Analysis/models/gen-{regalloc-eviction|inline-oz}-test-model.py`
- tensorflow pip dependency: the way the AOT compiler & C++ wrapper sources are packaged *(so… install a python package just to get to C++ / native "stuff"? yup!)*

llvm::InteractiveModelRunner - ask an external agent

- available “off the shelf”
- implements a “dm_env”, or “gym”, interface
  - meant for training / research.
  - **NOT** intended for production

- “evaluate”:
  - write all features to a file desc
  - wait for external agent to give answer
- use standard LLVM IO file descriptors (`sys::fs` APIs) - *can be named pipes*

```
std::make_unique<InteractiveModelRunner>(
        M.getContext(), Features, OutputSpec,
        InteractiveChannelBaseName + ".out",
        InteractiveChannelBaseName + ".in")
```

- complete examples:
  `llvm/test/CodeGen/MLRegAlloc/interactive-mode.ll`
  `llvm/test/Transforms/Inline/ML/interactive-mode.ll`

yes, yes, yet another serialization format...

```
serializer:   llvm/Analysis/TrainingLogger.h
deserializer: lib/Analysis/models/log_reader.py
```

regalloc example:

1. `{“features”:[{“name”:”mask”,”type”:”int64_t”....],]..}`
2. `{“context”:”aFunctionName”}`
3. `{“observation”:0}`
4. ***<binary data dump of tensor values>***`\n`
5. `{“observation”:1}`

`...`

llvm::ModelUnderTrainingRunner - load and interpret

- works with build systems, but slower than AOT - it's an interpreter!
- initially used for training, also valuable for the added flexibility
- ***must embed the TFLite runtime***:

```
$ mkdir /tmp/tflite
$ cd /tmp/tflite
$ curl -s https://raw.githubusercontent.com/google/ml-compiler-opt/main/buildbot/build_tflite.sh | bash

$ cd $LLVM && mkdir build && cd build
$ cmake <...> -C /tmp/tflite/tflite.cmake
```

```
std::unique<MLModelRunner> Runner =
    ModelUnderTrainingRunner::createAndEnsureValid(
        Ctx,
        ModelPath,      // <- you can pass a model from command line
        DecisionName,
        InputSpecs)
```

- `ModelPath` points to a dir containing a `model.tflite` file and an `output_spec.json`
  - canonical saved model -> tflite converter: `lib/Analysis/models/saved-model-to-tflite.py`
  - canonical json: `lib/Analysis/models/gen-{inline-oz|regalloc-eviction}-test-model.py`

Corpus Collection

Corpus collection

- independently (re)compile individual modules, in production configuration
- leverage `.llvmbc` and `.llvmcmd` (existing feature)

Steps:

1) build your project with your build system...

    ...but pass additional flags

2) find[?] the native .o files and scrape the 2 sections

```
llvm-objcopy -dump-section= .llvmbc=<output.bc> native.o /dev/null
```

[?] *compile_commands.json | linker .params |...* *see* https://github.com/google/ml-compiler-opt/blob/main/compiler_opt/tools/extract_ir.py

Details

- Frontend (pre-(Thin)LTO) clang:

  ```
  clang <...> -Xclang=-fembed-bitcode=all
  ```

- ThinLTO “distributed”:

  ```
  clang <...> -mllvm
  -thinlto-embed-bitcode=post-merge-pre-opt
  ```

- ThinLTO “local”:

  ```
  ld.lld <...> -WL,--save-temps=import \
      -Wl,--thinlto-emit-index-files
  ```

  - this dumps files named *xyz*.**3**.import.bc and *xyz*.thinlto.bc in our output dir
  - **not using .llvmbc / .llvmcmd**

A corpus is...

- a directory of files
- a corpus element is:
  - a .bc (IR)
  - a .cmd file
  - (thinlto) a .thinlto.bc index file (still needed for WholeProgramDevirt)
- to re-run compilation:
  - run clang with the .cmd options (note: they are ‘\0’ separated…)
  - adjust input/output paths (and thinlto index)
  - pass `-mllvm -thinlto-assume-merged` if ThinLTO
- a corpus element is **compilable independently from the build system**