Common Facilities for ML-Guided Optimizations in LLVM

Mircea Trofin (Google)

(pronounced “Meercha”)

Agenda | Scope

- how to talk to ML models
  - different scenarios: production, evaluation, training, research...
- infra *supporting* training:
  - extracting training data
  - IR corpus collection
  - (ongoing) scoring
    - “is this new policy better / worse (and by how much)”
  - *build bots (ml-opt-*. not covered here)*
- next (short-term) mlgo infra

Not in scope: *how* to train

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**

Scoring/Rewards

Policy Scoring: the most important problem (for policy training)

- [WIP] llvm-cm: nexus point for latency models
  - but latency without profiles is pointless…

- we can dump “last-seen” MBB freqs:

```
-mllvm -basic-block-sections=labels \
   -mllvm -mbb-profile-dump=file.csv
```

What next for MLGO infrastructure

- emitc: AOT to .h / .cc: *not even build-time deps!* (D146483)
  - thanks Natasha Kononenko & Jacob Hegna

- a registry for MLModelRunners
  - the initial “development/release” split got more nuanced

- explore variable tensor support, esp. for “interactive mode”
  - ...but scenarios should drive the design

How to get in contact

https://discourse.llvm.org

Tag: mlgo