C Code on ARM

Introduction

The computation graph of Region-based Fully Convolutional Network (R-FCN) is partitioned into two, along the blobs `rfcn_bbox`, `rfcn_cls`, `rpn_cls_prob_reshape`, and `rpn_bbox_pred`. The bottom part is ran on the vector processor (VP, aka CVflow); the top part is ran on the ARM core.

Quick start

Common usage involves running the following commands:

```
$ cd ..
$ bash run_all.sh
$ make
```

Overview

This document is intended to introduce the structure and API of the C codes. This document also describes the steps that can be taken to verify the implementation and the some directions for future improvements.

The following is the hierarchy of the files associated with embedded R-FCN. Please refer to the sister directory at `TODO` for information on running R-FCN in a Python environment.

```
diag/caffe/R-FCN/
  |-- Makefile
  |-- README.md
  |-- README.pdf
  |-- arm/
  |     |-- Makefile
  |     |-- README.md <<< YOU ARE HERE
  |     |-- README.pdf
  |     |-- main.c
  |     |-- blob.h
  |     |-- blob.c
  |     |-- ProposalLayer.h
  |     |-- ProposalLayer.c
  |     |-- PSRoIPoolingLayer.h
  |     |-- PSRoIPoolingLayer.c
  |     |-- sort/
  |     |     +-- (empty)
  |     +-- test/
  |           |-- test0.py
  |           |-- test1.py
  |           |-- test2.py
  |           |-- test3.py
  |           +-- test_all.py
  |-- model/
  |     +-- (empty)
  |-- data/
  |     +-- (empty)
```

```
|-- scripts/
|     |-- download_sorter.sh
|     |-- copy_models.sh
|     +-- copy_reference_input_output.sh
+-- run_all.sh
```

The hierarchy AND the relative location of the sister directory must not be altered for the scripts to work properly.

Structure and API

The proposal layer (ProposalLayer) and position-sensitive pooling layer (PSRoILayer) are chained together by pass blobs among functions. A blob is a `struct` containing its size information (in the order of `(n,c,h,w)`), its type information (`int8_t`, `uint32_t`, etc.), and a `void*` pointer to the block of data in C order. (See `blob.h` for exact definition.)

C files

`ProposalLayer.c` is an almost direct translation (with minor type conversions) of `py-R-FCN/lib/rpn/proposal_layer.py` from the py-R-FCN repo. `setup()` and `backward()` are not used at all, and `reshape()` is done implicitly with C ordering. The verification of `forward()` is detailed in the following section.

`PSRoIPoolingLayer.c` is mostly a translation of `caffe/src/caffe/layers/psroi_pooling_layer.cpp` from the Intel

Caffe repo, which is a C++ implementation of the incorrect (albeit original) CUDA implementation `caffe-rfcn/src/caffe/layers/psroi_pooling_layer.cu` from a fork of Caffe for R-FCN.† The major difference is that the C implementation guarantees that the bins do not pool from overlapping (h,w) pixels, whereas the C++ or CUDA implementations might have different bins pooling from the same (h,w) pixels (despite from different channels). This not only improves performance as only `int` operations are used, but also avoids false sharing when the code is parallelized. Moreover, `int` is used as frequently as possible within the inner loop, as it is more efficient than `float`. The inaccuracies arising from this change, however, should be negligible. It is also noteworthy that the voting step (i.e. average pooling) following PSRoIPooling is combined with PSRoIPooling for performance reasons. Again, only `forward()` is implemented. Verification is less rigorous with the slight change in algorithm.

`main.c` arranges the layers according to the original Caffe model and then applied softmax, followed by post-processing steps. The steps are delineated in `demo()` of `py-R-FCN/tools/demo_rfcn.py`. It is assumed that the input image size is (375,500). Constants are mostly the same as those presented in the reference source code, with the exception that the very last NMS step is tweaked.

Last but not least, non-maximum suppression (NMS), which is used by both `ProposalLayer.c` and `main.c`, is implemented as faithful to `py-R-FCN/lib/nms/py_cpu_nms.py` as possible. Ideas were taken from this repo, but parallelization reminds difficult, as explained in the last section.

Library usage

Sorting routines are cloned from this repo. In particular, quick sort (not to be confused with the `qsort()` function from `stdlib.h`) is used. Please note that little endianness is assumed (and compilation would fail otherwise).

Each element of the array can be considered as a struct of two 16-bit numbers, a score and an index. The array is then sorted according to the score, while the index is read after sorting. Therefore, the initialization of `sort.h` uses `int32_t` as the type, but only the lower 16 bits are used towards the ordering of the elements.

Verification

Due to the large number of custom implementations that feature successive approximations, it is necessary to test the reference implementation with different sets of inputs.

These tests each feature a unique of generating reference outputs and of checking said outputs against those from custom implementations. The tests are indexed and are only referred to by number in the source code.

Index	Input	Reference	Custom	Compari method
0	`rpn_cls_prob_reshape.bin`, `rpn_bbox_pred.bin`	Python	C	absolute error
1	`rfcn_cls.bin`, `rois.bin`	C++	C	mean squared

				squared error
2	`rfcn_bbox.bin`, `rois.bin`	C++	C	mean squared error
3	`cls_score.bin`, `bbox_pred_pre.bin`	Python	C	equivalen (up to machine precision
4	random array	Python	C	equivalen (up to machine precision
4	`rpn_cls_prob_reshape.bin`, `rpn_bbox_pred.bin`, `rfcn_cls.bin`, `rfcn_bbox.bin`	Python, C++	C	mAP@.75
5	`test.bin`	Python, C++	ADES, C	visual inspectio

These tests only run the designated functions and perform the necessary comparisons. The final error metric is printed, which is then compared against an arbitrary, albeit reasonable, threshold.

Notes on reference Python code

The reference Python code from the py-R-FCN repo is relatively easy to read. Pre-processing and post-processing steps are mostly standard, with two exceptions:

- Pre-processing includes upsampling images to a larger size. For example, an input image of (375, 500) is resized to (600, 800). While a larger size generally produces more fine-grained region proposals, it also takes a longer time to process.
- The results from inference are displayed on a per class, per image basis, so a lot of windows are spawned from the script.

Many of the functions unique to F-RCNN or R-FCN are defined in code inside the `src\` directory. They are not well-documented, but tracing the function calls from `tools\demo_rfcn.py` is helpful.

Future improvements

Improving performance

The current C code is not optimized to honor portability and readability. Such a baseline runs for >130ms on one logical thread in Intel Xeon CPU E5-2660 v4 at 1266MHz. A running time below 10ms is conceivable.

SIMD parallelization

The main for loops in the code are written in such a way that it's conducive to SIMD parallelization. However, some modifications are needed for prior to vectorization:

- Return values from `malloc()` are not aligned in heap memory. Use `memalign()`.
- Nested for loops are not combined. This can cause divergence.
- The function `nms()` might or might not benefit from vectorization

of `iou()`, as there is a significant degree of divergence should SIMD instructions be used. One solution is to use soft NMS instead. Another solution is to divide the image into chucks for more parallelism, but handling inter-chunk dependencies can be tricky.

Multi-thread parallelization

The ARM code can benefit from multithreading, if a warp of threads is created at the beginning, and if the threads are reused across multiple runs of the code. However, there are a number of caveats:

- Designating `master`, `single`, and `parallel` code blocks throughout the source can make it very difficult to read.
- False sharing can be a problem, as return values from `malloc()` are not properly aligned.
- `nms()`, which is the most expensive operation, is difficult to parallelize, even after switching to soft NMS. Chunking is the only viable approach here.

Hyperparameter tuning

Increasing the threshold of NMS and reducing the number of potential proposals entertained can improve performance (and jeopardize accuracy). Tuning is required to obtain the most desirable combination of hyperparameters.

Algorithmic approximation

Within NMS, the removal of array elements can be simulated by stable sorting, like so or so. NMS could also be replaced by other more methods. Alternative options are yet to be explored thoroughly.

Improving accuracy

Inaccuracies of the ARM code should be tolerable. However, in the rare scenario where the margin of error needs to be minimized, the follow changes can be made:

Floating point arithmetic

Currently, some floating point calculations are approximated by integer arithmetic. Reverting those changes might be helpful.

Algorithmic approximation

Now, the binning policy in position-sensitive RoI pooling is slightly altered. Reverting it might also be helpful.

Cascading

As described in `../README.md`, cascading can improve accuracy.

Footnote

† The correctness of line 39 is dependent on whether `nthread <= blockDim.x * gridDim.x` holds. If not, `bottom_rois` would be modified more than once from its original value, resulting in an incorrect address. Moreover, dependencies across `for` loop iterations are also discouraged, as it can result in undefined behavior, depending on the memory model used. Hence, while unlikely, such an implementation can be buggy.