Tinsmith: Using a GPU for a chess engine

Graham Jones, www.indriid.com

2019-01-04, January 6, 2019

WORK IN PROGRESS

1 Introduction

I have been working on a chess engine which uses GPUs. I call it Tinsmith. It owes things to Stockfish, Giraffe, and AlphaZero, but is not really like any of them. The code is incomplete: it will be a few months at least before Tinsmith plays its first game. **This account is very preliminary.**

Evaluating a chess position in a typical engine is a tiny amount of work for a GPU. Even for AlphaZero and LC0, it's quite small, and for LC0 at least, large batch sizes are needed. This note is mainly about implementing a tree search in the GPU to deal with last few ply of search, so that a light- or medium-weight evaluation and move-ordering can be used. There is also a main tree used by the host. A leaf in the host tree is a root in a device tree. I'm using NVIDIA GPUs and CUDA.

2 Search algorithm

2.1 Overview

The input to the GPU tree search algorithm is a single board state, and the output is an evaluation of this board and of all its children. The 1-ply evaluations can be converted into a policy vector via a softmax-type formula. This information can be used by the host tree in a PUCT-type search (Rosin 2011), as used by AlphaZero and LC0, for example.

A search tree is built in GPU memory. This data structure is the only global one (global in the sense that it is accessed by different blocks in the same kernel). There are also per-block 'workloads' which are used in the main expand-and-evaluate kernel. The search tree and workloads are discarded when the search is complete. The host can cache the result.

The general idea is to assign a 'budget' of the total number of evaluations to the root node, and divide the budget between children recursively until it runs out. The search algorithm is a 'Shannon type B' search, where promising nodes are explored further. A node has all moves evaluated, the evaluations are converted into positive numbers totalling 1.0 via a softmax-type formula, and the remaining budget is shared out in proportion to these. These numbers can regarded as estimates of the probability that the move belongs to the principal variation (PV) (see the Giraffe paper (Lai 2015) for what that might mean), or the probability that the move constitutes perfect play given the parent node, but I do not take such interpretations very seriously. I think these numbers provide a reasonable way of sharing out the budget. The main aim here is to exploit parallelism in a practical way, not to implemented a particular theory.

Simple negamaxing is used to propagate evaluations back to to the root, except that a special formula is used at nodes which have had some but not all moves evaluated. This may change.

The following subsections describe the algorithm in a bit more detail.

2.2 2-ply budgets

The root node is expanded fully, and the 2-ply moves are all evaluated. These are negamaxed back to 1-ply nodes, and budgets are assigned to 1-ply nodes using these. All 2-ply moves are then assigned budgets using their static evaluations.

Four CUDA kernels are used:

```
onePlyGrow()
twoPlyEvalAndGrow()
make2plyPPVs()
twoPlyPPVsToBlockLIPPVs()
```

2.3 Assigning nodes to workloads

This step assigns leaves (a leaf is a node plus an evaluated move) to a set of ‘workloads’. In the next step, each CUDA block will be responsible for one of these workloads. The aim here is to make the workloads roughly equal (they get similar budgets) so that we won’t spend too long waiting for the slowest blocks to complete.

Some 2-ply leaves will not be expanded further, because their budgets are too small. Some have large budgets that must be divided up. These are expanded to 3-ply and 4-ply leaves as needed. (4-ply seems enough but this could change.) At the end of this step, each workload is a list of leaves with budgets. All leaves in a workload have a budget that is large enough to justify expanding the leaf.

Four CUDA kernels are used:

```
divideStraddling2PlyNodes()
divideStraddling3PlyNodes()
assignStraddling4PlyNodes()
lowPlyLeavesToGrowQ()
```

2.4 Main expand and evaluate

Each CUDA block expands the leaves in its workload into nodes with unevaluated moves. The moves are gathered into groups equal to the number of threads in the block, and evaluated in parallel. A node may have all moves evaluated, and some of these these moves can be added to a new workload. Or a node may run out of budget, in which case it gets a final evaluation. (Currently, I don’t expand a move until all its siblings have been evaluated; a good move ordering might change this.)

This sequence is repeated several times, until there are no new leaves to add to the next workload. It is roughly a breadth-first way of visiting leaves, but it starts with some mixture of 2-ply, 3-ply, and 4-ply leaves, and is stopped by budget limits, not depth.

One big CUDA kernel is used, which dominates the total time:

```
blockSearch()
```

2.5 Negamax back to root

All nodes have final evaluations from the last step. One CUDA kernel is used:

`blockNegamax()`

2.6 Tip node evaluation

When a node runs out of budget, and (as is typical) only some of its moves have been evaluated I use a formula for giving it a final evaluation. This is based on the node's own static evaluation, the move evaluations that have been done, the number of evaluations, and the total number of moves.

Suppose r of n moves have been evaluated, and that $x = \max(r \text{ evals}) + (\text{parent static eval})$. If $f()$ is some parameterised function of r, n, x, statistics can be collected from the evaluation function to estimate the parameters, so that $f()$ matches the value obtained by evaluating all moves as well as possible. I have ideas about the form of $f()$.

2.7 Move ordering

I have ideas but no decision yet on what to do.

2.8 Move evaluation

At least 32 evaluations will be carried out in parallel. Efficient GPU code has other constraints too. On the other hand, there are plenty of possible features, machine learning techniques, and particular formulas, that are suitable. No decision yet on what to do. See below.

3 Code status, Jan 2019

3.1 Move generation

Jul 2018. My first nontrivial CUDA program was a move generator for chess. Not well tested. I don't like it anymore, but it is fast enough not to be a bottleneck for the rest of the engine.

3.2 Tree search

Aug-Oct 2018. In order to focus on the tree search algorithm, I invented a new simple game. Move generation and move evaluation are simple and fast, and there are no terminal nodes. I designed it to have similar statistical properties to chess: similar branching factors, and similar distribution of evaluations. The code for the tree search is mostly complete, both as a serial CPU version (2-ply budgets, Assigning nodes to workloads, Main expand and evaluate, Negamax back to root) and a GPU version (same but missing the Negamax back to root).

Using a GTX 1070 it takes about 2ms for a search tree with 128000 leaves and depth up to about 8-ply. Narrower, deeper trees take longer than others. Essentially all the time is spent building the tree, traversing it, assigning budgets, and organising threads. I have tried to write efficient code, but have not done any code optimization, not even testing different numbers for threads-per-block and number of blocks. 2ms is not great, but I consider it is good enough that this is no longer the most urgent concern.

3.3 Move evaluation

Nov-Dec 2018, continuing. I have been experimenting with various ideas. I am using C++, and guessing how long it will take a GPU to do the inference.

Recently, I have been experimenting with 12 very small neural nets, one for each piece type. A board with 20 occupied squares needs 20 calls to the appropriate ones of these. I have tried 22 binary 'attacked-by' features, 26 binary 'attacks' features, among others. My general idea is that they code 'what the board looks like from this square'.

Training data is made by using python-chess, SF, and LC0 to evaluate board positions. SF and LC0 both evaluate each position using 150ms. I convert centipawns into values (Pr(win) estimates) in [0,1] with an arctan formula. I average the SF and LC0 values to make a target to train against.

I am currently using the limited-memory BFGS algorithm for training, as implemented in libLBFGS (Chokkan, Nocedal). The evaluation and gradient calculation is parallelized over training samples.

4 Refs

```
@article{DBLP:journals/corr/Lai15a,
  author    = {Matthew Lai},
  title     = {Giraffe: Using Deep Reinforcement Learning to Play Chess},
  journal   = {CoRR},
  volume    = {abs/1509.01549},
  year      = {2015},
  url       = {http://arxiv.org/abs/1509.01549},
  archivePrefix = {arXiv},
  eprint    = {1509.01549},
  timestamp = {Mon, 13 Aug 2018 16:47:30 +0200},
  biburl    = {https://dblp.org/rec/bib/journals/corr/Lai15a},
  bibsource = {dblp computer science bibliography, https://dblp.org}
}
```

Rosin, C.D. Ann Math Artif Intell (2011) 61: 203. https://doi.org/10.1007/s10472-011-9258-6

https://github.com/chokkan/liblbfgs

J. Nocedal. Updating Quasi-Newton Matrices with Limited Storage (1980), Mathematics of Computation 35, pp. 773-782.

D.C. Liu and J. Nocedal. On the Limited Memory Method for Large Scale Optimization (1989), Mathematical Programming B, 45, 3, pp. 503-528.