EE 290-002: Hardware for AI

Nelson Lojo

Final Project Proposal

Description / Motivation

Evaluating novel hardware accelerators is currently done by experts that manually implement kernels to support representative workloads. However, some time after the accelerator has existed, these kernels are often optimized, improving the value proposition of the accelerator architecture. This project seeks to make steps toward reducing the delta in the value proposition (i.e. measured performance of) an accelerator design by automatically optimizing kernels by generating schedules with Exo-lang.

Specifically, ***I want to see if leveraging modern code-generation techniques can improve the accuracy of hardware evaluation on model workloads.***

For a minimum result, I need to:

1) Set up an environment to evaluate generated kernels at reasonable speed (anything more than 10s or so would be a deal-breaking bottleneck)
   ⇒ In this vein, I am looking at `spike` to evaluate correctness and count the number of memory accesses as a heuristic for cycle count – ignoring caches entirely. I also have been reading the FireSim docs and watching the tutorials to catch myself up with what I can attempt in AWS. However, this avenue seems expensive to run (I expect *many* hours of training, since FPGA instances do not also have GPUs to do LLM inference)
2) Establish a method for generating schedules to be transpiled by Exo into C kernels
   ⇒ Early ideations were on leveraging AlphaDev, although this would require a notable amount of work to design an "exo game" to mirror the publication's "AssemblyGame." On the other hand, Atlan Haan (not in this course) suggested using LLMs to generate schedules. LLMs are favorable mostly due to the lack of up-front work necessary to get to generated code (even if it's likely incorrect). That said, even within this branch (using LLMs) there is the choice between fine-tuning an LLM or optimizing a prompt for a high-performance LLM (e.g. GPT 4, Claude 3 Opus)
3) Loop feedback back into the schedule generation scheme and search/learn
   ⇒ This is highly dependent on (1), as it requires a signal to loop back in the first place. In the same vein, the time interval at which this loop is performed will determine the approach of schedule generation that is viable within the semester and with a student's wallet. If it takes seconds to evaluate a correct kernel, then AlphaDev is entirely impossible to train. Once we hit ~10 seconds on 20 vcores, then fine tuning an LLM is too expensive, as instances with LLM-ready GPUs often don't have much more compute budget to run alongside inferencing an LLM and thus the time necessary to train such a model would rack up the bill too much. The final (albeit least interesting) approach is to manually prompt a high-performance LLM until it

somewhat reliably produces valid Exo schedules and gradually generate better schedules. The feedback signal would have to be served together with the unscheduled kernel for each in-context example.

Relevant Research

AlphaDev is the primary inspiration for this work, where an encoder-decoder transformer architecture guides a Monte Carlo tree search where nodes of the tree are programs. The key driver in the paper that caught my attention was that the network was not encoded with a closed-form representation for programs, but rather learned it as a consequence of end-to-end optimization over their reward. Nature Link: https://www.nature.com/articles/s41586-023-06004-9

I hope to either extend this work into schedule generation for Exo – which requires a more ingenious approach to formulating schedule generation as a turn-based game with a finite action space – or apply the end-to-end representation learning takeaway to LLM finetuning – to have a sequence model not only encapsulate the state encoding and choice steering, but also the MCTS algorithm in its entirety (and thus open the possibility to optimizing it out).

Roadmap

Checkpoint 1 (4/12):

Set up an automated flow for getting a schedule to run with `spike`. Additionally, extract trivial diagnostics such as number of instructions run and number of total memory fetches made.

Checkpoint 2 (4/26):

Standardize the automated flow to deploy into the cloud with GPU-accelerated instances. Determine which of the three approaches to move forward with and implement the full loop.

Final Report:

Report on generated kernel performance compared to both naive implementations and reference implementations for Gemmini.