Automated CUDA Kernel Generation via Reinforcement Learning on LLMs

Abhishek Vangipuram William Huang

{abhiv, willsh}@stanford.edu

Problem

As deep neural networks get larger and larger, inference and training time has considerably increased. There has been much research to reduce this bottleneck, from making smaller models more efficient, to more sophisticated training and inference procedures.

```python
class Model(nn.Module):
    """
    Simple model that performs a ReLU activation.
    """
    def __init__(self):
        super(Model, self).__init__()

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        """
        Applies ReLU activation to the input tensor.

        Args:
            x (torch.Tensor): Input tensor of any shape.

        Returns:
            torch.Tensor: Output tensor with ReLU applied, same shape as input.
        """
        return torch.relu(x)
```

```cpp
template <typename scalar_t> __global__ void relu_kernel(scalar_t* __restrict__ output,
                                                         const scalar_t* __restrict__ input,
                                                         const int64_t size) {
    const int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < size) {
        output[idx] = input[idx] > 0 ? input[idx] : 0;
    }
}

// PyTorch wrapper function
torch::Tensor forward(torch::Tensor input) {
    auto output = torch::empty_like(input);

    const int threads = 256;
    const int blocks = (input.numel() + threads - 1) / threads;

    AT_DISPATCH_FLOATING_TYPES(input.type(), "relu_kernel", ([&] {
        relu_kernel<scalar_t><<<blocks, threads>>>(
            output.data_ptr<scalar_t>(),
            input.data_ptr<scalar_t>(),
            input.numel()
            );
        }));

    return output;
}

PYBIND11_MODULE(TORCH_EXTENSION_NAME, m) {
m.def("forward", &forward, "ReLU forward (CUDA)");
}
```

Figure 1. Here is an example of PyTorch (left) and CUDA (right) code for simply applying a ReLU

One aspect that some are trying to tackle is improving efficiency at the CUDA level. CUDA is the backbone of PyTorch that is what actually runs on NVIDIA GPUs. It is well known that the initial PyTorch CUDA implementations are not built for speed or efficiency. Thus, we could improve the scalability of deep neural networks if we are able to get efficient CUDA kernels from easily writeable PyTorch code. Moreover, CUDA kernels are notoriously annoying to write. This motivates our driving question: **Can a base LLM be trained via reinforcement learning to convert PyTorch modules to CUDA kernels more effectively than just a one-shot query?**

Related Work

The following include some recent advances in the space of CUDA kernel optimization and automation.

- Thunderkittens: a framework that makes it easier to write fast deep learning kernels in CUDA. Improvements like this at the CUDA level have helped inspire our project.
- KernelBench: a dataset of 270 PyTorch modules split into levels 1,2,3,4, with each level increasing in complexity. The goal of this dataset is to provide a standardized benchmark for automated PyTorch module to CUDA kernel generation. Funnily enough this was released right after we submitted our project proposal.
- Sakana-AI AI-CUDA-Engineer: this is an attempt at automatic CUDA kernel generation from PyTorch modules that is mainly done via continual reprompting as genetic algorithms. of an intelligent base model (o1 or R1).

Methods

- **CUDA Evaluation Harness:** We constructed a custom test harness to take CUDA code, load it as a PyTorch module, and evaluate it on several inputs.
- **Test Time Training:** We utilize a similar approach to test time training in our training procedure. Rather than attempting to solve a general CUDA generation problem (which would be very hard), for each task we attempt to solve just this one task using RL. Doing this helps simplify our objective and potentially provide better results.
- **Initial LLM Querying:** We make a detailed custom initial prompt to the LLM to get outputs in a correct format, a custom parser to read this output, and also attempt multiple queries in order to get a better response.

- **PEFT via LoRA:** Both for effective training and due to compute constraints, we utilize PEFT. Specifically, we use Low-Rank Adaptation (LoRA) with the LLM models in order to train a significantly smaller subset of the parameters and still keeping high flexibility.

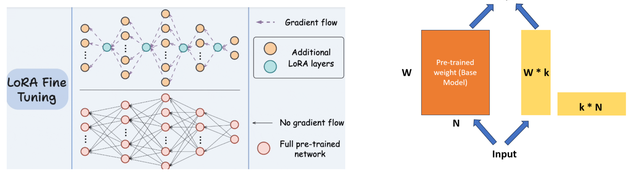

Figure 2. LoRA allows us to efficiently update the model separately from pre–trained weights, through a low-rank weight update of $\Delta W = AB$

- **mini-GRPO:** We utilize Group Relative Policy Optimization, which is a modification to the traditional Proximal Policy optimization algorithm but does not need a learned value model. Instead, the value function baseline is just the average reward of multiple outputs from the same query. We only use 2 to 3 outputs for our case due to constraints.

Here is the reward function we use for our task:

$$r(x) = \begin{cases} -1 & \text{output not in parsable format} \\ \max(-1, -\#\text{errors}/10) & \text{CUDA compilation errors} \\ -1/10 & \text{Harness (non-CUDA) error} \\ \mathbf{1}[\text{output is correct}] & \text{CUDA kernel fully compiles} \end{cases} \quad (1)$$

This is in order to separate differing levels of poor output from the model, and we choose to scale the reward such that $|r(x)| \leq 1$ for stability reasons. In general we could have these rewards be larger in scale, and may be something to test in the future.

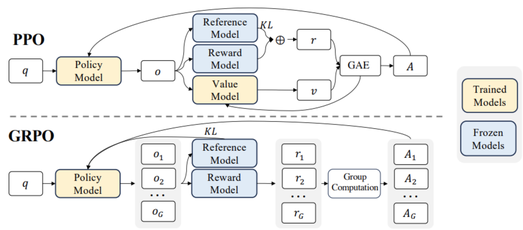

Figure 3. Demonstration of PPO and our GRPO. GRPO foregoes the value model, instead estimating the baseline from group scores, significantly reducing training resources.

We use non-stateful optimizers such as SGD and Adafactor for the gradient steps. This is to conserve GPU memory, as optimizers such as Adam and AdamW would require $2\times$ or more memory to store internal gradient averages.

Experiments

Data: We obtained PyTorch problems from the KernelBench set, which has 270 such problems. We are focused on a few problems from the Level 1 set.

Task: The task we aimed to solve was correct conversion and code generation from PyTorch code to CUDA code. This included getting the model to output CUDA code in the desired format, for it to compile, be correct, and lastly we check for performance.

Models: We plan on testing a number of small models, due to limited GPU capacity. These models include Deepseek-R1-Distill-Qwen1.5B (with and without reasoning) and CodeLlama-7B-Instruct. For this poster, we just have tested Deepseek-R1-Distill-Qwen1.5B without reasoning.

Preliminary Results

Configuration	Average Reward
Base Model	−0.88
Base Model + Reprompt	−0.51
Base Model + Reprompt + RL	−0.34

Table 1. Average Reward for Different Training Configurations on a few problems from the Level 1 set. We used Deepseek-R1-Distill-Qwen1.5B and limit thinking tokens due to compute constraints.

Analysis

- We found that the initial models are quite bad at getting even compilable code, so most of our time was spent on getting compilable and correct kernels, with high speed being achievable with continued training on larger GPUs.
- Re-prompting with the error message was surprisingly effective at solving compilation and parsing errors.
- Reinforcement learning was able to achieve higher rewards a greater proportion of the time. In several instances we are able to compile CUDA kernels, but not many have correct output.

Conclusion

- We use a number of low-memory techniques and combine prompting and RL to achieve better results for CUDA kernel generation.
- While most of our attempts did not result in optimized CUDA kernels, we were able to achieve progress on reducing parsing and CUDA compilation errors, achieving some functional CUDA kernels.
- Avenues for future work include scaling up these strategies on larger GPUs (allowing for larger models and use of more reasoning tokens), expanding on the reprompting strategy to allow for longer trajectories of reprompts, and optimizing on performance of CUDA kernels via these rewards.

References

[1] Anne Ouyang, Simon Guo, and Azalia Mirhoseini. Kernelbench: Can llms write gpu kernels?, 2024.