ECS 251 Project Proposal

Anders Museth, Zachery Fogg

Paper Introduction

Problem Statement. Deep learning training performance is traditionally analyzed through the lens of algorithmic efficiency or raw hardware throughput. However, a significant portion of training latency is likely consumed by system-level overheads that are often overlooked: data loading latency, process management context switching, and the communication cost of transferring data across the PCIe bus to the GPU. As hardware accelerators, like GPUs, become faster, the CPU and OS might increasingly become bottlenecks in neural network training. If the OS spends more time dispatching kernels or context-switching between data workers than the GPU spends executing instructions, expensive accelerator resources are not being efficiently utilized. We seek to isolate and quantify these specific OS and deep learning framework (e.g. PyTorch) overheads to determine where optimization efforts, if any, should be prioritized across different model training pipelines (e.g. small vs. large models).

Shortcomings of Existing Analysis Methods. While standard profiling tools (such as PyTorch Profiler) are able to visualize GPU kernel execution, they often treat the rest of the system (OS, CPU, RAM) as a monolith. These tools fail to separate framework logic from OS-level costs, such as context switching, and interrupt handling. To address this limitation, we will utilize NVIDIA Nsight Systems (a tool for system-wide performance analyses) as our primary analysis tool. This allows us to explicitly correlate OS overheads with GPU kernel launches and isolate system overhead from the model training pipeline.

Proposed Study. We propose a study that segments the training pipeline into three distinct categories where overhead can occur: Data loading, CPU → GPU data transfer, and Kernel dispatch latency.

1. Data Loading. Data loading encompasses the gathering and preprocessing of a training batch and preparing it to be copied to GPU memory. PyTorch provides mechanisms for parallelizing this process with multiple threads, which we will use to investigate this hypothesis. We will compare single-threaded data loading versus multi-threaded data loading to measure the specific cost of OS context switching and I/O delays, testing the hypothesis that this cost disproportionately affects small models that have significantly faster training steps.

2. CPU → GPU Data Transfer. CPU → GPU data transfer analyses will involve measuring the latency of moving the data from main memory to GPU memory. We will evaluate the OS and memory translation overhead by contrasting two transfer patterns: high-frequency, small-payload (e.g. small batch size) transfers versus consolidated large-batch transfers. We will then replicate these benchmarks using page-locked (pinned) memory.

3. Kernel Dispatch Latency. Kernel dispatch latency analyses will investigate the efficiency of the CPU dispatching CUDA kernels to the GPU and the GPU executing those kernels. We will compare standard eager execution of PyTorch code against compiled computation graphs (using torch.compile) to quantify the OS overhead of kernel launching versus actual GPU execution time of the kernel operations and test these differences as model/batch sizes scale up.

Challenges. The primary challenge in this proposed study is isolation. Modern deep learning frameworks like PyTorch are highly asynchronous; the CPU dispatches kernels to the GPU and continues execution, making it difficult to pinpoint exactly when a system call or page fault triggers a delay in GPU execution. Furthermore, distinguishing between software overhead (e.g., training data preparation) and hardware limits (e.g., physical PCIe bandwidth saturation) requires careful analysis that goes deeper than standard high-level APIs allow for.

Anticipated Results. We hypothesize that for small models, system overheads (specifically kernel dispatch and data worker context switching) will be the dominant bottleneck, often exceeding actual GPU computation time. Conversely, for large models, we anticipate these overheads will be negligible, effectively hidden by long GPU computation times. We expect to demonstrate that naively increasing the number of data loading workers will yield diminishing returns more rapidly on small models due to context-switching costs. Furthermore, we predict torch.compile will significantly reduce the "dispatch tax" proportional to the number of kernels a model forward pass requires, instead of instead of

being merely proportional to model or training batch size

Detailed Timeline

2/2/26 - 2/9/26: Implement a PyTorch model training pipeline for an MNIST neural network that will provide the environment for carrying out various analyses. We will implement logic to test different sizes of models, data loading setups, and methods for creating CUDA computation graphs (i.e. using or not using torch.compile).

2/9/26 - 2/16/26: Get familiar with using Nvidia Nsight and begin analyses with a "base case" network (single linear with non-linear activation function) to build up the infrastructure for conducting further analyses.

2/16/26 - 2/24/26: Begin testing various dataloading configurations (multiple workers, memory pinning, etc.) for our training pipelines and analyze the CPU → GPU datapath speeds and sources of communication overhead.

2/24/26 - 3/2/26: Analyze the kernel launches of eagerly executed model forward passes versus models that have been compiled with torch.compile, testing on multiple model/data size scales.

3/2/26 - 3/9/26: Finalizing results and preparing our class presentation.

3/9/26 - 3/18/26: Writing project final report

Evaluation Plan

Decompose the total execution time of the key training stages (data loading, CPU → GPU data transfer, and CUDA kernel dispatches) and quantify the latency caused by the hardware and the operating system in different model training scenarios (e.g. small vs. large models, small vs. large training batches). We will create tables showing the percentage breakdowns of hardware and software bottlenecks for each of these different stages. Along with these tables, we will have a 'system calls' table ranking the most expensive OS calls that were commonly used.