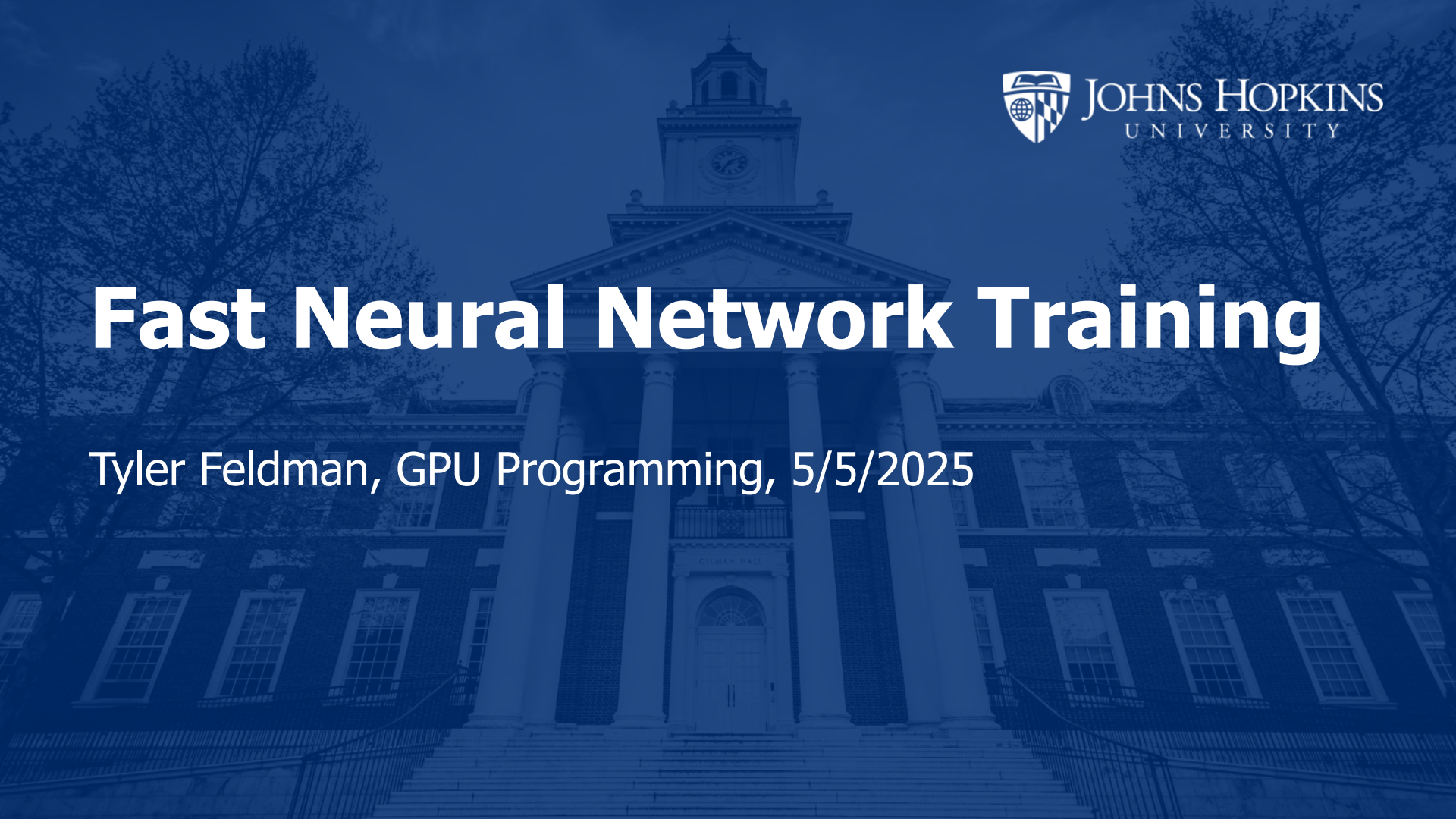


Fast Neural Network Training

Tyler Feldman, GPU Programming, 5/5/2025

Purpose and Goals

- Training large neural networks takes a long time!
  - Estimated ~70 days for llama 3.1 405B parameter model [How Long Does It Take to Train the LLM From Scratch? | Towards Data Science]
  - Training compute of frontier AI models growing by 4-5x per year [Training Compute of Frontier AI Models Grows by 4-5x per Year | Epoch AI]
- Over long periods of time, any performance optimizations can matter quite a lot
- Note there are many alternative methods to save training runtime, including innovations in network architecture, predictable scaling [2303.08774], hardware improvements for faster data transfer, etc
- **Goal: explore various optimizations we can make to the software side of neural network training to improve runtimes**

Optimizations Identified

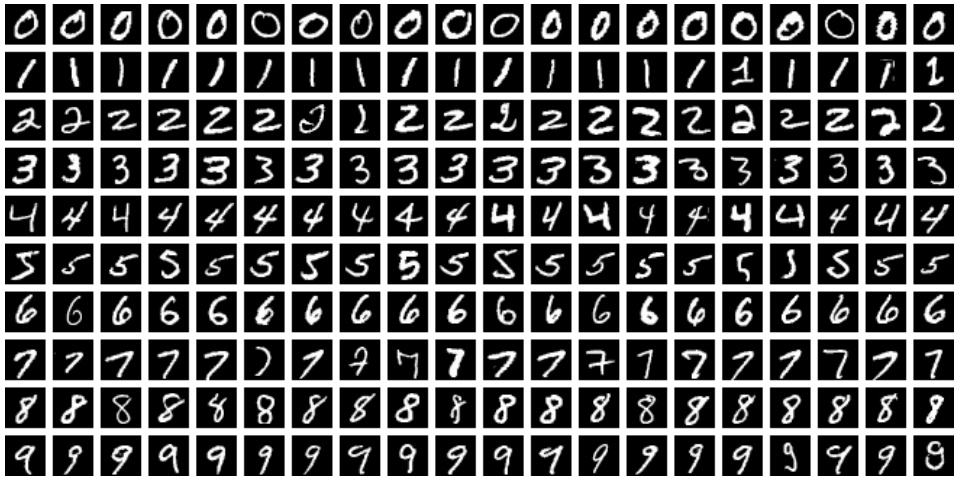
- Kernel fusion
- Reducing synchronization with CPU
- Reducing kernel launch latency through CUDA Graphs
- Using pinned memory for CPU memory
- Maximizing batch size
- Leveraging optimized libraries for NN operations (e.g. GEMM, cuBLAS, cuDNN)
- Memory access patterns: shared memory usage, data layout
- Adjusting data types (e.g. using FP16 for faster operations)
- Launch configuration tuning: grid dimensions and block dimensions

Project Approach: Iterative Improvement

- Dataset: MNIST image dataset
- Neural network architecture: simple single layer fully connected with softmax and cross entropy loss
- Got the network working and fitting on the dataset, then optimized
- Iterative profiling and adding features to improve runtimes
- Implementation of:
  - Baseline CUDA/C++ approach
  - Optimized CUDA/C++ approach
  - Python numpy CPU approach
  - Python pytorch GPU-accelerated approach
  - Python pytorch GPU-accelerated optimized approach

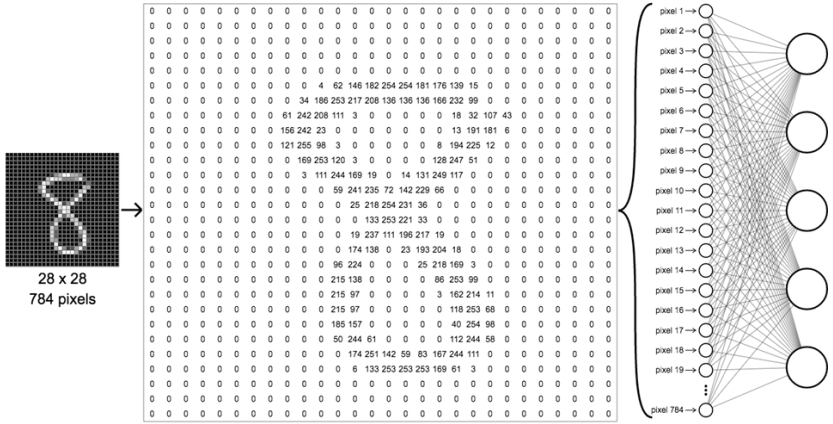
Dataset: MNIST

- Handwritten digits from 0-9
- Relatively old but standard machine learning dataset
- 60k training samples, 10k testing samples
- Each sample is 28x28 grayscale
- Source: MNIST Dataset



Data Preprocessing and Representation

- Adapted an open-source parser for MNIST dataset, loads in entire dataset as uint8_t into a contiguous block of memory
- I then normalize the data to 0-1 and convert to float
- Since we are using a fully connected network, the images are represented as a 784-item float vector



https://ml4a.github.io/ml4a/neural_networks/

Approach – Baseline CUDA / C++

- Vanilla gradient descent
- Using mini batch stochastic gradient descent for better convergence
- Repeat: Matmul -> softmax -> loss -> accuracy -> softmax backprop -> weight gradient backprop -> update weights
- Computing loss and accuracy each epoch and reporting the results

Approach – Optimized CUDA / C++

- Reduced synchronization and memcpy's to/from host and device (e.g. for loss calculations)
- Implemented forward and backward pass matrix multiplication with cuBLAS Sgemm
- Only computing loss and accuracy every 10 epochs
- Fused loss + accuracy computation kernel
- Utilized cuDNN's softmax function
- CUDA graphs to reduce launch latency

Approach – Python Implementations

- Numpy CPU: basic gradient descent, vectorized numpy operations
- PyTorch GPU: Uses PyTorch logistic regression model, CrossEntropyLoss, stochastic gradient descent, dataloader with 128 batch size, accesses loss every epoch
- PyTorch GPU Optimized:
  - Torch.compile()
  - Automatic mixed precision (AMP)
  - Preloading all data to GPU
  - Pinned host memory

Notes on Timing Results

- Using an approximate average timing per epoch taken from the later half of the training process to allow for GPU warmup
- Also using CPU wall clock time using chrono
- Not including any initialization timing (e.g. transferring training data to GPU or building PyTorch network)

Results

Implementation Description	Average Epoch Runtime	Total Wall Clock Time
Baseline CUDA/C++	~38 ms	874 ms
+ reducing synchronization and memcpy to host	~27 ms	554 ms
+ cuBLAS matrix multiplication	~20.73 ms	439 ms
+ only performing loss & accuracy calculation every 10 epochs	~19 ms	398 ms
+ CUDNN implementation of softmax	~18 ms	386 ms
+ CUDA Graphs to minimize kernel launch latencies	~17 ms	367 ms
Python + Numpy (CPU)	~12500 ms	251360 ms
Python + Pytorch (GPU)	~10500 ms	211260 mss
+ torch.compile(), pinned memory, AMP, preloading dataset	~600 ms	14610 ms

Code Demo

- Walkthrough of repo structure
- Walkthrough of main executing code
- Compilation
- Running

CUDA Concepts and Libraries Used

- CUDA libraries: cuRAND, cuDNN, cuBLAS
- CUDA events for recording timing information
- 2D grid and block sizes (e.g. matmul kernel)
- CUDA streams
- CUDA Graphs (not in class)

Conclusions

- Exploring optimizations for neural network training is very important
- PyTorch offers quite a lot in terms of optimizations that are very easy to add (e.g. torch.compile, AMP)
- CUDA library functionality is great for speedups – cuBLAS and cuDNN
- Quite a big difference comparing optimized Python PyTorch to my optimized C++
- Learned a lot about neural network optimizations for training and inference

Resources

- Dataset
  - MNIST Dataset
  - GitHub - wichtounet/mnist: Simple C++ reader for MNIST dataset
- CUDA libraries
  - cuBLAS: 1. Introduction — cuBLAS 12.9 documentation
  - cuDNN Softmax: cudnn_ops Library — NVIDIA cuDNN Backend
- CUDA Graphs
  - CUDA Runtime API :: CUDA Toolkit Documentation
  - Getting Started with CUDA Graphs | NVIDIA Technical Blog
  - 1. Introduction — CUDA C++ Programming Guide
- PyTorch
  - Introduction to torch.compile — PyTorch Tutorials 2.7.0+cu126 documentation
  - Automatic Mixed Precision package - torch.amp — PyTorch 2.7 documentation