High-Performance Data Loading and Augmentation for Deep Neural Network Training

Trevor Gale
tgale@ece.neu.edu

Steven Eliuk
steven.eliuk@gmail.com

Cameron Upright
c.upright@samsung.com
Roadmap

1. The General-Purpose Acceleration Framework (GPAF) project
   1. Systems & software
   2. Key features

2. Data loading & augmentation systems
   1. The data loading & augmentation task
   2. Motivation for our system

3. Data augmentation pipeline implementation
   1. Data augmentation on CPU & GPU
   2. Multi-threading augmentation on CPU
   3. Automatic performance tuning
   4. Memory management
   5. Levels of parallelism

4. Results & analysis
General-Purpose Acceleration Framework Project
Systems & Software

Goal: Design & build software & hardware infrastructure to accelerate machine learning and mathematical workloads. Specifically through the use of many-GPU, distributed systems

Hardware
Custom GPU clusters used across Samsung

Software
Distributed math library (dMath). Used to accelerate popular machine learning frameworks
- Kaldi speech recognition toolkit
- Caffe deep learning framework

Samsung Advanced Learning v1.0

Caffe + = Expresso (topic for today)
Key Features

- Pooled memory management, avoid costly allocation, de-allocation, and registration for RDMA transfers
- Asynchronous replication of shared data, overlapping parameter distribution w/ forward pass computation
- Caching of distributed job metadata to minimize overhead when starting common tasks
- Multi-threaded, asynchronous, CPU/GPU data loading pipeline, automatically tuned at runtime (topic for today)
- Highly optimized DNN operations (cuDNN, cuBLAS, custom combined backward convolution)
- Distributed batch norm (strict or relaxed)
- Half-precision support as storage and computation on supporting hardware

(See EDMNN@NIPS, GTC 2016 talk)
Data Loading & Augmentation System
Design & Motivation
Data Loading & Augmentation Task

1. Load images from database
2. Decode image
3. Perform any data augmentation
4. Copy image to the GPU for training
Typical Augmentation System

- Multiple threads are used to accelerate data augmentation
- Data loading for next batch is done in parallel with the forward-backward pass for the previous batch
Motivation for Our System

• Advances in GPUs and systems for deep learning have accelerated DNN training to the point where data loading and augmentation can be the main bottleneck.

• The typical approach of multi-threading and overlapping preprocessing with training on the previous batch is no longer sufficient for some networks.

Peaks training speed (bars) for AlexNet on 8 GPUs. Dotted lines mark peak data loading speeds with 5 threads / GPU.
Question

• How can we accelerate data loading & augmentation so that we can continue to leverage training speedups from the latest GPUs and software libraries?

Our solution
• Utilize the GPU for data augmentation
Problem

- Data loading is only the main bottleneck for some networks. How can we accelerate data loading with the GPU when necessary, but avoid wasting GPU resources on data augmentation when data loading is not the main bottleneck?
Goal for Our System

- We need a data loading & augmentation system that can adapt to the computational needs of the network so that we can continue to leverage training speedups from the latest GPUs & software systems.

Source: developer.nvidia.com/cudnn
Data Augmentation Pipeline Implementation
Key Features

1. Data augmentation on CPU & GPU
2. Multi-threading augmentation on CPU
3. Automatic performance tuning
4. Memory management
5. Levels of parallelism
Data Augmentation on CPU & GPU

- Central augmentation pipeline composed of data augmentation operations on each worker process
- All operations implemented on CPU and GPU
  - Used BLAS, cuBLAS, and OpenCV CPU/GPU to build fast data augmentation operations
- Operations are moved between CPU and GPU between batches to avoid thread safety issues

(note: we refer to the stage of the pipeline at which the data is moved to the GPU as the transfer index)
Multi-Threading Augmentation on CPU

- Multiple threads are used by each worker to augment data on the CPU.
- Number of worker threads is the same across the workers and is managed centrally.
- The transfer index and the number of threads per worker are managed centrally by the master process for all workers to avoid resource imbalances.
- Each batch is prepared in parallel with training on the previous batch.
Automatic Performance Tuning

- User would need to try \( \text{max\_num\_threads} \times (\text{num\_ops} + 1) \) settings to find the optimal number of threads & transfer index.
  - Harms experimentation speed.
  - Could be waste of time for some networks where data loading is insignificant compared to network training (e.g. very deep networks).

However, state space is small enough for us to search at runtime programatically.
Automatic Performance Tuning

- At runtime, master process samples performance for all different combinations of thread counts and transfer indices (referred to as states).
- Samples for each state are taken over batches. We take N samples for each state, where $N = \text{ceil}(128 / \text{batch\_size\_per\_worker})$.
- The ideal setting is found by selecting the lowest runtime from the medians of the N samples for each state.
Memory Management

Page-locked host memory allocations and device memory allocations cause implicit synchronization. We need to avoid synchronization with the GPU so that we do not interfere with network training

- Host & device buffers are allocated by each thread on startup and only resized when samples exceed the current buffer size
- Page-locked host memory is used to allow overlap of transfers to device with computation on device
- Data types are promoted lazily to avoid unnecessary data transfers
Lazy data type promotion
• RGB images are loaded in uint8
• Operations like mean subtraction and scaling of data are very sensitive to precisions and cannot be done in uint8
• Rather than performing all ops in float when mean sub or scale are present, we wait until the higher precision is needed to promote the data type
Memory Management

Benefits

- We can avoid unnecessary memory transfers to GPU when mean subtraction is moved to GPU

- Avoids 4x increase in traffic to the GPU
- Auto-tuning frequently achieves significant performance improvements by
  a. Moving high-precision ops to GPU (saves data transfer)
  b. Moving computationally intensive ops to GPU (slow on CPU)
Levels of Parallelism

1. The processing pipeline is replicated across the workers and controlled centrally by the master process.
2. Within each worker, data augmentation is threaded:
   • Threads load from central DB to ensure replicable training and testing results.
Levels of Parallelism

3. Within each thread, processing on host is overlapped with transfers to device and computation on GPU
   • Pinned memory is used for host buffers
   • Host buffers are associated with CUDA events
   • Device buffers are associated with CUDA streams
   • Thread keeps multiple of each buffer and ping-pongs between them
Levels of Parallelism

For each sample:
1. Select next host & device buffers
2. Block on host buffer event (confirm any previous copy is complete)
3. Fill host buffer with training sample
4. Run host-side augmentation
5. Block on dev buffer stream (confirm all work in this buffer is complete)
6. Start async copy from host buffer to dev buffer
7. Enqueue host buffer event in dev buffer stream
8. Launch all dev-side augmentation & final async copy into batch
Performance Results & Analysis
Peak Data Loading Performance

Peak training speed (bars) for AlexNet on 8 GPUs. Dotted lines mark peak data loading speeds with 5 threads / GPU

Peak Processing Rates With Our System

• Crop/mirror/meansub: 18910 FPS (2.13x speedup)
• Crop/mirror/meansub/interp/colordist: 9475 FPS (2.45x speedup)
Results With AlexNet

- **DataLoader V2** (orange) is system described in this talk
- **DataLoader V1** (dark blue) is previous system
  - Extremely pipelined: crop is done as copy into GPU memory, single kernel does mean sub, scale, mirror
  - Only supports crop, mean subtraction, scale, mirror
  - 1 thread per worker
- **No DataLoader** (light blue) is training speed with dummy data
- Augmentation pipeline is basic crop, mirror, and mean subtraction
Results With AlexNet

Data loader V2 provides 19.4% speedup on average.
Performance gain for more complex pipelines is likely to be larger.
Move from M40 to P100 significantly increased the problem. Likely to continue to increase as systems advance.
There is still room for improvement: data loader v2 provides 87.4% of peak performance on average.
Results With AlexNet

Looking at auto-tuner decisions (*num_threads/transfer_index*)

- Auto-tuner generally moves augmentation to GPU before threading
- GPU side augmentation + high thread counts are chosen as the amount of work increases
Results With AlexNet

Room for improvement
• New data loader peak w/ crop, mirror, mean sub is 18910 FPS

Questions
1. What is stopping us from matching peak performance with no data loader?
2. Max speed is not too far off our peak, what else can we do to improve?
Question 1: What is stopping us from matching peak performance with no data loader?

Contention for resources with training
- Large amount of data moving to each worker from host CPU
- Many data augmentation kernels competing with network computation
Question 2: Max speed on P100 is relatively close to our peak, what else can we do to improve?

- Lock-free system for loading data from the DB
- Optimization of some operations (avoiding synchronous stuff, better kernels)
- Work to increase overlap between augmentation and training
- Reduce system overhead (stalling threads between batches, global sync to get performance samples, etc.)
- Decoding images on GPU

Note: because data augmentation can be done on GPU, it will be able to leverage speedups from new GPUs as well
Results With GoogleNet

GoogleNet on 8 M40 GPUs

- Network is ~3x more difficult to train than AlexNet. Data loading is not a bottleneck yet
- 6.5% average speedup from reduced device side synchronization
Takeaways

Benefits
• 15.5% speedup on average across our experiments
  • Reduced interference with network training
  • Reduced memory transfers
  • GPU-side augmentation to accelerate complex pipelines
• Automatic performance tuning finds optimal performance settings, reducing user burden

Room for improvement
• Dedicated network between GPUs (NVLink!)
• Optimizing augmentation operations
• Reducing system overhead
Thank You.
Expresso AlexNet training speed compared to BVLC Caffe (12/09/16). 2.12x speedup on average