Effectively Scaling out
Deep Learning Frameworks with GPUs

Hars Vardhan
On Behalf of Dr. Steven Eliuk
Computer Science Innovation Center
Samsung Research America
Feb 5th 2017
Outline

i. Why Scale?
   i. Motivation

ii. Current Open-Source offerings
   i. Our sentiments

iii. Bottlenecks
   i. Matrix Multiplication
   ii. Data Loading & Augmentation

iv. Tips and Tricks

v. Real-World Results
   i. Strong & weak scaling, training time

vi. HPC Environment
   i. What makes for an effective environment?

vii. Q & A
Deep Learning: utilized in an ever growing number of projects throughout Samsung, from s/w to h/w.
Long Training Times Impact Time to Market

Effect of Experiment Time

Minutes, Hours
- Interactive investigation! Instant gratification!
- Parameter exploration

1-4 Days
- Tolerable
- Interactivity replaced by parallelization of experiments

1-4 Weeks
- High value experiments only
- Progress stalls

> 1 Month
- Don’t even try

Keynote at 2015 NVIDIA GTC Conference, Jeff Dean
Current Open-Source Offerings:

<table>
<thead>
<tr>
<th>Single-GPU</th>
<th>Multi-GPU</th>
<th>Distributed</th>
</tr>
</thead>
<tbody>
<tr>
<td>BVLC / caffe</td>
<td>TensorFlow</td>
<td>(CNTK)</td>
</tr>
<tr>
<td>torch</td>
<td>theano</td>
<td>dmlc mxnet</td>
</tr>
</tbody>
</table>

Past
Current Open-Source Offerings:

<table>
<thead>
<tr>
<th>Single-GPU</th>
<th>Multi-GPU</th>
<th>Distributed</th>
</tr>
</thead>
<tbody>
<tr>
<td>BVLC / caffe</td>
<td>torch</td>
<td>TensorFlow (CNTK)</td>
</tr>
</tbody>
</table>

Now

Samsung Research America – Silicon Valley
Deep Learning Platform: Framework

<table>
<thead>
<tr>
<th>Current Approaches</th>
<th>Samsung – SRA Approach</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Application</strong></td>
<td><strong>Frameworks (DNN – Kaldi ASR, Caffe AIR, Theano, Torch, etc)</strong></td>
</tr>
<tr>
<td>LAPACK</td>
<td>LAPACK</td>
</tr>
<tr>
<td>FFT</td>
<td>FFT</td>
</tr>
<tr>
<td>General Purpose Math</td>
<td>General Purpose Math</td>
</tr>
<tr>
<td>cuBLAS</td>
<td>cuBLAS</td>
</tr>
<tr>
<td>cuDNN</td>
<td>cuDNN</td>
</tr>
<tr>
<td>CUDA</td>
<td>CUDA</td>
</tr>
</tbody>
</table>

**Node 1**
- GPU #1
- GPU #16

**Node N**
- GPU #1
- GPU #16

- Does not “Scales Out” effectively
- Accuracy can degrade
- Small models
- Tractability issues

- “Scales Out” Effectively
- Comprehensive Library Developed
- Larger models – frontier research
- Extended feature set
dMath : A distributed math library

- Abstracted away complicated distributed & multi-GPU programming from high-level Machine Learning user
- Uses MPI-based client-server computing paradigm
- Has in-device (GPU) data storage
- Customized routine to exploit PCI switching and underlying HW
- Full support for DNN pipeline
- Integration w/ existing open-source frameworks.
Matrix data is divided into multiple blocks and distributed across the workers.

On the left are four examples of distributing the same matrix across four workers, the numbers represent the worker that stores that block.

Block size and location are important for efficiency.

Most algorithms require consistent block sizes, some require consistent layout on the workers.

Arbitrary layouts are supported in dMath.
Data Layout and Algorithm Efficiency

Efficient, no communication between GPUs

\[
\begin{array}{cccc}
0 & 1 & 2 & 3 \\
0 & 1 & 2 & 3 \\
0 & 1 & 2 & 3 \\
0 & 1 & 2 & 3 \\
\end{array}
+ \begin{array}{cccc}
0 & 1 & 2 & 3 \\
0 & 1 & 2 & 3 \\
0 & 1 & 2 & 3 \\
0 & 1 & 2 & 3 \\
\end{array}
= \begin{array}{cccc}
0 & 1 & 2 & 3 \\
0 & 1 & 2 & 3 \\
0 & 1 & 2 & 3 \\
0 & 1 & 2 & 3 \\
\end{array}
\]

Inefficient, off diagonal blocks require communication between GPUs

\[
\begin{array}{cccc}
0 & 1 & 2 & 3 \\
0 & 1 & 2 & 3 \\
0 & 1 & 2 & 3 \\
0 & 1 & 2 & 3 \\
\end{array}
+ \begin{array}{cccc}
0 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 \\
2 & 2 & 2 & 2 \\
3 & 3 & 3 & 3 \\
\end{array}
= \begin{array}{cccc}
0 & 1 & 2 & 3 \\
0 & 1 & 2 & 3 \\
0 & 1 & 2 & 3 \\
0 & 1 & 2 & 3 \\
\end{array}
\]

Efficient, no communication between GPUs

\[
\begin{array}{cccc}
0 & 1 & 2 & 3 \\
0 & 1 & 2 & 3 \\
0 & 1 & 2 & 3 \\
0 & 1 & 2 & 3 \\
\end{array}
= \begin{array}{cccc}
0 & 1 & 2 & 3 \\
0 & 1 & 2 & 3 \\
0 & 1 & 2 & 3 \\
0 & 1 & 2 & 3 \\
\end{array}
\]

Less efficient, due to memory access patterns

\[
\begin{array}{cccc}
0 & 1 & 2 & 3 \\
0 & 1 & 2 & 3 \\
0 & 1 & 2 & 3 \\
0 & 1 & 2 & 3 \\
\end{array}
= \begin{array}{cccc}
0 & 1 & 2 & 3 \\
\end{array}
\]

Very inefficient, requires temporary buffers and communication between GPUs

\[
\begin{array}{cccc}
0 & 1 & 2 & 3 \\
0 & 1 & 2 & 3 \\
0 & 1 & 2 & 3 \\
0 & 1 & 2 & 3 \\
\end{array}
= \begin{array}{cccc}
0 & 1 & 2 & 3 \\
\end{array}
\]

\[
\begin{array}{cccc}
0 & 1 & 2 & 3 \\
0 & 1 & 2 & 3 \\
0 & 1 & 2 & 3 \\
0 & 1 & 2 & 3 \\
\end{array}
\]
Matrix Multiplication

- One of the most important algorithms in DL
- Probably the most difficult to optimize
- Most built on cuBLAS routines for the individual blocks
- Typical sizes in deep learning applications have communication between GPUs and servers as the bottleneck
- Multiple implementations are needed for different circumstances
- We have over 15 different distributed GEMM implementations that are used for various types of matrix dimensions and h/w configurations
Matrix Multiplication Scaling

- Smaller blocks doesn’t scale, due to communication overhead and cuBLAS GEMM efficiency
- As matrices get larger, communication becomes less of an issue compared to computation, scaling is less difficult
- Shown below is the relative performance for the same multiplication, distributed over a different number of GPUs within our framework
Matrix Multiplication Performance

- Shown below are runtimes for square matrix multiplications of various sizes
- dMath performance on a variety of GPUs is compared with a cuBLAS 7.5 baseline and cuBLAS-XT

<table>
<thead>
<tr>
<th>Size</th>
<th>1 GPU</th>
<th>2 GPUs</th>
<th>8 GPUs</th>
<th>32 GPUs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>cublas 7.5</td>
<td>dMath</td>
<td>cublasXt 7.5</td>
<td>dMath</td>
</tr>
<tr>
<td>4096</td>
<td>0.052</td>
<td>0.035</td>
<td>0.146</td>
<td><strong>0.021</strong></td>
</tr>
<tr>
<td>6144</td>
<td>0.174</td>
<td>0.109</td>
<td>0.589</td>
<td><strong>0.038</strong></td>
</tr>
<tr>
<td>8192</td>
<td>0.413</td>
<td>0.245</td>
<td>1.209</td>
<td><strong>0.076</strong></td>
</tr>
<tr>
<td>12288</td>
<td>1.450</td>
<td>0.840</td>
<td>3.268</td>
<td>0.433</td>
</tr>
<tr>
<td>16384</td>
<td>3.340</td>
<td>2.034</td>
<td>8.455</td>
<td><strong>0.528</strong></td>
</tr>
<tr>
<td>24576</td>
<td>12.279</td>
<td>6.809</td>
<td>28.062</td>
<td>1.744</td>
</tr>
<tr>
<td>32768</td>
<td>-</td>
<td>-</td>
<td>68.618</td>
<td>4.015</td>
</tr>
<tr>
<td>49152</td>
<td>-</td>
<td>-</td>
<td>187.344</td>
<td>14.016</td>
</tr>
<tr>
<td>65536</td>
<td>-</td>
<td>-</td>
<td>461.233</td>
<td>-</td>
</tr>
</tbody>
</table>

* Tesla K80 result

Samsung Research America – Silicon Valley
Replication Motivation

- Often desirable for workers to have identical copies of a matrix (abundant memory and rarely changing data)
- Useful for parameters in a network
- Equivalent to the distribution of parameters in a parameter server
- Caching (using replication) of a distributed weight matrix, forward pass caches it, backward pass requires no communication
Replication Motivation (AlexNet Example)

- The matrix multiplication in the three fully connected layers shown above requires the weight matrix to be shared between the workers in the forward pass.
- We cache this matrix, allowing the backward data gradient to be computed without any communication between GPUs.
• Each worker stores an entire copy of the matrix, in addition to the blocks they are responsible for.
• When a matrix is changed, the workers automatically redistribute the matrix.
• Replication can be temporarily suspended while multiple updates are made to a matrix, then resumed.
Asynchronous Replication

- Parameters are updated all at once and then replicated in SGD training
- Updated parameters often not needed until much deeper into the network
- Replication frequently followed by forward convolution layers (high computation, zero communication)
- Replace synchronous replication with asynchronous replication, overlapping parameter redistribution with computation
Tips and Tricks

• Any serial portion of the program will hurt scaling (Amdahl’s law)
• Dispatching jobs to the workers has a small overhead (dozens of microseconds)
• Additional time is spent broadcasting the metadata, and setting up data structures required to perform a distributed job
• For large jobs this overhead is not a problem
• Scaling to more GPUs without increasing batch size, strong scaling, presents significant issues (small convolutions in GoogLeNet)
• Can be alleviated by avoiding lazy solutions
  • Before
    history_data->scale(momentum);
    history_data->addMatrix(local_rate, *param_diff);
  • After
    history_data->addMatrix(momentum, *history_data,
                            local_rate, *param_diff);
Tips and Tricks

• Combine common operations together into a single job
  • Backward convolutions involve computing data, filter and bias gradients, can be done all at once
  • Allows for multiple CUDA streams to be used
  • Overlapping the filter and bias gradient reductions with data gradient computation helps optimize performance
• Avoid multiple MPI calls when possible, instead copy the data into a single buffer and do a single call (e.g. filter and bias gradient reductions)
• Implement batched versions of jobs (e.g. matrix addition used in the parameter update)
• For Nvidia Tesla K80, we have found much better performance by locking the clocks at 758Mhz and disabling auto-boost, e.g. ~5-10% for multi-GPU jobs
• Registering memory with the IB driver is costly, try to reuse your buffers to prevent this costly registration, i.e. use a memory manager of some sort
### Real World: soumith/convnet-benchmarks -- 128 Batch, i.e. Strong Scaling

<table>
<thead>
<tr>
<th>Library</th>
<th>Class</th>
<th>Time (ms)</th>
<th>forward (ms)</th>
<th>backward (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CuDNN[R4]-fp16 (Torch)</td>
<td>cudnn.SpatialConvolution</td>
<td>71</td>
<td>25</td>
<td>46</td>
</tr>
<tr>
<td>Nervana-neon-fp16</td>
<td>ConvLayer</td>
<td>78</td>
<td>25</td>
<td>52</td>
</tr>
<tr>
<td>CuDNN[R4]-fp32 (Torch)</td>
<td>cudnn.SpatialConvolution</td>
<td>81</td>
<td>27</td>
<td>53</td>
</tr>
<tr>
<td>TensorFlow</td>
<td>conv2d</td>
<td>81</td>
<td>26</td>
<td>55</td>
</tr>
<tr>
<td>Nervana-neon-fp32</td>
<td>ConvLayer</td>
<td>87</td>
<td>28</td>
<td>58</td>
</tr>
<tr>
<td>fbfft (Torch)</td>
<td>fbnn.SpatialConvolution</td>
<td>104</td>
<td>31</td>
<td>72</td>
</tr>
<tr>
<td>Chainer</td>
<td>Convolution2D</td>
<td>177</td>
<td>40</td>
<td>136</td>
</tr>
</tbody>
</table>

- Expresso 64.5ms average on two titans
- Expresso 48.6ms average on four titans
- Expresso 39.1ms average on eight titans
- Expresso 121.8ms average on one titan
Real World: Training AlexNet -- 256 Batch, i.e. Strong Scaling
Real World: Inception v3, 32 Batch per GPU

FPS

Expresso

TensorFlow

Inception v3  2 GPU  4 GPU  8 GPU  16 GPU  32 GPU  64 GPU

Samsung Research America – Silicon Valley
# Real World: Training Time

**AlexNet 1024:**

- **Caffe** -- 8 K80s -- 17h 20min
- **NV-Caffe 0.14** -- 8 K80s -- 12h 15min
- **Expresso 0.8** -- 8 K80s -- 05h 15min

**GoogLeNet v1 1024 Batch:**

- **Caffe** -- 8 K80s -- 03d 20hrs
- **Expresso** -- 8 K80s -- 02d 12hrs
- **Expresso** -- 32 K80s -- 01d 06hrs

*Scaling batch from single GPU to 64 GPUs results in the same accuracy, e.g. AlexNet 256 58.59+/-0.5% top one.*

Expresso results are for that of our version hybrid parallelism [Krizhevsky14].
GoogLeNet V3 (aka Inception v3):

**Tesla K40 GPUs**
- TensorFlow -- 16, 32, 64 GPUs -- 507h 58min
- TensorFlow Internal -- 100 GPUs -- 65h 00min
- Expresso v0.8 -- 64 GPUs -- 48h 22min

**Tesla m40 GPUs**
- Expresso v0.8 -- 96 GPUs -- 13h 45min

**ImageNet2016 Scene Classification competition**
- Third place in single model category
- 7th place in ensemble category
• Data loading quickly becomes a bottleneck when scaling to multiple GPUs
• Not practical to load from a single process, queues, etc. don’t work well
• Automatically tuned CPU, GPU multi-threaded data loader runs on each worker
• Periodically runs, (0.1%) a check is done to determine if the current policy is adequate... remembering multi-tenancy
• No policy is good for all models as there is a different proportion of work in different models.
Data Loading & Augmentation

- At runtime, the master process samples performance for different combinations of thread counts and transfer indices.
- This allows the system to accelerate data augmentation when it is the bottleneck.

**Memory Management:**
- Page-locked host allocations and device allocations cause implicit synchronization and are expensive.
- Host & device buffers are allocated by each thread on start-up and only resized when a sample exceeds the current buffer size. Memory footprint quickly stabilizes.
Data Loading & Augmentation

AlexNet Perf on 8 M40s

- Single Thread
- Multi-Thread
- Auto-tuned

Frames / Second

Batch Size

64 | 128 | 256 | 512 | 1024 | 2048 | 4096
Motivations
• Lower data storage and communication
• Better performance with newer GPUs (Pascal)
• Proven to work great for inference – Fast, Accurate

Challenges
• Instability for certain layers of training
• Partial support by software stack (yet)

Addressing Accuracy Loss
• Performing certain operations in higher precession
  • Softmax, Inner product, weight updates
H/W: Samsung Advanced Learning

Diagram showing
- InfiniBand EDR Fabric
- EDR IB
- PCIe Switch
- PCIe 3.0 Root Complexes
- Multicore X86-64 Xeon
- Main Memory Banks
- K80 GPU

Samsung Research America – Silicon Valley
Conclusion

- dMath provides effective scaling to DL framework: Caffe, Kaldi
- DNN pipeline with data loader and array of features
- Can also be used in other applications like Medical Imaging, Finite Element Analysis, etc.

Future Work
- Half precision training
- Layers implementation for RNN, etc.
Acknowledgments

....
Contact

hars.vardhan@samsung.com