Demystifying Learning @ Scale: From Distributed Math to Effective HPC Infra

S. Eliuk Ph.D. CS (Project Lead, Chief Architect)
C. Upright M.Sc. CS (Lead Developer)
Computing Science Innovation Center
Samsung Research America
April 7th 2016
Outline

i. Why a distributed Mathematics (dMath) library?
   i. Distributed world means we need unique perspectives on algorithms

ii. Current Open-Source offerings...
   i. Audience perspective
   ii. Our sentiments

iii. {Data, Model, Hybrid} Parallelism
   i. Expresso, Caffe – master powered by dMath
   ii. Distributed Kaldi, Kaldi – master powered by dMath

iv. Best Known Practices
   i. Common issues in the NN pipeline
   ii. Tips & 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 Platform that enables Samsung to efficiently prototype distributed DL algorithms on cutting-edge hardware infrastructure for large-scale, computation-intensive applications and services.
Deep Learning Platform: Framework

### Current Approaches

<table>
<thead>
<tr>
<th>Application (DNN – Kaldi ASR, Caffe AIR, Theano, NLP)</th>
<th>LAPACK</th>
<th>FFT</th>
<th>General Purpose Math</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>cuBLAS</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>CUDA</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Node 1

- GPU #1
- ... GPU #16

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

### Samsung – SRA Approach

<table>
<thead>
<tr>
<th>dMath Framework</th>
<th>Application (DNN – Kaldi ASR, Caffe AIR, Theano, Torch, etc)</th>
<th>LAPACK</th>
<th>FFT</th>
<th>General Purpose Math</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>cuBLAS</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>CUDA</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Node 1

- GPU #1
- ... GPU #16

Node N

- GPU #1
- ... GPU #16

- “Scales Out” Effectively
- Comprehensive Library Developed
- Larger models – frontier research
- Extended feature set

- Abstract away complicated distributed & multi-GPU programming from high-level Machine Learning user,
- Integration w/ existing open-source frameworks.
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 GPU Conference

Jeffrey Dean
Senior Fellow
Google Brain

Samsung Research America – Silicon Valley
Current Open-Source Offerings: Discussion

Thoughts?
Current Open-Source Offerings:

Single-GPU
- BVLC / caffe
- torch

Multi-GPU
- TensorFlow
- CNTK
- theano

Distributed
- dmlc
- mxnet
Best Known Practices: Data Layout

- Matrix data is divided into multiple blocks and distributed across the workers
- Arbitrary layouts are supported
- 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
Best Known Practices: Data Layout and Algorithm Efficiency

Efficient, no communication between GPUs

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

Inefficient, off diagonal blocks require communication between GPUs

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

Efficient, no communication between GPUs

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

Less efficient, due to memory access patterns

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

Very inefficient, requires temporary buffers and communication between GPUs

\[
\begin{array}{cc}
0 & 1 \\
2 & 3 \\
\end{array}
\quad = \quad
\begin{array}{cc}
0 \\
1 \\
2 \\
3 \\
\end{array}
\]
Best Known Practices: 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
- Caching of a distributed weight matrix, forward pass caches it, backward pass requires no communication
- When a matrix is changed, the workers automatically redistribute the matrix
- Equivalent to the distribution of parameters in a parameter server
Best Known Practices: 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.
Best Known Practices: Replication Implementation

- Each worker stores an entire copy of the matrix, in addition to the blocks they are responsible for.
- Replication can be temporarily suspended while multiple updates are made to a matrix, then resumed.
Best Known Practices: 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
Best Known Practices: Matrix Multiplication

- One of the most important algorithms in dMath
- Probably the most difficult to optimize
- 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
  - General purpose
  - Variant of Fox’s algorithm [1987]
  - Fully local (replicated weight matrix, local data)
  - Currently over 15 different distributed algorithms for GEMM.
**Best Known Practices:** Cyclic GEMM Implementation

\[
C = A \times B
\]

- One dimensional decomposition of the matrices across workers
Best Known Practices: Cyclic GEMM Implementation

- One dimensional decomposition of the matrices across workers
Best Known Practices: Cyclic GEMM Implementation

- One dimensional decomposition of the matrices across workers
- Multiplication divided into multiple stages, overlapping computation with cyclic asynchronous transfers
- Darkly colored blocks are currently being used in a computation
- Dark orange indicates initiation of data transfer
Best Known Practices: Cyclic GEMM Implementation

- One dimensional decomposition of the matrices across workers
- Multiplication divided into multiple stages, overlapping computation with cyclic asynchronous transfers
- Darkly colored blocks are currently being used in a computation
- Dark orange indicates initiation of data transfer
- Light orange is a transfer still in progress
Best Known Practices: Cyclic GEMM Implementation

- One dimensional decomposition of the matrices across workers
- Multiplication divided into multiple stages, overlapping computation with cyclic asynchronous transfers
- Darkly colored blocks are currently being used in a computation
- Dark orange indicates initiation of data transfer
- Light orange is a transfer still in progress
- Many transfers and GEMM calls happen concurrently
Best Known Practices: Cyclic GEMM Implementation

- One dimensional decomposition of the matrices across workers
- Multiplication divided into multiple stages, overlapping computation with cyclic asynchronous transfers
- Darkly colored blocks are currently being used in a computation
- Dark orange indicates initiation of data transfer
- Light orange is a transfer still in progress
- Many transfers and GEMM calls happen concurrently
- The process repeats again in the next outer stage
**Best Known Practices: Inner Product Layer**

- In a neural network, the cyclic GEMM routine transfers the weight matrix through the workers.
- We take advantage of this, by caching it on the forward pass.
- On the backward pass, all workers have a copy of the weight matrix, and can calculate the input data gradients without costly communication.
Best Known Practices: Matrix Multiplication Scaling

- Can only distribute so far, due to communication overhead and cuBLAS GEMM efficiency with smaller blocks
- As matrices get larger communication becomes less of an issue compared to computation, scaling is less difficult
- In practice we copy our data to a subset of workers, and perform our GEMM routines on this subset
- Shown below is the time in milliseconds for the same multiplication, distributed over a different number of GPUs
Best Known Practices: 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 cublas 7.5</th>
<th>1 GPU dMath</th>
<th>2 GPUs cublasXt 7.5</th>
<th>2 GPUs dMath</th>
<th>8 GPUs cublasXt 7.5</th>
<th>8 GPUs dMath</th>
<th>32 GPUs dMath</th>
</tr>
</thead>
<tbody>
<tr>
<td>4096</td>
<td>0.052</td>
<td>0.035</td>
<td>0.146</td>
<td>0.021</td>
<td>0.169</td>
<td>0.101</td>
<td></td>
</tr>
<tr>
<td>6144</td>
<td>0.174</td>
<td>0.109</td>
<td>0.589</td>
<td>0.038</td>
<td>0.259</td>
<td>0.104</td>
<td></td>
</tr>
<tr>
<td>8192</td>
<td>0.413</td>
<td>0.245</td>
<td>1.209</td>
<td>0.076</td>
<td>0.493</td>
<td>0.121</td>
<td></td>
</tr>
<tr>
<td>12288</td>
<td>1.450</td>
<td>0.840</td>
<td>3.268</td>
<td>0.433</td>
<td>1.420</td>
<td>0.258</td>
<td></td>
</tr>
<tr>
<td>16384</td>
<td>3.340</td>
<td>2.034</td>
<td>8.455</td>
<td>0.528</td>
<td>3.223</td>
<td>0.726</td>
<td></td>
</tr>
<tr>
<td>24576</td>
<td>12.279</td>
<td>6.809</td>
<td>28.062</td>
<td>1.744</td>
<td>10.295</td>
<td>1.091</td>
<td></td>
</tr>
<tr>
<td>32768</td>
<td>-</td>
<td>-</td>
<td>68.618</td>
<td>4.015</td>
<td>24.657</td>
<td>2.389</td>
<td></td>
</tr>
<tr>
<td>49152</td>
<td>-</td>
<td>-</td>
<td>187.344</td>
<td>14.016</td>
<td>82.161</td>
<td>5.260</td>
<td></td>
</tr>
<tr>
<td>65536</td>
<td>-</td>
<td>-</td>
<td>461.233</td>
<td>-</td>
<td>192.380</td>
<td>10.592</td>
<td></td>
</tr>
</tbody>
</table>
Best Known Practices: Reductions

- Important to be cognizant of what MPI is doing under the hood
- CUDA aware MPI_Reduce was copying to the CPU to perform the reduction
- Instead implemented a custom reduction that was aware of the hardware
  - Transfer within a root complex is fast
  - QPI between root complexes is slow
  - A reduction is done on each root complex individually, then between root complexes using CUDA aware MPI_Send / MPI_Recv.
  - The MPI_Send / MPI_Recv pair is further accelerated via a custom branch of MPI that allows intranode GDR, for extremely low latency and high bandwidth via Mellanox EDR (100Gb/s)... more to come shortly on that one.
Best Known Practices: Data Loading

- Data loading quickly becomes a bottleneck when scaling to multiple GPUs
- Not practical to load from a single process
- Threaded image loader runs on each worker, supporting both raw and encoded images
- Extra cores required to handle the processing, and fast data access to feed it
- Caching in RAM is the best option for the database
- Randomization of batches have next to zero cost when cached in RAM
- Caching of pages in Linux happens on a per socket basis (NUMA), significant performance degradation can occur if QPI gets involved, i.e. when page migration between NUMA-nodes takes place
- Each worker loads from a subset of the entire database (QPI aware caching)
- Optional direct caching in memory of the DB on each worker
**Best Known Practices:** Logging, Debugging & Profiling

Most important, left for last, Logging, Debugging and Profiling:
- Flexible per-process logging is indispensable (master and workers)
- Inevitably, difficult bugs will arise, sometimes due to rare and subtle timing variations between processes
- Flexible and comprehensive logging records the sequence of events
- Scripting then analyzes those logs
- Logging effects timing in the application, and will easily hide bugs
- A separate logging thread can be used to minimize this effect
  - Main places data or a function object in a queue
  - Logging thread processes this and logs it
Best Known Practices: Logging, Debugging & Profiling

- Profiling is trivial to implement in this framework, basic timing code and logging combined with scripting helps to locate bottlenecks.
- Performance can be measured on a per layer or per layer type basis, using a variety of metrics:
  - Average time
  - Speedup ratio
  - Percentage efficiency
  - Scaling residual (time that could be saved with perfect linear scaling)
Best Known Practices: Logging, Debugging & Profiling

Sample profiling CSV for AlexNet:

<table>
<thead>
<tr>
<th>Efficiency</th>
<th>num_worker</th>
<th>iteration</th>
<th>apply_update</th>
<th>forward</th>
<th>backward</th>
<th>forward_Convolution</th>
<th>forward_Data</th>
<th>forward_InnerProduct</th>
<th>backward_Convolution</th>
<th>backward_InnerProduct</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>135.897</td>
<td>125.391</td>
<td>102.887</td>
<td>155.467</td>
<td>153.576</td>
<td>44.859</td>
<td>31.133</td>
<td>266.911</td>
<td>20.848</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>56.06</td>
<td>27.016</td>
<td>43.48</td>
<td>65.902</td>
<td>85.914</td>
<td>12.775</td>
<td>9.126</td>
<td>150.747</td>
<td>5.433</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Speedup</th>
<th>num_worker</th>
<th>iteration</th>
<th>apply_update</th>
<th>forward</th>
<th>backward</th>
<th>forward_Convolution</th>
<th>forward_Data</th>
<th>forward_InnerProduct</th>
<th>backward_Convolution</th>
<th>backward_InnerProduct</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>14.503</td>
<td>7.972</td>
<td>10.808</td>
<td>17.235</td>
<td>18.879</td>
<td>3.482</td>
<td>2.563</td>
<td>33.56</td>
<td>1.747</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Scaling Residual</th>
<th>num_worker</th>
<th>iteration</th>
<th>apply_update</th>
<th>forward</th>
<th>backward</th>
<th>forward_Convolution</th>
<th>forward_Data</th>
<th>forward_InnerProduct</th>
<th>backward_Convolution</th>
<th>backward_InnerProduct</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>-122.523</td>
<td>-1.979</td>
<td>-3.536</td>
<td>-116.758</td>
<td>-34.823</td>
<td>1.15</td>
<td>20.969</td>
<td>-175.498</td>
<td>51.122</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>90.891</td>
<td>6.601</td>
<td>40.962</td>
<td>42.33</td>
<td>4.092</td>
<td>1.597</td>
<td>23.598</td>
<td>-23.619</td>
<td>58.591</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Average Time</th>
<th>num_worker</th>
<th>iteration</th>
<th>apply_update</th>
<th>forward</th>
<th>backward</th>
<th>forward_Convolution</th>
<th>forward_Data</th>
<th>forward_InnerProduct</th>
<th>backward_Convolution</th>
<th>backward_InnerProduct</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>3710.74</td>
<td>78.1952</td>
<td>1008.372</td>
<td>2618.044</td>
<td>798.57</td>
<td>7.48312</td>
<td>75.83648</td>
<td>2245.134</td>
<td>107.7208</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>341.319</td>
<td>7.79513</td>
<td>122.51</td>
<td>210.498</td>
<td>64.99805</td>
<td>2.08519</td>
<td>30.44827</td>
<td>105.1442</td>
<td>64.58733</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>255.858</td>
<td>9.80906</td>
<td>93.2953</td>
<td>151.907</td>
<td>42.29886</td>
<td>2.14932</td>
<td>29.59355</td>
<td>66.8088</td>
<td>61.67784</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>206.852</td>
<td>9.04509</td>
<td>72.4734</td>
<td>124.144</td>
<td>29.04686</td>
<td>1.83056</td>
<td>25.96815</td>
<td>46.54185</td>
<td>61.95691</td>
</tr>
</tbody>
</table>
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)
Tips and Tricks

• Caching of job data structures and metadata can significantly reduce parallelization overhead and incredibly useful for fixed pipelines
  • No metadata will be sent (other than the cached job id)
  • Only the computation is performed, no initialization or cleanup
  • Eliminates as much of the parallelization overhead as possible
• 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, with various sized allocation that are then reused
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>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>TensorFlow</td>
<td>conv2d</td>
<td>151</td>
<td>34</td>
<td>117</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

![Graph showing performance of different frameworks with varying GPU counts.]
### Real World: Training Time

#### AlexNet 1024:
- **Caffe** -- 8 K80s -- 17h 20min
- **NV-Caffe 0.14** -- 8 K80s -- 12h 15min
- **Expresso** -- 8 K80s -- 07h 40min
- **Expresso** -- 32 K80s -- 05h 30min

#### 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].
**H/W:** Samsung Advanced Learning v1.0

**TABLE I.** COMMUNICATION MEDIUMS: LATENCY

<table>
<thead>
<tr>
<th>Transfer Size Bytes</th>
<th>Shared Memory Host</th>
<th>Shared Memory GPU</th>
<th>IntraNode IB - EDR GDR</th>
<th>InterNode IB - EDR GDR</th>
<th>CUDA P2P</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.86</td>
<td>0.87</td>
<td>0.91</td>
<td>1.33</td>
<td>0.93</td>
</tr>
<tr>
<td>1</td>
<td>1.11</td>
<td>31.70</td>
<td>6.13</td>
<td>5.98</td>
<td>19.41</td>
</tr>
<tr>
<td>128</td>
<td>1.28</td>
<td>26.25</td>
<td>5.83</td>
<td>5.77</td>
<td>15.51</td>
</tr>
<tr>
<td>512</td>
<td>1.54</td>
<td>26.20</td>
<td>12.00</td>
<td>11.58</td>
<td>15.33</td>
</tr>
<tr>
<td>16384</td>
<td>6.95</td>
<td>30.97</td>
<td>16.95</td>
<td>16.74</td>
<td>17.50</td>
</tr>
<tr>
<td>524288</td>
<td>138.61</td>
<td>163.39</td>
<td>218.72</td>
<td>157.12</td>
<td>80.91</td>
</tr>
<tr>
<td>2097152</td>
<td>501.10</td>
<td>515.71</td>
<td>458.22</td>
<td>425.37</td>
<td>279.04</td>
</tr>
<tr>
<td>4194304</td>
<td>971.19</td>
<td>936.43</td>
<td>765.36</td>
<td>741.60</td>
<td>541.65</td>
</tr>
</tbody>
</table>

**TABLE II.** COMMUNICATION MEDIUMS: BANDWIDTH

<table>
<thead>
<tr>
<th>Transfer Size Bytes</th>
<th>Shared Memory Host</th>
<th>Shared Memory GPU</th>
<th>IntraNode IB - EDR GDR</th>
<th>InterNode IB - EDR GDR</th>
<th>CUDA P2P</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.76</td>
<td>0.06</td>
<td>0.58</td>
<td>0.68</td>
<td>0.13</td>
</tr>
<tr>
<td>128</td>
<td>213.95</td>
<td>9.41</td>
<td>69.99</td>
<td>87.41</td>
<td>16.41</td>
</tr>
<tr>
<td>512</td>
<td>679.82</td>
<td>37.50</td>
<td>226.62</td>
<td>268.72</td>
<td>67.28</td>
</tr>
<tr>
<td>16384</td>
<td>5269.01</td>
<td>107.76</td>
<td>3558.15</td>
<td>3922.10</td>
<td>2336.16</td>
</tr>
<tr>
<td>524288</td>
<td>4540.58</td>
<td>4081.20</td>
<td>5298.05</td>
<td>6110.80</td>
<td>8984.97</td>
</tr>
<tr>
<td>2097152</td>
<td>4901.57</td>
<td>5148.11</td>
<td>7543.43</td>
<td>8105.62</td>
<td>9604.57</td>
</tr>
<tr>
<td>4194304</td>
<td>5064.01</td>
<td>5266.48</td>
<td>7758.30</td>
<td>8657.90</td>
<td>9720.82</td>
</tr>
</tbody>
</table>

Via: custom branch of OpenMPI & Mvapich2 - PCC
H/W: Samsung Advanced Learning v1.0 & v2.0

Two variants, dual socket, one Mellanox InfiniBand EDR (100Gb/s) per socket connected directly to a PCle v3.0 96 lane switch, w/
1. 8 Nvidia Tesla k80s per node, 8 nodes, 128 GPUs, per rack,
2. 8 Nvidia Tesla m40 24GB per node, 12 nodes, 96 GPUs, per rack.

Remember to be sure to have an adequate number of cores per socket to sufficiently power the GPUs, min of 1.25 physical cores per GPU.
Recap:

- Current open-source offering are incredible but have many shortcomings
- Data layout is very important, really need to consider the layout, often optimal layout may be to distribute to only a subset of workers
- Cache data to prevent unnecessary communication, huge benefit on backward pass
- Async replication, overlap computation & communication,
- GEMM, need an arsenal of algorithms w/ consideration of underlying h/w architecture
Recap:

• Work for strong scaling & secondly weak scaling
• Likely need to augment chosen MPI version w/ custom routines for reduction, broadcasts, intranode GDR, etc
• Distributed, multi-threaded data loading is important but do not forget NUMA nodes
• Most importantly, have solid logging, profiling, test suites from the beginning
Publications:

- dMath architecture paper available on arxiv
- A few under submission so expect a bunch from us in Q2-Q4
Employment Opportunities:

s.eliuk@samsung.com
Acknowledgements & Additional Information

http://arxiv.org/abs/1604.01416
http://arxiv.org/abs/1604.01416