CSS600 – Spring 2017 – Final Report
Nathaniel J. Grabaskas
Computing and Software Systems Department, University of Washington

Contents
1. Introduction ............................................................................................................................................ 3
2. Power Efficiency ................................................................................................................................... 3
   A. SXAPY ............................................................................................................................................... 3
      i. Introduction ................................................................................................................................... 3
      ii. Method ......................................................................................................................................... 3
      iii. Conclusion ................................................................................................................................. 3
   B. Results ............................................................................................................................................. 4
   C. Power Modes ................................................................................................................................... 5
3. Jetson TX1 Cluster Setup ....................................................................................................................... 5
   A. Installation ....................................................................................................................................... 5
   B. Benchmarks ..................................................................................................................................... 6
4. Game AI Analysis ................................................................................................................................ 6
   A. Monte-Carlo-Tree-Search ................................................................................................................ 7
   B. General-Video-Game AI .................................................................................................................. 7
      i. UCT and MCTS for 2-player deterministic games ..................................................................... 7
      ii. FUEGO ......................................................................................................................................... 8
      iii. General Video Game AI ........................................................................................................... 8
      iv. DeepMind Deep Q-learning Network ....................................................................................... 8
   C. DeepMind ....................................................................................................................................... 8
      i. DQN – Structure ............................................................................................................................ 8
5. Torch 7.0.............................................................................................................................................. 9
   A. Torch – Structure ............................................................................................................................. 9
   B. TorchMPI – Structure ..................................................................................................................... 10
   C. Cross Entropy ............................................................................................................................... 10
   D. Neural Network Parallelization Methods ...................................................................................... 10
   E. TorchMPI Parallelization Methods ............................................................................................. 11
   F. Benchmarks ..................................................................................................................................... 11
1. CSS 595 – Proposal Summary ........................................................................................................... 12
   A. Working Title ............................................................................................................................... 12
1. Introduction

Choosing a research path for my Capstone project was a daunting task. But I knew it was crucial to pick a topic and path that not only held the promise of opportunities to improve but also held my interest. These two criteria go hand in hand, you cannot have one without the other. If you do then either I would have a project that I was unmotivated about or a project that didn’t have anywhere to go.

This paper reports that history of my journey this quarter and how I went from optimizing Linear Algebra (LA) algorithms for power efficiency to working on the framework to parallelize neural network training on embedded devices. I show my initial results from power efficiency analysis in Section 2. In Section 3, I detail how I setup my cluster and initial benchmarks. Next, covers the Game AIs that I research and how that led me to Torch. In Section 5, I give a more detailed look at Torch/TorchMPI. Finally, I conclude with a summary of my proposal and conclusions from this quarter.

2. Power Efficiency

A. SAXPY

i. Introduction

Using the new Jetson TX2 developer kit we do an in-depth analysis of the power efficiency of using both float and float4 data types for vector addition. The TX2 boasts up to 1 TFLOPs of performance using only 7.5 Watts. There are 12 embedded sensors that can be used to monitor the voltage, current, and power for each individual component on the board. The System on a Chip (SOC) offers advantages by having both the CPU and GPU share the same system memory, cutting out the overhead of transferring data to and from the GPU. The SOC has a 128-bit bus that is used to transfer memory from the LPDDR to the CPU and GPU. This was why we chose to look at the difference using float versus float4, as float4 data type is 128-bit which allows for maximum utilization of the system bus.

ii. Method

Originally, we wrote the code to launch many kernels from the CPU thread. We had 10 vectors that would be cycled through thousands of time and record the power usage. During the analysis, we found that using this method meant the GPU was only running about 5% of the time and thus the OS overhead was affecting our results. We then decided to move the loop to the GPU and only make 1 kernel call. This allowed for the GPU to be used nearly the entire time.

To measure the power consumption of the board, we used a script that launched the code with timestamps down to the nanosecond for beginning and end. We then logged the power consumption from 5 of the available sensors: SYS_GPU, SYS_SOC, SYS_CPU, SYS_IN, and SYS_DDR. Sensor readings were taken an average of 55-60 times per second. Using the start and stop time we averaged those power readings for each sensor and multiplied that by the execution time to get the total watt seconds.

iii. Conclusion

Our experiments allowed us to clearly show how using a data type to maximize system bus utilization can have a profound impact not only on power consumption but also on execution time.
B. Results

Figure 1. Shows the average power per second consumption for the four rails was consistently higher using the float4 type. SYS_CPU was not shown as they were nearly the same.

Table 1. Shows the average watt second consumption per power rail and the increase when compared to float type.

<table>
<thead>
<tr>
<th>Rail Names</th>
<th>Float</th>
<th>Float4</th>
<th>Increase</th>
</tr>
</thead>
<tbody>
<tr>
<td>VDD_SYS_GPU</td>
<td>1.58</td>
<td>1.83</td>
<td>15.82%</td>
</tr>
<tr>
<td>VDD_SYS_SOC</td>
<td>1.05</td>
<td>1.22</td>
<td>16.19%</td>
</tr>
<tr>
<td>VDD_SYS_IN</td>
<td>8.22</td>
<td>8.86</td>
<td>7.79%</td>
</tr>
<tr>
<td>VDD_SYS_CPU</td>
<td>1.22</td>
<td>1.21</td>
<td>-0.82%</td>
</tr>
<tr>
<td>VDD_SYS_DDR</td>
<td>2.87</td>
<td>3.31</td>
<td>15.33%</td>
</tr>
</tbody>
</table>

Table 2. Shows that overall the DDR and GPU consumed less power when compared to float data type and the execution time was also decreased.

<table>
<thead>
<tr>
<th>Total Watt/Sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Watt Seconds Minus CPU/IN</td>
</tr>
<tr>
<td>269.8</td>
</tr>
<tr>
<td>248.5</td>
</tr>
<tr>
<td>-7.89%</td>
</tr>
<tr>
<td>DDR/GPU Watt/sec</td>
</tr>
<tr>
<td>239.9</td>
</tr>
<tr>
<td>200.82</td>
</tr>
<tr>
<td>-16.29%</td>
</tr>
<tr>
<td>Total</td>
</tr>
<tr>
<td>Time (Seconds)</td>
</tr>
<tr>
<td>53.89</td>
</tr>
<tr>
<td>39.11</td>
</tr>
<tr>
<td>-27.43%</td>
</tr>
</tbody>
</table>
C. Power Modes

The Jetson TX2 device can run in various power modes that configure how many CPU cores are running the max frequencies for both CPU and GPU. From these experiments, I concluded that just increasing the frequency does not always lead to better performance or efficiency.

Table 3. Shows power draw recording using FLUKE 8846A meter and CUDA example applications. See [TX2 Power Modes](#) for description of modes.

<table>
<thead>
<tr>
<th>Application</th>
<th>Power Mode</th>
<th>Performance</th>
<th>Power Draw</th>
</tr>
</thead>
<tbody>
<tr>
<td>cdLU Decomposition</td>
<td>0</td>
<td>0.9 GFLOPs</td>
<td>0.3 Amps</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1 GFLOPs</td>
<td>0.13 Amps</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1.2 GFLOPs</td>
<td>0.15 Amps</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>1.2 GFLOPs</td>
<td>0.19 Amps</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1.2 GFLOPs</td>
<td>0.22 Amps</td>
</tr>
<tr>
<td>BlackScholes</td>
<td>0</td>
<td>2.9 GFLOPs</td>
<td>0.78 Amps</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>2.3 GFLOPs</td>
<td>0.54 Amps</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2.6 GFLOPs</td>
<td>0.68 Amps</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>2.6 GFLOPs</td>
<td>0.69 Amps</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>2.6 GFLOPs</td>
<td>0.70 Amps</td>
</tr>
</tbody>
</table>

3. Jetson TX1 Cluster Setup

A. Installation

i. Run NVIDIA JetPack 3.0 on host machine.
   a. Flash Ubuntu 16.04 LTS to TX1
   b. Install CUDA 8.0 on TX1
   c. Install OpenCV 2.4.13 on TX1

ii. Build and Install OpenMPI 2.0.1.
   a. Download OpenMPI source package.
   b. Unzip (gunzip -c openmpi-2.1.1.tar.gz | tar xf -) and move to directory /usr/local/src/
   d. Following configure completion execute: `sudo make all install`
   e. Copy openmpi.conf to /etc/ld.so.conf.d/
   f. Refresh shared library location: `sudo ldconfig`

iii. Install Torch 7.0
   b. Run: `sudo apt-get install libopenblas-dev`
   c. Run: `cd ~/torch`
   d. Run: `vim install-deps`
   e. Comment out line #194 of 'install-deps' which reads "install_openblas || true"
   f. Run: `bash install-deps`
   g. Run: `./install.sh`

iv. Install torch: `luarocks install torch`

v. Install cutorch: `luarocks install cutorch`

vi. Install cunn: `luarocks install cunn`

vii. Install mnist: `luarocks install mnist`
viii. Install torchnet:  
`luarocks install torchnet`

ix. Install numactl:  
`sudo apt-get install numactl`

x. Build and Install TorchMPI

a. Download TorchMPI source to home directory.
   https://github.com/facebookresearch/TorchMPI

b. From the TorchMPI folder execute: 
   ```
   MPI_C_COMPILER=/opt/openmpi-2.0.1/bin/mpicc
   MPI_CXX_COMPILER=/opt/openmpi-2.0.1/bin/mpicxx
   MPI_CXX_COMPILE_FLAGS="-O3"
   MPIEXEC=/opt/openmpi-2.0.1/bin/mpiexec
   /home/ubuntu/torch/install/bin/luarocks
   make /home/ubuntu/TorchMPI-master/rocks/torchmpi-scm-1.rockspec
   ```

c. Execute TorchMPI examples using the following command: 
   ```
   /opt/openmpi-2.0.1/bin/mpirun -n 4 -npernode 1 --hostfile /home/ubuntu/TorchMPI-master/hostfile --bind-to none ./scripts/wrap.sh /home/ubuntu/torch/install/bin/luajit ./examples/mnist/*filename
   ```

B. Benchmarks

Initial analysis of the cluster was conducted using an MPI communication benchmark test and Wave2D simulation.

Table 4. MPI Communication Benchmark Results using (4) TX1s and (4) UWB-320 machines.

<table>
<thead>
<tr>
<th>Method</th>
<th>Env</th>
<th>Size</th>
<th>Time</th>
<th>Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Send + MPI_Recv</td>
<td>TX1s</td>
<td>1 byte</td>
<td>0.71 ms</td>
<td>-</td>
</tr>
<tr>
<td>MPI_Send + MPI_Recv</td>
<td>TX1s</td>
<td>100 MB</td>
<td>-</td>
<td>68 MB/s</td>
</tr>
<tr>
<td>MPI_Send + MPI_Recv</td>
<td>UWB</td>
<td>1 byte</td>
<td>0.13 ms</td>
<td>-</td>
</tr>
<tr>
<td>MPI_Send + MPI_Recv</td>
<td>UWB</td>
<td>100 MB</td>
<td>-</td>
<td>116 MB/s</td>
</tr>
<tr>
<td>MPI_Broadcast + MPI_Reduce</td>
<td>TX1s</td>
<td>1 byte</td>
<td>20.42 ms</td>
<td>-</td>
</tr>
<tr>
<td>MPI_Broadcast + MPI_Reduce</td>
<td>TX1s</td>
<td>100 MB</td>
<td>-</td>
<td>99 MB/s</td>
</tr>
<tr>
<td>MPI_Broadcast + MPI_Reduce</td>
<td>UWB</td>
<td>1 byte</td>
<td>2.86 ms</td>
<td>-</td>
</tr>
<tr>
<td>MPI_Broadcast + MPI_Reduce</td>
<td>UWB</td>
<td>100 MB</td>
<td>-</td>
<td>174 MB/s</td>
</tr>
</tbody>
</table>

Table 5. Wave2D Simulation Results using (4) TX1s and (4) UWB-320 machines. Time is the average from 10 test runs.

<table>
<thead>
<tr>
<th>Method</th>
<th>Env</th>
<th>Time</th>
<th>Efficiency</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Node</td>
<td>TX1s</td>
<td>7,380,509</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2 Nodes</td>
<td>TX1s</td>
<td>4,199,555</td>
<td>87.87%</td>
<td>1.76</td>
</tr>
<tr>
<td>3 Nodes</td>
<td>TX1s</td>
<td>3,013,061</td>
<td>81.65%</td>
<td>2.45</td>
</tr>
<tr>
<td>4 Nodes</td>
<td>TX1s</td>
<td>2,402,566</td>
<td>76.80%</td>
<td>3.07</td>
</tr>
<tr>
<td>1 Node</td>
<td>UWB</td>
<td>1,929,872</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>4 Nodes</td>
<td>UWB</td>
<td>632,621</td>
<td>76.26%</td>
<td>3.05</td>
</tr>
</tbody>
</table>

4. Game AI Analysis

Following my power efficiency analysis, I decided to take my research into a different direction. My new goal was to find an area that I could improve that would be more easily relatable to a future employee than fine grain power consumption or recursively blocked Cholesky decomposition. I began looking at real-world applications for Cholesky decomposition. From there I discovered the Monte Carlo method and how it was applied to Game AI by using the Monte-Carlo-Tree-Search.
A. Monte-Carlo-Tree-Search

The Monte-Carlo-Tree-Search (MCTS) consists of four steps. 
https://en.wikipedia.org/wiki/Monte_Carlo_tree_search
https://jeffbradberry.com/posts/2015/09/intro-to-monte-carlo-tree-search/

- **Selection**: start from root node R and select successive child nodes down to a leaf node L.
- **Expansion**: unless L ends the game with a win/loss for either player, create one or more child nodes and choose C from them.
- **Simulation**: play a random playout from node C.
- **Backpropagation**: use the result of the simulation to update information in the nodes on the path from C to R.

![Figure 2. Shows a tree diagram for each of the MCTS steps: selection, expansion, simulation, and backpropagation.](https://example.com/figure2.png)

MCTS can be parallelized in three different methods. First, is leaf parallelism where many child or leaf threads are launched from a single node in parallel. Second, is root parallelism where multiple trees are launched in parallel with one thread each. And finally, block parallelism is a mix between the first two where several trees are launched with multiple threads each. The third method is commonly used for GPU implementation as it fits well with the block and thread parallelization methodology.

B. General-Video-Game AI

i. **UCT and MCTS for 2-player deterministic games**
Summary: Both projects use MCTS to play 2-player games (connect four, Othello, nim, etc.). Majority of time (50-80%) is spent on MCTS.

Pro: MCTS should be easily parallel on CPUs. Easily to spread across nodes. Many games can played.
Cons: GPU parallelization can be difficult as each game will require different moves. What can I add to research already conducted?

ii. FUEGO

Summary: Boost library (multi-threading) had issues being installed. Fuego is a more advanced MCTS library used for the game of Go.
Pro: Go AI is widely recognized and the library is more advanced.
Cons: GPU parallelization, boost library support, already supports multi-threading.

iii. General Video Game AI

Summary: General Video Game AI uses a Java framework and allows users to create their own AI to be played against various games.
Pro: Annual GVG-AI competition.
Cons: Java does not support CUDA, most user AIs are not Open Source.

iv. DeepMind Deep Q-learning Network

Summary: Deep Q-learning Network is complex and highly recognized AI. Parallelization work would be in the underlying Torch/Lua CNN framework.
Pro: Cutting edge, well established support, good documentation.
Cons: Complexity.

C. DeepMind

i. DQN – Structure

Figure 4. DQN dependent libraries down to a theoretical MPI parallelized level. Theoretical dependency are in yellow.
5. Torch 7.0

After analyzing the structure of DeepMind’s DQN and considering how this network would be distributed over multiple nodes for training, I saw that because it was built on top of the Neural Network framework of Torch that what I really needed to do was add parallel support to Torch. This would not only enable parallelization of DQN but also allow any similar type network to benefit from the parallelization.

As described from torch.ch “Currently, Torch is a scientific computing framework with wide support for machine learning algorithms that puts GPUs first. It is easy to use and efficient, thanks to an easy and fast scripting language, LuaJIT, and an underlying C/CUDA implementation (torch.ch). TorchMPI, which is developed by the Facebook AI Research Group, already provides some parallel training capabilities.”

The question that needs to be answered is why Torch over TensorFlow or Theano (Other popular machine learning tools)? Even though DeepMind (Google) is shifting support to TensorFlow from Torch, I still believe Torch has several advantages that make it the right choice for my research.


- Still has support from large companies support from Facebook, Twitter, and NVIDIA.
- Smaller footprint (python is 824 Kb vs Lua is less than 100 Kb) without libraries. (http://lua-users.org/wiki/LuaVersusPython)
- Lua uses less memory.
- Few external modules make Lua easier to bundle for a specialized purpose.
- Lua is already popular for embedded systems.

A. Torch – Structure

Figure 5. This figure extrapolated from Torch documentation shows Torch 7.0 class architecture without any addition libraries.
B. TorchMPI – Structure

Figure 6. This figure extrapolated from TorchMPI documentation shows TorchMPI class architecture.

C. Cross Entropy

In information theory, the cross entropy between two probability distributions $p$ and $q$ over the same underlying set of events measures the average number of bits needed to identify an event drawn from the set, if a coding scheme is used that is optimized for an "unnatural" probability distribution $q$, rather than the "true" distribution $p$.

Cross entropy can be used to define the loss function in machine learning and optimization. The true probability $p_i$ is the true label, and the given distribution $q_i$ is the predicted value of the current model.

Cross entropy was used to determine the error and loss for my neural networks because it considers how close the network was on each prediction. This gives a more accurate picture of the network quality.

D. Neural Network Parallelization Methods

Node Parallel – is typically used when the neural network is complex and has many neurons. The network will be broken down and each node will have a subset of neurons to be trained and all will use the same dataset.
Data Parallel – is used when the neural network is less complex but the size of the dataset and the cost of I/O operations is high. Each node is given the complete neural network and trains on only a subset of the data.

E. TorchMPI Parallelization Methods

Parameter Server EASGD - An instance of torchmpi.Update initializes a parameterserver handle for each parameter in the network. The shardingCommunicator is the communicator on whose ranks each parameter tensor is sharded. Collective operation which shards a tensor t on all ranks in the current communicator. On the first such call, a threadpool is initialized to handle client requests in the background.

Parameter Server EASGD Data Parallel – The dataparallelCommunicator is the unit of worker granularity with dataparallel synchronous SGD. This is used instead of shardingCommunicator.

Model Parallel – AllReduce is used to updateGradInput across the nodes. The dataset is split evenly amongst the nodes. (must be even number of nodes)

AllReduce – AllReduce call is done synchronously after gradient accumulations.

Async AllReduce – AllReduce is asynchronously interleaved with gradient accumulations with a final synchronization

F. Benchmarks

Table 6. Shows the comparison of CPU to GPU runtimes using Torch and cunn.

<table>
<thead>
<tr>
<th>Method</th>
<th>Throughput</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear-CPU</td>
<td>36.32 GFLOPs</td>
<td>-</td>
</tr>
<tr>
<td>Linear-GPU</td>
<td>331.25 GFLOPs</td>
<td>9.12x</td>
</tr>
<tr>
<td>Convolution-CPU</td>
<td>1.11 GFLOPs</td>
<td>-</td>
</tr>
<tr>
<td>Convolution-GPU</td>
<td>101.24 GFLOPs</td>
<td>91.21x</td>
</tr>
</tbody>
</table>

Table 7. Shows initial 5 method benchmark results.

<p>| TorchMPI Performance Analysis: 784,10 Linear NN BS = 672 with Jetson TX1 SoCs (--usegpu) |
|-----------------------------------------------|--------|--------|--------|--------|--------|--------|--------|
| Method                                      | Nodes  | Time   | Cycles | Loss   | Error  | Speedup| Efficiency |
|                                             | / Proc | (sec)  |        |        |        |        |           |
| easgd                                       | 1 / 1  | 46.48  | 450    | 6,325.18 | 10.62  | 1.00   | 100.00%   |
| easgd                                       | 2 / 1  | 32.41  | 450    | 6,975.54 | 11.81  | 1.43   | 71.71%    |
| easgd                                       | 2 / 2  | 26.49  | 450    | 8,499.38 | 15.49  | 1.75   | 43.87%    |
| easgd                                       | 3 / 1  | 26.84  | 450    | 10,801.94| 17.99  | 1.73   | 57.72%    |
| easgd                                       | 4 / 1  | 23.71  | 450    | 8,499.38 | 15.49  | 1.96   | 49.01%    |
| easgd data parallel                         | 1 / 1  | 56.64  | 450    | 6,771.85 | 9.42   | 1.00   | 100.00%   |
| easgd data parallel                         | 2 / 1  | 40.92  | 450    | 10,153.63| 10.47  | 1.38   | 69.21%    |
| easgd data parallel                         | 3 / 1  | 35.32  | 450    | 20,036.15| 13.24  | 1.60   | 53.45%    |
| easgd data parallel                         | 4 / 1  | 31.25  | 450    | 18,659.83| 10.92  | 1.81   | 45.31%    |
| model parallel                              | 1 / 1  | 55.02  | 450    | 1,083.10 | 8.4    | 1.00   | 100.00%   |
| model parallel                              | 2 / 1  | 62.44  | 450    | 1,217.71 | 9.23   | 0.88   | 44.06%    |
| model parallel                              | 4 / 1  | 69.19  | 450    | 1,205.48 | 9.32   | 0.80   | 19.88%    |</p>
<table>
<thead>
<tr>
<th>Method</th>
<th># Nodes</th>
<th>BatchSize</th>
<th>LoC/FC/O</th>
<th>Complex</th>
<th>Time</th>
<th>Error</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>allReduce</td>
<td>1 / 1</td>
<td>51.46</td>
<td>450</td>
<td>1,083.09</td>
<td>8.4</td>
<td>1.00</td>
<td>100.00%</td>
</tr>
<tr>
<td>allReduce</td>
<td>2 / 1</td>
<td>41.11</td>
<td>450</td>
<td>2,884.06</td>
<td>10.45</td>
<td>1.25</td>
<td>62.59%</td>
</tr>
<tr>
<td>allReduce</td>
<td>3 / 1</td>
<td>35.16</td>
<td>450</td>
<td>5,459.84</td>
<td>14.69</td>
<td>1.46</td>
<td>48.79%</td>
</tr>
<tr>
<td>allReduce</td>
<td>4 / 1</td>
<td>30.86</td>
<td>450</td>
<td>10,255.85</td>
<td>18.74</td>
<td>1.67</td>
<td>41.69%</td>
</tr>
<tr>
<td>async Allreduce</td>
<td>1 / 1</td>
<td>56.82</td>
<td>450</td>
<td>1,083.09</td>
<td>8.4</td>
<td>1.00</td>
<td>100.00%</td>
</tr>
<tr>
<td>async Allreduce</td>
<td>2 / 1</td>
<td>42.55</td>
<td>450</td>
<td>2,884.06</td>
<td>10.45</td>
<td>1.34</td>
<td>66.77%</td>
</tr>
<tr>
<td>async Allreduce</td>
<td>3 / 1</td>
<td>35.99</td>
<td>450</td>
<td>5,459.84</td>
<td>14.69</td>
<td>1.58</td>
<td>52.63%</td>
</tr>
<tr>
<td>async Allreduce</td>
<td>4 / 1</td>
<td>31.94</td>
<td>450</td>
<td>10,255.85</td>
<td>18.74</td>
<td>1.78</td>
<td>44.47%</td>
</tr>
</tbody>
</table>

Table 8. Shows initial efficiency and usability metrics using the CIFAR (object classification) dataset. All tests were conducted using a 10-layer 3-stage Convolutional Neural Network. LoC represents the number of lines of code needed to implement parallelization, FC is the number of function calls, and O is the number of operands. Complexity is LoC * FC * O, where 0 is replaced with 1.

*Estimated Values

1. CSS 595 – Proposal Summary

A. Working Title

“Automated Parallelization to Improve Usability and Efficiency of Distributed Neural Network Training”
B. Goals

Increase usability and efficiency of distributed neural network training for embedded Systems on a Chip (SoC). Usability refers to how much effort is required of the user to take a network from a single to multiple nodes and will be measured in Lines of Code (LoC). Efficiency on an embedded system means more than just how fast it will train, this will be measured in speed, power consumption, and weight (size needed for Torch modules).

Quantitative or qualitative analysis between existing usability and my proposed usability. (LoCs, hours spent, user feedback)

C. Problem / Opportunity

Embedded systems typically have more constrained resources, because of this the training and parallelization of neural networks can be difficult and inefficient. There are three main problems when considering our goal. First, any framework that we use must be lightweight enough to operate in a resource constrained environment. Second, the parallelization implementation must be easy to use as we do not expect machine learning developers to be experts in distributed/parallel computing. Finally, the system must be resource conscious and efficient not only in speed but also in power consumption.

Current systems such as Torch 7.0 already provide a lightweight modular framework. But, there are improvements to be made in both the second and third problems. TorchMPI took approximately ~25 hours to build and get working and about 125 LoCs added to parallelize a Neural Network across 4 nodes. Additionally, A Convolution Neural Network across 4 nodes used on the CIFAR-10 database showed a speedup of 3.15x with a batch size of 500.

Scope:

- The focus will be on the TorchMPI framework with regard to our two goals (usability and efficiency).
- Only GPU training, although CPU training does show better speedup efficiency the total training times are still 4-5 times slower than GPU training.
- We will use existing neural networks and datasets.
- The embedded platform will be the Jetson TX/TX2 Developer Boards.

Autonomous Vehicles (AV)

Motivation: real-time learning for AVs

How does Torch connect to the development and usage of real-time learning for AV (cars, drones),

D. Stakeholders

Facebook, Twitter, and NVIDIA have a stake in Torch.

"Jetson TX2 brings powerful AI capabilities at the edge, making possible a new class of intelligent machines," said Deepu Talla, vice president and general manager of the Tegra business at NVIDIA. "These devices will enable intelligent video analytics that keep our cities smarter and safer, new kinds of robots that optimize manufacturing, and new collaboration that makes long-distance work more efficient."

Current Embedded projects that could benefit from this:
1. Intelligent Flying Machines (IFM) that’s developed a Jetson-powered autonomous drone. [https://www.youtube.com/watch?v=AMDiR61f86Y](https://www.youtube.com/watch?v=AMDiR61f86Y)

2. Horus - The Horus wearable device uses computer vision and machine learning to aid visually impaired people through the help of Jetson. [https://www.youtube.com/watch?v=9TEJC5F5nu8](https://www.youtube.com/watch?v=9TEJC5F5nu8)

3. MIT professor Sertac Karaman is teaching his students about robotics, drones and autonomous vehicles using the Jetson platform. [https://www.youtube.com/watch?v=V5Leg6dnw2w](https://www.youtube.com/watch?v=V5Leg6dnw2w)

4. Kespry - The NVIDIA Jetson TX1 supercomputer is designed to meet the demanding computational needs of visual computing applications like deep learning in the embedded space. [https://www.youtube.com/watch?v=wm4kCy8qNVc](https://www.youtube.com/watch?v=wm4kCy8qNVc)

E. Existing Systems

**Torch** - As described from torch.ch “Currently, Torch is a scientific computing framework with wide support for machine learning algorithms that puts GPUs first. It is easy to use and efficient, thanks to an easy and fast scripting language, LuaJIT, and an underlying C/CUDA implementation (torch.ch). TorchMPI, which is developed by the Facebook AI Research Group, already provides some parallel training capabilities.”

**TensorFlow** – As described from tensorflow.org “TensorFlow™ is an open source software library for numerical computation using data flow graphs. The flexible architecture allows you to deploy computation to one or more CPUs or GPUs in a desktop, server, or mobile device with a single API.”

TensorFlow and Torch are of two of the leading Open Source machine learning libraries available. They both show excellent performance benchmarks and have widespread support from both corporate and private contributors. Extensive add-on libraries are available for almost any application that may be needed.

Why choose Torch over TensorFlow or other popular machine learning tools? Even though DeepMind (Google) is shifting support to TensorFlow from Torch, I still believe Torch has several advantages that make it the right choice for my research.

1. Debugging - Torch is built for automatic differentiation (reverse-mode) while TensorFlow uses symbolic computation. This makes debugging painful as an error in the graph is harder to associate to a line in the code. ([https://www.quora.com/Is-TensorFlow-better-than-other-leading-libraries-such-as-Torch-Theano](https://www.quora.com/Is-TensorFlow-better-than-other-leading-libraries-such-as-Torch-Theano))

2. Still has support from large companies support such as Facebook, Twitter, and NVIDIA.

3. Smaller footprint. TensorFlow is built on python whose basic library is 824 Kb vs Torch which is built on Lua and their basic library is less than 100 Kb. Python does have more modules available than Lua, but these relate mostly to visualization and this is not required for our application and would cause unnecessary overhead. ([http://lua-users.org/wiki/LuaVersusPython](http://lua-users.org/wiki/LuaVersusPython))

4. Lua uses less memory.

5. Fewer external modules makes Lua easier to bundle for a specialized purpose.

6. Lua is already popular for embedded systems.

NVIDIA says, “Jetson TX2 offers twice the performance of its predecessor. This allows Jetson TX2 to run larger, deeper neural networks on edge devices. The result: smarter devices with higher accuracy and faster response times for tasks like image classification, navigation and speech recognition.”
F. What am I taking from existing systems

I will be building my package on top of the existing Torch 7.0 and TorchMPI framework. I will use existing neural networks and focus on those that provide capabilities most similar to those needed in autonomous embedded systems.

Networks and datasets from https://github.com/torch/demos

1. *Person detector (INRIA person dataset) 3-stage CNN plus 2-stage classifier
2. Tracker (uses real-time video) 3-stage CNN (network is already trained)
3. *Train-on-cifar (object recognition) CNN, MLP, Linear
4. Train-face-detector (4-stage CNN)

* Primary networks and datasets that will be used for this project.

G. What is my “value proposition”

Neural Networks have been around for decades, but it is only in the past 10 years that computing power and data storage capabilities have reached the level that it is truly viable. The push for autonomous systems such as autonomous cars and autonomous drones continues to drive forward heavily relying on deep learning. Although these systems will be given an already trained network there will still be the need to learn and adapt to the current environment and that is where my distributed learning capabilities will contribute to these embedded autonomous systems.

5. Conclusion

This quarter has been extremely beneficial! Through the power efficiency analysis, I not only gained familiarity with the Jetson platform and the associated software but also how to use power consumption measurement tools. This knowledge allowed me to quickly setup and configure my Jetson TX1 cluster (Cobra Cluster). My research into Game AIs helped me better understand the current state and challenges of parallelizing artificial intelligence algorithms.

Finally, my work with Torch and TorchMPI helped me identify the specific platform and problem that I am going to be working with. And to clearly define the scope of the work that I hope to accomplish. The working title for my Capstone is “Automated Parallelization to Improve Usability and Efficiency of Distributed Neural Network Training.” Usability will be defined by Lines of Code and a measurement of the complexity of those lines, and efficiency will be measured in terms of power consumption, speed, and weight.

All of this and along with the guidance of my professors, I feel well prepared to begin my Capstone work.
Appendix A: GPU-syncGradients Source Code

-- This script shows how to train different models on the CIFAR dataset, using multiple optimization techniques (SGD, ASGD, CG)
-- This script demonstrates a classical example of training well-known models (convnet, MLP, logistic regression)
-- on a 10-class classification problem.
-- It illustrates several points:
-- 1/ description of the model
-- 2/ choice of a loss function (criterion) to minimize
-- 3/ creation of a dataset as a simple Lua table
-- 4/ description of training and test procedures
--
-- Clement Farabet

require 'nn'
require 'optim'
require 'image'
require 'cunn'
require 'torchmpi'
local mpi = require('torchmpi')
mpi.start(true)

-- parse command-line options
--
dname,fname = sys.fpath()
cmd = torch.CmdLine()
cmd:text()
cmd:text('CIFAR Training')
cmd:text()
cmd:text('Options:')
cmd:option('-save', fname:gsub('.lua',''), 'subdirectory to save/log experiments in')
cmd:option('-network', '', 'reload pretrained network')
cmd:option('-model', 'convnet', 'type of model to train: convnet | mlp | linear')
cmd:option('-dataset', 2000, 'number of samples to use, up to 50,000')
cmd:option('-visualize', false, 'visualize input data and weights during training')
cmd:option('-seed', 1, 'fixed input seed for repeatable experiments')
cmd:option('-optimization', 'SGD', 'optimization method: SGD | ASGD | CG | LBFGS')
cmd:option('-learningRate', 1e-3, 'learning rate at t=0')
cmd:option('-batchSize', 1, 'mini-batch size (1 = pure stochastic)')
cmd:option('-weightDecay', 0, 'weight decay (SGD only)')
cmd:option('-momentum', 0, 'momentum (SGD only)')
cmd:option('-t0', 1, 'start averaging at t0 (ASGD only), in nb of epochs')
cmd:option('-maxIter', 5, 'maximum nb of iterations for CG and LBFGS')
cmd:option('-threads', 2, 'nb of threads to use')
cmd:option('-usegpu', false, 'use gpu for model training')
cmd:text()
opt = cmd:parse(arg)

-- fix seed
torch.manualSeed(opt.seed)

-- threads
torch.setnumthreads(opt.threads)
if mpi.rank() == 0 then
    print('<torch> set nb of threads to ' .. opt.threads)
end

-- define model to train
-- on the 10-class classification problem
--
classes = {'airplane', 'automobile', 'bird', 'cat',
    'deer', 'dog', 'frog', 'horse', 'ship', 'truck'}
if opt.network == '' then
    -- define model to train
    model = nn.Sequential()

    if opt.model == 'convnet' then
        -- convolutional network
        -- stage 1 : mean suppresion -> filter bank -> squashing -> max pooling
        model:add(nn.SpatialConvolutionMM(3, 32, 5, 5))
        model:add(nn.Tanh())
        model:add(nn.SpatialMaxPooling(3, 3, 3, 3, 1, 1))
        -- stage 2 : mean suppresion -> filter bank -> squashing -> max pooling
        model:add(nn.SpatialConvolutionMM(32, 64, 5, 5))
        model:add(nn.Tanh())
        model:add(nn.SpatialMaxPooling(2, 2, 2, 2, 1, 1))
        -- stage 3 : standard 2-layer MLP:
        model:add(nn.Reshape(64*3*3))
        model:add(nn.Linear(64*3*3, 200))
        model:add(nn.Tanh())
        model:add(nn.Linear(200, 10))
        --
    elseif opt.model == 'mlp' then
        -- regular 2-layer MLP
        model:add(nn.Reshape(3*32*32))
        model:add(nn.Linear(3*32*32, 1*32*32))
        model:add(nn.Tanh())
        model:add(nn.Linear(1*32*32, #classes))
elseif opt.model == 'linear' then

-- simple linear model: logistic regression

model:add(nn.Reshape(3*32*32))
model:add(nn.Linear(3*32*32,#classes))

else
  print('Unknown model type')
cmd: text()
  error()
end

else
  print('<trainer> reloading previously trained network')
  model = nn.Sequential()
  model: read(torch.DiskFile(opt.network))
end

-- retrieve parameters and gradients
parameters, gradParameters = model:getParameters()

-- verbose
if mpi.rank() == 0 then
  print('<cifar> using model:')
  print(model)
end

-- loss function: negative log-likelihood
--
  model:add(nn.LogSoftMax())
criterion = nn.ClassNLLCriterion()

-- get/create dataset
--
  trsize = opt.dataset
  tesize = opt.dataset / 4

-- download dataset
if not paths.dirp('cifar-10-batches-t7') then
  local www = 'http://torch7.s3.amazonaws.com/data/cifar-10-torch.tar.gz'
  local tar = paths.basename(www)
  os.execute('wget ' .. www .. '; .. 'tar xvf .. tar')
end

-- load dataset
trainData = {
    data = torch.Tensor(50000, 3072),
    labels = torch.Tensor(50000),
    size = function() return trsize end
}

for i = 0, 4 do
    subset = torch.load('cifar-10-batches-t7/data_batch_' .. (i+1) .. '.t7', 'ascii')
    trainData.data[{{i*10000+1, (i+1)*10000}}] = subset.data: t()
    trainData.labels[{{i*10000+1, (i+1)*10000}}] = subset.labels
end

trainData.labels = trainData.labels + 1

subset = torch.load('cifar-10-batches-t7/test_batch.t7', 'ascii')
testData = {
    data = subset.data:double(),
    labels = subset.labels[1]:double(),
    size = function() return tesize end
}
testData.labels = testData.labels + 1

-- resize dataset (if using small version)
trainData.data = trainData.data[{{1, trsize} }]
trainData.labels = trainData.labels[{{1, trsize} }]

testData.data = testData.data[{{1, tesize} }]
testData.labels = testData.labels[{{1, tesize} }]

-- reshape data
trainData.data = trainData.data:reshape(trsize, 3, 32, 32)
testData.data = testData.data:reshape(tesize, 3, 32, 32)

-- preprocess/trainSet
normalization = nn.SpatialContrastiveNormalization(1, image.gaussian1D(7))
for i = 1, trainData:size() do
    local rgb = trainData.data[i]
    local yuv = image.rgb2yuv(rgb)
    -- normalize y locally:
    yuv[1] = normalization(yuv[{{1}}])
    trainData.data[i] = yuv
end

-- normalize u globally:
mean_u = trainData.data[{{}, 2, {}, {}}]:mean()
std_u = trainData.data[{},2,{},{}].std()
trainData.data[{},2,{},{}].add(-mean_u)
trainData.data[{},2,{},{}].div(-std_u)
-- normalize v globally:
mean_v = trainData.data[{},3,{},{}].mean()
std_v = trainData.data[{},3,{},{}].std()
trainData.data[{},3,{},{}].add(-mean_v)
trainData.data[{},3,{},{}].div(-std_v)
---
-- preprocess testSet
for i = 1,testData:size() do
  -- rgb -> yuv
  local rgb = testData.data[i]
  local yuv = image.rgb2yuv(rgb)
  -- normalize y locally:
  yuv[{1}] = normalization(yuv[{1}])
  testData.data[i] = yuv
end
-- normalize u globally:
trainData.data[{},2,{},{}].add(-mean_u)
trainData.data[{},2,{},{}].div(-std_u)
-- normalize v globally:
trainData.data[{},3,{},{}].add(-mean_v)
trainData.data[{},3,{},{}].div(-std_v)
---
-- define training and testing functions
--
-- this matrix records the current confusion across classes
confusion = optim.ConfusionMatrix(classes)

-- display function
function display(input)
  iter = iter or 0
  require 'image'
  win_input = image.display{image=input, win=win_input, zoom=2, legend='input'}
  if iter % 10 == 0 then
    if opt.model == 'convnet' then
      win_w1 = image.display{
        image=model:get(1).weight, zoom=4, nrow=10,
        min=-1, max=1,
        win=win_w1, legend='stage 1: weights', padding=1
      }
      win_w2 = image.display{
        image=model:get(4).weight, zoom=4, nrow=30,
        min=-1, max=1,
        win=win_w2, legend='stage 2: weights', padding=1
      }
    elseif opt.model == 'mlp' then
      local W1 = torch.Tensor(model:get(2).weight):resize(2048,1024)
    end
  end
end
win_w1 = image.display{
    image=W1, zoom=0.5, min=-1, max=1,
    win=win_w1, legend='W1 weights'
}
local W2 = torch.Tensor(model:get(2).weight):resize(10,2048)
win_w2 = image.display{
    image=W2, zoom=0.5, min=-1, max=1,
    win=win_w2, legend='W2 weights'
}
end
end
iter = iter + 1
end
local mpinn = require('torchmpi.nn')
mpinn.synchronizeParameters(model)
stripe = trsize / mpi.size()
if mpi.rank() == 0 then
    print('stripe = ' .. stripe)
end
-- training function
function train(dataset)
    -- epoch tracker
    epoch = epoch or 1
    if opt.usegpu then
        if mpi.rank() == 0 then
            print('setting model and dataset to GPU in train')
        end
        model:cuda()
        dataset.data = dataset.data:cuda()
        dataset.labels = dataset.labels:cuda()
        criterion = criterion:cuda()
    end
    -- local vars
    local time = sys.clock()
    local trainError = 0
    -- do one epoch
    print('<trainer> on training set:')
    print('<trainer> online epoch # ' .. epoch .. ' [batchSize = ' .. opt.batchSize .. ']')
    for t = (mpi.rank() * stripe) + 1,dataset:size() - ((mpi.size() - (mpi.rank() + 1)) * stripe),opt.batchSize do
        -- disp progress
        if mpi.rank() == 0 then
            xlua.progress((t - 1) / opt.batchSize, (dataset:size()/opt.batchSize) / mpi.size())
        end
        -- create mini batch
local inputs = {}
local targets = {}
for i = 1, math.min(t + opt.batchSize - 1, dataset:size()) do
    -- load new sample
    local input = dataset.data[i]
    local target = dataset.labels[i]
    table.insert(inputs, input)
    table.insert(targets, target)
end

-- create closure to evaluate f(X) and df/dX
local feval = function(x)
    -- get new parameters
    if x ~= parameters then
        parameters:copy(x)
    end

    -- reset gradients
    gradParameters:zero()

    -- f is the average of all criterions
    local f = 0

    -- evaluate function for complete mini batch
    for i = 1, #inputs do
        -- estimate f
        local output = model:forward(inputs[i])
        local err = criterion:forward(output, targets[i])
        f = f + err

        -- estimate df/dW
        local df_do = criterion:backward(output, targets[i])
        model:backward(inputs[i], df_do)
            -- only sync at the end of a batch
            if i == opt.batchSize then
                -----------Syncing Gradients-------------
                mpinn.synchronizeGradients(model)
            end

        -- update confusion
        confusion:add(output, targets[i])
        -- visualize?
        if opt.visualize then
            display(inputs[i])
        end
    end

    -- normalize gradients and f(X)
    gradParameters:div(#inputs)
    f = f / #inputs
trainError = trainError + f

-- return f and df/dX
return f, gradParameters
end

-- optimize on current mini-batch
if opt.optimization == 'CG' then
    config = config or {maxIter = opt.maxIter}
    optim.cg(feval, parameters, config)
elseif opt.optimization == 'LBFGS' then
    config = config or {learningRate = opt.learningRate,
                       maxIter = opt.maxIter,
                       nCorrection = 10}
    optim.lbfgs(feval, parameters, config)
elseif opt.optimization == 'SGD' then
    config = config or {learningRate = opt.learningRate,
                        weightDecay = opt.weightDecay,
                        momentum = opt.momentum,
                        learningRateDecay = 5e-7}
    optim.sgd(feval, parameters, config)
elseif opt.optimization == 'ASGD' then
    config = config or {eta0 = opt.learningRate,
                        t0 = nbTrainingPatches * opt.t0}
    __.average = optim.asgd(feval, parameters, config)
else
    error('unknown optimization method')
end
end

-- train error
trainError = trainError / math.floor(dataset:size()/opt.batchSize)

if mpi.rank() == 0 then
    -- time taken
    time = sys.clock() - time
    time = time / dataset:size()
    print("<trainer> time to learn 1 sample = ", (time*1000), 'Sec')

    -- print confusion matrix
    print(confusion)
    local trainAccuracy = confusion.totalValid * 100
    confusion:zero()
end

-- next epoch
epoch = epoch + 1
function test(dataset)
  local vars
  local testError = 0
  local time = sys.clock()

  if opt.usegpu then
    print('setting model and dataset to GPU in test')
    model:cuda()
    dataset.data = dataset.data:cuda()
    dataset.labels = dataset.labels:cuda()
    criterion = criterion:cuda()
  end

  -- averaged param use?
  if average then
    cachedparams = parameters:clone()
    parameters:copy(average)
  end

  -- test over given dataset
  print('<trainer> on testing Set:')
  for t = 1,dataset:size() do
    -- disp progress
    xlua.progress(t, dataset:size())

    -- get new sample
    local input = dataset.data[t]
    local target = dataset.labels[t]

    -- test sample
    local pred = model:forward(input)
    confusion:add(pred, target)

    -- compute error
    err = criterion:forward(pred, target)
    testError = testError + err
  end

  -- timing
  time = sys.clock() - time
  time = time / dataset:size()
  print("<trainer> time to test 1 sample = " .. (time*1000) .. 'Sec')

  -- testing error estimation
  testError = testError / dataset:size()
-- print confusion matrix
print(confusion)
local testAccuracy = confusion.totalValid * 100
confusion:zero()

-- averaged param use?
if average then
    -- restore parameters
    parameters:copy(cachedparams)
end

return testAccuracy, testError
end

-- and train!
--

sys.tic()
trainAcc, trainErr = train(trainData)
mpi.barrier()
t = sys.toc()

if mpi.rank() == 0 then
    testAcc, testErr = test(testData)
end

if mpi.rank() == 0 then
    print('Num Nodes:      ', mpi.size())
    print('Dataset Size:   ', trsize)
    print('Train Time:     ', t)
end

mpi.stop()