UNLOCK PERFORMANCE LIMIT OF DNN BY CUDA® IN R

Patric Zhao, GPU Architect, NVIDIA
patricz@nvidia.com
AGENDA

1. Background

2. Build DNN by R language

3. CUDA Accelerations and Optimizations

4. Scale out by Multi-GPUs

5. Summary
BACKGROUND
DNN: Deep Neural Network
- Great successful in CV, NLP, etc.
- Automatic Feature Extraction
- Computation intensive algorithm
- Still a rapid development field
### Mature Packages in R:

<table>
<thead>
<tr>
<th>Packages</th>
<th>Backend</th>
<th>Compute Resources</th>
</tr>
</thead>
<tbody>
<tr>
<td>nnet</td>
<td>C/C++</td>
<td>Single thread</td>
</tr>
<tr>
<td>neuralnet</td>
<td>C/C++</td>
<td>Single thread</td>
</tr>
<tr>
<td>DARCH</td>
<td>C/C++</td>
<td>Single thread</td>
</tr>
<tr>
<td>deepnet</td>
<td>C/C++</td>
<td>Single thread</td>
</tr>
<tr>
<td>H2O</td>
<td>JAVA</td>
<td>Multi-threads, multi-nodes</td>
</tr>
<tr>
<td><strong>mxnet</strong></td>
<td><strong>C/C++/CUDA</strong></td>
<td><strong>Multi-threads, GPUs, multi-nodes</strong></td>
</tr>
</tbody>
</table>

S6853 - MXNet: Flexible Deep Learning Framework from Distributed GPU Clusters to Embedded Systems  
L6143 - Train and Deploy Deep Learning for Vision, Natural Language and Speech Using MXNet
In this talk, I am going to introduce how to:

- Build DNN network by native R
- Accelerate R code under CUDA ecosystem
- Make our solutions as simple as possible

![Diagram showing performance and flexibility comparison between C/C++, CRAN, and R+CUDA]
BUILD DNN BY R LANGUAGE
- Classification Network
  - 1 hidden layer w/ softmax, ReLu
- Vectorization Representation
- Fully connected network
- Python to R translations
Core complements of DNN in R:

Weights and Bias: matrix representation

```r
weight <- 0.01*matrix(rnorm(h*k), nrow=h, ncol=k)
bias <- matrix(0, nrow=1, ncol=H)
```

Neuron: computation parts

```r
neuron <- sweep(input %*% weights ,2, bias, '+')
neuron <- pmax(neuron, 0)  # ReLu
```

Cost function: Softmax

```r
score.exp <- exp(score)
probs <- sweep(score.exp, 1, rowSums(score.exp), '/')
```
Prediction: Feed Forward

```r
predict <- function(model, data = X.test) {
  new.data <- data.matrix(data)
  # Feed Forward
  hidden.layer <- sweep(new.data %*% model$W1 ,2, model$b1, '+')
  # neurons: Rectified Linear
  hidden.layer <- pmax(hidden.layer, 0)
  score <- sweep(hidden.layer %*% model$W2, 2, model$b2, '+')
  # Loss Function: softmax
  score.exp <- exp(score)
  probs <- sweep(score.exp, 1, rowSums(score.exp), '/')
  labels.predicted <- max.col(probs)
  return(labels.predicted)
}
```
Training: Feed Forward + Back propagation

```r
train <- function(x, y, model, traindata, hidden,...) {
    # 1. Feed Forward . . .
    # 2. Compute the loss . . .
    # 3. Backward
    dscores <- probs
    dscores[Y.index] <- dscores[Y.index] -1
    dscores <- dscores / batchsize
    dW2 <- t(hidden.layer) %*% dscores
    db2 <- colSums(dscores)
    dhidden <- dscores %*% t(W2)
    dhidden[hidden.layer <= 0] <- 0
    dW1 <- t(X) %*% dhidden
    db1 <- colSums(dhidden)
    # update ....
}
```
CUDA ACCELERATIONS AND OPTIMIZATIONS
RECAP: How to accelerate R by CUDA?

My GTC15 talk: Accelerate R Applications with CUDA
Benchmark: **MNIST** handwritten digit dataset

- Input features: 28X28 = 784, Output classes: 10 (0-9);
- Training Set 60,000, testing set: 10,000
- DNN Architecture: 2-layers fully connected neural network

**Performance comparison of R.DNN and H2O**

CPU: Ivy Bridge E5-2690 v2 @ 3.00GHz, dual socket 10-core, 128G RAM;
## Profiling

### Rprof(), summaryRprof()

<table>
<thead>
<tr>
<th>Break Down R DNN Runtime</th>
<th>total.time</th>
<th>total.pct</th>
<th>self.time</th>
<th>self.pct</th>
</tr>
</thead>
<tbody>
<tr>
<td>train.dnn</td>
<td>1386</td>
<td>100</td>
<td>9.74</td>
<td>0.7</td>
</tr>
<tr>
<td>%*%</td>
<td>1250.08</td>
<td>90.19</td>
<td>1250.08</td>
<td>90.19</td>
</tr>
<tr>
<td>sweep</td>
<td>676.32</td>
<td>48.8</td>
<td>1.58</td>
<td>0.11</td>
</tr>
<tr>
<td>t</td>
<td>61.64</td>
<td>4.45</td>
<td>0.02</td>
<td>0</td>
</tr>
<tr>
<td>t.default</td>
<td>61.62</td>
<td>4.45</td>
<td>61.62</td>
<td>4.45</td>
</tr>
<tr>
<td>pmax</td>
<td>28.42</td>
<td>2.05</td>
<td>24.4</td>
<td>1.76</td>
</tr>
<tr>
<td>aperm</td>
<td>21.96</td>
<td>1.58</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>aperm.default</td>
<td>11.6</td>
<td>0.84</td>
<td>11.6</td>
<td>0.84</td>
</tr>
<tr>
<td>array</td>
<td>10.36</td>
<td>0.75</td>
<td>10.36</td>
<td>0.75</td>
</tr>
<tr>
<td>&lt;=</td>
<td>5.72</td>
<td>0.41</td>
<td>5.72</td>
<td>0.41</td>
</tr>
<tr>
<td>mostattributes&lt;-</td>
<td>4.02</td>
<td>0.29</td>
<td>4.02</td>
<td>0.29</td>
</tr>
<tr>
<td>exp</td>
<td>3.6</td>
<td>0.26</td>
<td>3.6</td>
<td>0.26</td>
</tr>
<tr>
<td>unname</td>
<td>1.46</td>
<td>0.11</td>
<td>0.18</td>
<td>0.01</td>
</tr>
<tr>
<td>is.data.frame</td>
<td>1.28</td>
<td>0.09</td>
<td>1.28</td>
<td>0.09</td>
</tr>
<tr>
<td>data.matrix</td>
<td>1.28</td>
<td>0.09</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>colSums</td>
<td>0.86</td>
<td>0.06</td>
<td>0.86</td>
<td>0.06</td>
</tr>
<tr>
<td>/</td>
<td>0.52</td>
<td>0.04</td>
<td>0.52</td>
<td>0.04</td>
</tr>
<tr>
<td>rowSums</td>
<td>0.36</td>
<td>0.03</td>
<td>0.36</td>
<td>0.03</td>
</tr>
<tr>
<td>-</td>
<td>0.04</td>
<td>0</td>
<td>0.04</td>
<td>0</td>
</tr>
<tr>
<td>sum</td>
<td>0.02</td>
<td>0</td>
<td>0.02</td>
<td>0</td>
</tr>
</tbody>
</table>

![Pie chart showing percentages of R commands in DNN implementation](chart.png)

- %*%: 90.19
- t.default: 4.45
- pmax: 1.76
- Others: 3.59
DROP-IN ACCELERATION

By nvBLAS Library on Linux

> env LD_PRELOAD=libnvblas.so   R CMD BATCH MNIST_DNN.R
### OPTIMIZATIONS

- Profiling again after NVIDIA GPU acceleration

<table>
<thead>
<tr>
<th>function</th>
<th>total.time</th>
<th>total.pct</th>
<th>self.time</th>
<th>self.pct</th>
</tr>
</thead>
<tbody>
<tr>
<td>train.dnn</td>
<td>274</td>
<td>100</td>
<td>10.74</td>
<td>3.92</td>
</tr>
<tr>
<td><code>%*%</code></td>
<td>114.58</td>
<td>41.82</td>
<td>114.58</td>
<td>41.82</td>
</tr>
<tr>
<td>sweep</td>
<td><strong>87.28</strong></td>
<td>31.85</td>
<td><strong>1.8</strong></td>
<td><strong>0.66</strong></td>
</tr>
<tr>
<td>t.default</td>
<td><strong>73.42</strong></td>
<td>26.8</td>
<td><strong>73.42</strong></td>
<td><strong>26.8</strong></td>
</tr>
<tr>
<td>t</td>
<td><strong>73.42</strong></td>
<td>26.8</td>
<td><strong>0</strong></td>
<td><strong>0</strong></td>
</tr>
<tr>
<td>pmax</td>
<td>30.9</td>
<td>11.28</td>
<td>24.62</td>
<td>8.99</td>
</tr>
<tr>
<td>aperm</td>
<td>29.74</td>
<td>10.85</td>
<td>0.04</td>
<td>0.01</td>
</tr>
<tr>
<td>aperm.default</td>
<td>19.08</td>
<td>6.96</td>
<td>19.04</td>
<td>6.95</td>
</tr>
<tr>
<td>array</td>
<td>10.62</td>
<td>3.88</td>
<td>10.62</td>
<td>3.88</td>
</tr>
<tr>
<td>mostattributes&lt;-</td>
<td>6.28</td>
<td>2.29</td>
<td>6.26</td>
<td>2.28</td>
</tr>
</tbody>
</table>
Opt.1: replace $t(X) \%\% \text{matrix}$ and $\text{matrix} \%\% t(X)$ with R internal function

```r
# original: t() with matrix multiplication
dW2 <- t(hidden.layer) \%\% dscores
dhidden <- dscores \%\% t(W2)

# Opt1: use builtin function
dW2 <- \text{crossprod}(hidden.layer, dscores)
dhidden <- \text{tcrossprod}(dscores, W2)
```
Opt.2: replace `sweep()` by matrix multiplication

```r
# Opt2: original code
hidden.layer <- sweep(X %*% W1 ,2, b1, '+')
```

Matrix Multiplication

Sweep add bias

```
b11 b1n
```
Opt.2: replace `sweep()` by matrix multiplication

```r
# Opt2: remove `sweep`
hidden.layer <- X1 %*% W1b1
```

Matrix Multiplication

```
X1 <- cbind(X, rep(1, nrow(X)))
W1b1 <- rbind(W1, b1)
```
<table>
<thead>
<tr>
<th>Function</th>
<th>Original</th>
<th>Opt1: replace <code>t()</code></th>
<th>Opt2: remove <code>sweep()</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>%*%</code></td>
<td>112.02</td>
<td>53.28</td>
<td>53.72</td>
</tr>
<tr>
<td><code>sweep</code></td>
<td>90.46</td>
<td>86.7</td>
<td>-</td>
</tr>
<tr>
<td><code>t</code></td>
<td>73.98</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td><code>t.default</code></td>
<td>73.96</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td><code>aperm</code></td>
<td>33.34</td>
<td>30.78</td>
<td>-</td>
</tr>
<tr>
<td><code>pmax</code></td>
<td>32.52</td>
<td>31.44</td>
<td>31.58</td>
</tr>
<tr>
<td><code>aperm.default</code></td>
<td>22.36</td>
<td>19.84</td>
<td>-</td>
</tr>
<tr>
<td><code>array</code></td>
<td>10.98</td>
<td>10.92</td>
<td>-</td>
</tr>
<tr>
<td><code>crossprod</code></td>
<td>-</td>
<td>23.06</td>
<td>23.4</td>
</tr>
<tr>
<td><code>tcrossprod</code></td>
<td>-</td>
<td>2.52</td>
<td>2.54</td>
</tr>
<tr>
<td><code>cbind</code></td>
<td>-</td>
<td>-</td>
<td>8.9</td>
</tr>
<tr>
<td><strong>Total (sec)</strong></td>
<td>266.28</td>
<td>166.76</td>
<td>144.71</td>
</tr>
<tr>
<td><strong>Speedup</strong></td>
<td>1X</td>
<td>1.60X</td>
<td>1.84X</td>
</tr>
</tbody>
</table>

GREEN: GPU accelerated parts  
RED: Performance limiters
Opt. 3: Implement `pmax()` by CUDA

- `.Call()` function in R with simple CUDA implementation of `pmax()`

(w/ `.C()` to call cuBLAS API and on Windows Platform)

```r
# preload static object file
dyn.load("cudaR.so")

# GPU version of ReLU (pmax)
pmax.cuda <- function(A, threshold, devID=0)
{
  rst <- `.Call("pmax_cuda", A, threshold, as.integer(devID))`
  dim(rst) <- dim(A)
  return(rst)
}
```
// CUDA: simple implementation of pmax
__global__ void pmax_kernel(double *A, const int M, const int N, const double threshold) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    return;
}
// Specified for DNN by .CALL format
SEXP pmax_cuda(SEXP A, SEXP threshold) {
    // Initialization including R to C data transfer, CUDA preparations
    ... 
    pmax_kernel<<<(mm * nn - 1) / 512 + 1, 512>>>(A_d, mm, nn, gw);
    cudaMemcpy(REAL(Rval), A_d, mm * nn * sizeof(double), cudaMemcpyDeviceToHost);
    cudaDeviceSynchronize();
    // Free data, unprotect ...
    return Rval;
}
Final Profiling:

<table>
<thead>
<tr>
<th>by.total</th>
<th>native R</th>
<th>nvBLAS + R</th>
<th>CUDA</th>
</tr>
</thead>
<tbody>
<tr>
<td>train.dnn</td>
<td>1436</td>
<td>278.7</td>
<td>167.7</td>
</tr>
<tr>
<td>%*%</td>
<td>1250.08</td>
<td>112.02</td>
<td>53.28</td>
</tr>
<tr>
<td>sweep</td>
<td>676.32</td>
<td>90.46</td>
<td>86.7</td>
</tr>
<tr>
<td>t</td>
<td>61.64</td>
<td>73.98</td>
<td>-</td>
</tr>
<tr>
<td>t.default</td>
<td>61.62</td>
<td>73.96</td>
<td>-</td>
</tr>
<tr>
<td>aperm</td>
<td>21.96</td>
<td>33.34</td>
<td>30.78</td>
</tr>
<tr>
<td>pmax</td>
<td>28.42</td>
<td>32.52</td>
<td>31.44</td>
</tr>
<tr>
<td>aperm.default</td>
<td>11.6</td>
<td>22.36</td>
<td>19.84</td>
</tr>
<tr>
<td>array</td>
<td>10.36</td>
<td>10.98</td>
<td>10.92</td>
</tr>
<tr>
<td>crossprod</td>
<td>-</td>
<td>-</td>
<td>23.06</td>
</tr>
<tr>
<td>tcrossprod</td>
<td>-</td>
<td>-</td>
<td>2.52</td>
</tr>
<tr>
<td>cbind</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Total</td>
<td>1425.946</td>
<td>266.28</td>
<td>166.76</td>
</tr>
<tr>
<td>Speedup</td>
<td>1X</td>
<td>1.60X</td>
<td>1.84X</td>
</tr>
</tbody>
</table>

Optimizations for R DNN (HU=64)
Performance on Linux

R DNN: Parallel Acceleration from NVIDIA GPU

- 24 min for native
- 2 min for OPT+GPU
- 9 min for H2O 1-thread
- 5 min for H2O 20-threads

200 training steps of 1 hidden layer with 64 Units
SCALE OUT BY MULTI-GPUS
DATA PARALLEL BY HOGWILD!

HOGWILD!
- A lock-free approach to parallelizing stochastic gradient descent
- MapReduce-like parallel-processing framework

DNN Training
- Launch several workers
- Each worker updates local weights/bias based on parts (1/N) of data
- Master collects and average all weights/bias from each worker
- Each worker update its weights/bias
Extend ‘multicores’ solution to multiGPUs

R master

mclapply

fork

weights updates

mclapply

fork

P1 | offload tasks | GPU0

P2

R workers

GPU1

GPUs

GPU0

GPU1
DATA DECOMPOSITION

- mclapply function to map data into each R processor

```r
# Parallel Training
res <- mclapply(1:devNum, function(id) {
  train.dnn.cublas(x, y, omodel=para.model,
                  traindata=traindata[N.start[id]:N.end[id]],
                  devType="GPU", devID=(id-1), . . .)
  mc.cores=devNum, mc.preschedule=TRUE)

# Model Update
para.model <- list(W1= W1.sum/devNum, b1= b1.sum/devNum,
                   W2= W2/si, /devNum, b2= b2.sum/devNum)
```
OFFLOAD TASKS TO GPUs

- Explicitly call cuBLAS API and pmax.cuda functions
- Set the GPU ID based on R’s thread ID

```r
# R level function call
res <- cuBLAS(hidden.layer, dscores, transA=T, devID=devID)

// GEMM cuda call by .Call format and simplified for DNN
SEXP gemm_cuda(SEXP A, SEXP B, SEXP transA, SEXP transB, SEXP devID)
{
  // init . . .
  cudaSetDevice(gpuID);
  // cuBLAS: double precision matrix multiplication, DGEMM
  cublasDgemm(handle, cuTransA, cuTransB, mt, nt, kt, . . .);
  . . .
}
```
PERFORMANCE IMPROVEMENTS

CPU: Ivy Bridge E5-2690 v2 @ 3.00GHz, dual socket 10-core, 128G RAM; GPU: NVIDIA K40m, 12G RAM
In this talk, we introduce solutions for HPA in R to

- Keep flexibility
- Achieve high speedup for native R code
- Extend multicore solution to multiGPUs
- Easy to apply these methods to multiple NN & other R algorithms

Further Works:

- Memory Optimizations
- Data Dependency Analysis
- Heterogeneous Computing both in CPU and GPU
Related Materials:

CODES:

All codes, scripts and window templates in this talk in [here](#)

TALKS:

- GTC15: Accelerate R by CUDA, [slide](#)
- GTC16: Data Science Applications of GPUs in the R Language

BLOG:

- Parallel FORALL, [post](#)
- ParallelR, R For Deep Learning:
  - (I) [Build Fully Connected Neural Network From Scratch](#)
  - (II) [Achieve High-Performance DNN With Parallel Acceleration](#)
  - (III) [CUDA Acceleration And MultiGPUs Training](#)
THANK YOU

JOIN THE CONVERSATION
#GTC16  

Twitter  Facebook  LinkedIn
NOTE: Just to show the correctness of our codes and methods rather than achieve high accuracy of MNIST
gputools .vs. nvblas

Total seconds: time spent in function and calls.
Self seconds: time spent in function alone.

<table>
<thead>
<tr>
<th>% total</th>
<th>% self</th>
<th>self seconds</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>100.0</td>
<td>100.0</td>
<td>4.64</td>
<td>&quot;eval&quot;</td>
</tr>
<tr>
<td>100.0</td>
<td>100.0</td>
<td>4.64</td>
<td>&quot;gpuCrossprod&quot;</td>
</tr>
<tr>
<td>100.0</td>
<td>100.0</td>
<td>4.64</td>
<td>&quot;source&quot;</td>
</tr>
<tr>
<td>100.0</td>
<td>100.0</td>
<td>4.64</td>
<td>&quot;system.time&quot;</td>
</tr>
<tr>
<td>84.5</td>
<td>84.5</td>
<td>3.92</td>
<td>&quot;withVisible&quot;</td>
</tr>
<tr>
<td>15.5</td>
<td>15.5</td>
<td>0.72</td>
<td>&quot;t.default&quot;</td>
</tr>
<tr>
<td>15.5</td>
<td>15.5</td>
<td>0.72</td>
<td>&quot;t&quot;</td>
</tr>
</tbody>
</table>

Total seconds: time spent in function and calls.
Self seconds: time spent in function alone.

<table>
<thead>
<tr>
<th>% self</th>
<th>% total</th>
<th>self seconds</th>
<th>total seconds</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>84.5</td>
<td>84.5</td>
<td>3.92</td>
<td>3.92</td>
<td>&quot;withVisible&quot;</td>
</tr>
<tr>
<td>15.5</td>
<td>15.5</td>
<td>0.72</td>
<td>0.72</td>
<td>&quot;t.default&quot;</td>
</tr>
</tbody>
</table>

Total seconds: time spent in function and calls.
Self seconds: time spent in function alone.

<table>
<thead>
<tr>
<th>% total</th>
<th>% self</th>
<th>self seconds</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>100.0</td>
<td>100.0</td>
<td>2.18</td>
<td>&quot;&lt;Anonymous&gt;&quot;</td>
</tr>
<tr>
<td>100.0</td>
<td>100.0</td>
<td>2.18</td>
<td>&quot;crossprod&quot;</td>
</tr>
<tr>
<td>100.0</td>
<td>100.0</td>
<td>2.18</td>
<td>&quot;eval&quot;</td>
</tr>
<tr>
<td>100.0</td>
<td>100.0</td>
<td>2.18</td>
<td>&quot;source&quot;</td>
</tr>
<tr>
<td>100.0</td>
<td>100.0</td>
<td>2.18</td>
<td>&quot;StandardGeneric&quot;</td>
</tr>
<tr>
<td>100.0</td>
<td>100.0</td>
<td>2.18</td>
<td>&quot;system.time&quot;</td>
</tr>
<tr>
<td>100.0</td>
<td>100.0</td>
<td>2.18</td>
<td>&quot;withVisible&quot;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>% self</th>
<th>% total</th>
<th>self seconds</th>
<th>total seconds</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>100.0</td>
<td>100.0</td>
<td>2.18</td>
<td>2.18</td>
<td>&quot;&lt;Anonymous&gt;&quot;</td>
</tr>
</tbody>
</table>