1 GPU speedups training a logistic regression classifier in PyTorch

A binary logistic regression classifier was implemented in PyTorch (v1.4.0) and trained over the rcv1 dataset to illustrate the speed ups possible using a GPU (Nvidia Tesla V100) versus only multithreading (24 CPU threads using an Intel Xeon Gold 5118). Speedups were tested for both batch gradient descent (with a 0.001 learning rate) and L-BFGS. The rcv1 dataset was downloaded from https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/binary/rcv1_train.binary.bz2. Gradient descent converged after 3,000 iterations and L-BFGS converged after 100 iterations. For reference, a logistic regression classifier was trained using single-threaded TRON (as implemented in scikit-learn v0.20.4). All code is available in pyTorchLogisticRegression_rcv1.py.

<table>
<thead>
<tr>
<th>Solver</th>
<th>CPU training time (s)</th>
<th>GPU training time (s)</th>
<th>GPU Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gradient descent</td>
<td>395.58</td>
<td>27.05</td>
<td>14.63</td>
</tr>
<tr>
<td>L-BFGS</td>
<td>40.56</td>
<td>3.1</td>
<td>13.08</td>
</tr>
<tr>
<td>TRON (scikit-learn)</td>
<td>0.29</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>

Table 1: Logistic regression training times, measured in seconds, for the rcv1 dataset. Gradient descent and L-BFGS solvers are implemented in PyTorch, and single-threaded TRON is implemented in scikit-learn.

2 Derivation of TRON Hessian-vector products

Consider feature vectors \( x_i \in \mathbb{R}^n, i = 1, \ldots, l \) and label vector \( y \in \{-1, 1\}^l \), and let \( X = [x_1 \ldots x_l]^T \) be the feature matrix. For vectors, index-set subscripts denote subvectors and for matrices, pairs of index-set subscripts denote submatrices. Let \( 1 \) denote the indicator function.

The general \( L_2 \)-regularized objective, which we wish to minimize w.r.t. \( w \), is

\[
 f(w) = \frac{1}{2} w^T w + C \sum_{i=1}^l \ell(w; x_i, y_i),
\]

where \( \frac{1}{2} w^T w \) is the regularization term, \( C > 0 \) is a regularization hyperparameter, and \( \ell(w; x_i, y_i) \) is a loss function. When \( \ell(w; x_i, y_i) = \log(1 + \exp(-y_i w^T x_i)) \), commonly referred to as the logistic loss, minimizing Equation 1 corresponds to learning a classifier using logistic regression. Similarly, minimizing Equation 1 when \( \ell(w; x_i, y_i) = (\max(0, 1 - y_i w^T x_i))^2 \), commonly referred to as the L2-SVM or quadratic SVM loss, corresponds to learning a linear SVM classifier. The logistic loss results in an objective function that is twice differentiable and the L2-SVM loss yields a differentiable
TRON is detailed in Algorithm 1. At each iteration, given the current parameters \( w \) and trust region interval \( \Delta \), TRON considers the following quadratic approximation to \( f(w + d) - f(w) \),

\[
q(d) = \nabla f(w)^T d + \frac{1}{2} d^T \nabla^2 f(w) d.
\] (2)

A truncated Newton step, confined in the trust region, is then found by solving

\[
\min_d q(d) \quad \text{s.t.} \quad \|d\|_2 \leq \Delta.
\] (3)

If \( q(d) \) is close to \( f(w + d) - f(w) \), \( w \) is updated to \( w + d \) and the trust region interval is increased for the subsequent iteration. Otherwise, \( w \) remains unchanged and the trust region interval is shrunk.

**Algorithm 1** The TRON algorithm

1: Given \( w, \Delta, \) and \( \sigma_0 \)
2: Calculate \( f(w) \) \quad \text{// Critically depends on } z = X^T w
3: while Not converged do
4: \quad Find \( d = \arg\min_v q(v) \text{ s.t.} \|v\|_2 \leq \Delta. \) \quad \text{// Critically depends on } \nabla f(w), \nabla^2 f(w)v
5: \quad Calculate \( f(w + d), \sigma = \frac{f(w + d) - f(w)}{q(d)} \) \quad \text{// Critically depends on } z = X^T (w + d)
6: \quad if \( \sigma > \sigma_0 \) then
7: \quad \quad \quad \quad \quad \quad \quad w \leftarrow w + d, \text{ increase trust region } \Delta.
8: \quad else
9: \quad \quad Shrink \( \Delta \).
10: \quad end if
11: end while

Note that the function evaluation \( f(w) \) must be computed for each new iteration, as well as the gradient and the Hessian for Equation 2. However, Equation 2 involves only a Hessian-vector product, computation of which circumvents loading the entire Hessian into memory. For the logistic loss, we have

\[
\nabla f_{LR}(w) = w + C \sum_{i=1}^{I} (h(y_i w^T x_i) - 1) y_i x_i,
\] (4)

where \( h(y_i w^T x_i) = (1 + e^{-y_i w^T x_i})^{-1} \). For the L2-SVM loss, we have

\[
\nabla f_{L2}(w) = w + 2C \hat{X}^T \hat{z} = w + 2C X_{I,:}^T (X_{I,:} w - y),
\] (5)

where \( I \equiv \{ i | 1 - y_i w^T x_i > 0 \} \) is an index set and and the operator : denotes all elements along the corresponding dimension (i.e., all columns in this case). Thus, \( X_{I,:} \) is the submatrix of all \( X \) rows the indices of which are in \( I \).

Equation 3 involves only a single Hessian-vector product, the structure of which is exploited to avoid loading the entire Hessian into memory. For the logistic loss, we have

\[
\nabla^2 f_{LR}(w) = I + C X^T DX,
\] (6)

where \( D \) is a diagonal matrix with elements \( D_{i,i} = h(y_i w^T x_i) (1 - h(y_i w^T x_i)) \). Thus, for a vector \( v \), the Hessian-vector product is efficiently computed as \( \nabla^2 f_{LR}(w)v = v + C X^T (D(X)v) \). For the L2-SVM loss, we have

\[
\nabla^2 f_{L2}(w) = I + 2C X^T DX = I + 2C X_{I,:}^T X_{I,:},
\] (7)

where \( D \) is a diagonal matrix with elements \( D_{i,i} = 1_{i \in I} \). The Hessian-vector product is thus efficiently computed as \( \nabla^2 f_{L2}(w)v = v + 2C X_{I,:}^T (X_{I,:} v) \).

### 3 Concealing large-memory transfer latency between the host and device

To optimally conceal device-to-host transfer latency while maximizing host and device parallelism, it is necessary to:
(a) Add all dependent device-functions involving the data to be sent to an asynchronous device stream, \( s \),
(b) add the transfer of the data from device-to-host to \( s \),
(c) run independent host and/or device operations,
(d) synchronize \( s \) just prior to running a dependent operation on the host.
(e) Note that if the dependent data needed from the device on the host is a scalar, it may be returned without latency.

The other direction is slightly different. To optimally conceal host-to-device transfer latency while maximizing host and device parallelism, it is necessary to:

(a) launch the transfer on a device stream as soon as the data is available,
(b) add all dependent device-functions involving the data being sent to the device stream.

It is easy to see that algorithms with many sequential dependencies are at odds with these principles (they reveal transfer latency while minimizing host/device parallelism).

4 Optimization of TRON-LR Hessian-vector products for GPUs

We complete the total GPU-optimization of TRON-LR by considering the remaining bottleneck, the Hessian-vector product \( \nabla^2 f_{\text{LR}}(w)v = X + C X^T (D(v)) \). As with the previous optimizations, device variables are maximally decoupled from host-side dependencies, while using device-side functions which allow peak performance. In particular, we compute the diagonal matrix \( D \) in the same custom CUDA kernel used to compute \( \dot{z} \) (where \( D_{i,i} = h(y_i z_i)(1 - h(y_i z_i)) \)). \( D \) is also used in later host computations (for preconditioning [1]), so \( D \) is immediately transferred from device to host on an asynchronous device stream (the stream is synchronized just prior to host-variable use).

The candidate Newton step \( v \) (which is only of dimension \( n \)) is transferred from device to host on an asynchronous stream, and the following decompositions of \( \nabla^2 f_{\text{LR}}(w)v \) are added to this same stream: \( a_0 = Xv, a_1 = Da_0, a_2 = CX^T a_1 \). \( a_0 \) and \( a_2 \) are computed using cuSPARSE, while \( a_1 \) is computed using a custom kernel for element-wise multiplication along \( D \)’s diagonal. \( \nabla^2 f_{\text{LR}}(w)v \) is then transferred from host to device. However, the rest of the conjugate procedure is sequentially dependent on the dot-product \( v^T \nabla^2 f_{\text{LR}}(w)v \). In order to relieve this dependence while the \( \nabla^2 f_{\text{LR}}(w)v \) transfers from device to host, \( v^T \nabla^2 f_{\text{LR}}(w)v \) is computed on the device and the resulting scalar is available immediately to the host.

5 Summary of major TRON-LR-GPU operations

The following summarizes the major operations of the GPU-optimized TRON logistic regression solver, TRON-LR-GPU, as described in the main paper and herein. For each set of operations, the original lines from Algorithm 1 being optimized are listed in red.

- \( z = Xw \) is calculated and stored on the device (lines 2 and 5).
- The vectors \( \alpha, \dot{z} \) and diagonal matrix \( D \) are calculated on the device, such that \( \alpha = \log(1/h(y_i z_i)) \), \( \dot{z}_i = (h(y_i z_i) - 1)y_i \), and \( D_{i,i} = h(y_i z_i)(1 - h(y_i z_i)) \), where \( h(y_i z_i) = (1 + e^{-y_i z_i})^{-1} \) (lines 2 and 5). \( D \) is asynchronously transferred back to the host for future preconditioning computations.
- With \( \alpha \) in device memory, the objective \( f_{\text{LR}}(w) = \frac{1}{2} w^T w + C \sum_{i=1}^l \log(1 + \exp(-y_i z_i)) = \frac{1}{2} w^T w + C \sum_{i=1}^l \alpha \) is computed (lines 2 and 5).
- With \( \dot{z} \) in device memory, the gradient \( \nabla f_{\text{LR}}(w) = w + X^T \dot{z} \) is computed and transferred asynchronously back to the host (line 7).
- While all the above device-side quantities are being computed, the host runs independent, sequential operations concurrently, synchronizing the transfer streams for \( D \) and \( \nabla f_{\text{LR}}(w) \) just prior to host-side use (lines 7 and 4, respectively).
- The Hessian-product is computed on the device as \( \nabla^2 f_{\text{LR}}(w)v = v + C X^T (D(v)) \). Subsequently, the vector-Hessian-vector product \( v^T \nabla^2 f_{\text{LR}}(w)v \) is computed on the device and the resulting scalar is immediately available to the host (line 4).
6 Benchmark Dataset Statistics

<table>
<thead>
<tr>
<th>Dataset</th>
<th>#instances</th>
<th>#features</th>
<th>#nonzeros</th>
</tr>
</thead>
<tbody>
<tr>
<td>rcv1</td>
<td>20,242</td>
<td>47,236</td>
<td>1,498,952</td>
</tr>
<tr>
<td>SUSY</td>
<td>5,000,000</td>
<td>18</td>
<td>88,938,127</td>
</tr>
<tr>
<td>HIGGS</td>
<td>11,000,000</td>
<td>28</td>
<td>283,685,620</td>
</tr>
<tr>
<td>KDD2010-b</td>
<td>19,264,097</td>
<td>29,890,095</td>
<td>566,345,888</td>
</tr>
<tr>
<td>url</td>
<td>2,396,130</td>
<td>3,231,961</td>
<td>277,058,644</td>
</tr>
<tr>
<td>real-sim</td>
<td>72,309</td>
<td>20,958</td>
<td>3,709,083</td>
</tr>
<tr>
<td>Kim</td>
<td>23,330,311</td>
<td>18</td>
<td>419,945,598</td>
</tr>
<tr>
<td>Wilhelm</td>
<td>215,282,771</td>
<td>18</td>
<td>3,875,089,878</td>
</tr>
</tbody>
</table>

Table 2: Sparse and dense benchmark dataset statistics for TRON-LR and TRON-SVM, respectively.

References
