CHAPTER

Large-Scale Machine Learning 19

Jerod J. Weinman, Augustus Lidaka, Shitanshu Aggarwal

A typical machine-learning algorithm creates a classification function that inductively generalizes from training examples — input features and associated classification labels — to previously unseen examples requiring labels. Optimizing the prediction accuracy of the learned function for complex problems can require massive amounts of training data. This chapter describes a GPU-based implementation of a discriminative maximum entropy learning algorithm that can improve runtime on large datasets by a factor of over 200.

19.1 INTRODUCTION

Machine learning is used on a variety of problems, including time series prediction for financial forecasting [2], machine translation [1], character and speech recognition [8, 11], and even conservation biology [9]. Although there are many techniques for performing such classifications, the aforementioned approaches all use one type of model: the maximum entropy classifier, also known as multinomial logistic regression. The maximum entropy (hence, MaxEnt) technique builds a probability model that is consistent with training data, but is otherwise maximally noncommittal [6]. This property makes it theoretically ideal when training data is scant, but its superb performance when supplied with massive amounts of training data enhances its practical appeal as well. Such a probability model may then be used for making predictions about previously unseen data.

Train time for a MaxEnt classifier scales linearly with the size of the dataset. Because some effects can be seen only when training data is plentiful, running experiments with a MaxEnt classifier in the loop can be time-consuming. Fortunately, the training algorithm for MaxEnt can be parallelized, drastically reducing turnaround time for experiments. With this development, researchers can get results more quickly, and practitioners can deploy classifiers trained on much larger datasets.

Our motivation for employing MaxEnt has been to improve character recognition in arbitrary images of natural scenes [10, 11]. This task is more complex than typical document-based character recognition because it involves a wide variety of fonts and uncontrolled viewing conditions. As we demonstrate, gaining even a logarithmic performance improvement requires exponentially more data. Therefore, the training process should be parallelized to stay ahead of the computational demands of so much data.

GPU Computing Gems

19.2 CORE TECHNOLOGY

A maximum entropy model consists of an $L \times D$ matrix $\mathbf{W}$ that determines how much weight to assign each of D features in an input vector for each of L potential category labels. A $D \times 1$ column vector of input features $\mathbf{x}$ to be classified must simply be multiplied by the weight matrix. The label with the largest value in the product $\mathbf{Wx}$ can be taken as the assigned or predicted value. Thus, classification is a relatively efficient task. The goal of the learning algorithm is to find a weight matrix $\mathbf{W}$ that gives high values to correct labels and relatively low values to the rest for all input vectors.

The learning process involves optimizing a convex function of $\mathbf{W}$ for a fixed $D \times N$ matrix $\mathbf{X}$ of N training instances. The $L \times D$ matrix requires an optimization over many parameters. Though the optimization problem is convex, representing the second derivative of the objective function with the Hessian matrix is space prohibitive. Therefore, limited-memory quasi-Newton optimization algorithms (e.g., L-BFGS [3]) must be used to find the optimal weights. The bottleneck in the process is simply evaluating the objective function and its gradient when the number of training instances N is extremely large.

Fortunately, both the objective function and the gradient (detailed later in this chapter) are sums over the N training instances involving matrix products, vector sums, a vector max, and other straightforward computations. To parallelize the process, the data may be divided into smaller pieces whose contributions are repeatedly added to result accumulators.

The source code for the system described here is available under the GNU GPLv3 license.[1] A MATLAB implementation of a learning MaxEnt model is provided with the requisite CUDA kernels, as well as the optimization routines necessary for training.

19.2.1 Background: Maximum Entropy Theory

As its name suggests, the maximum entropy model is intimately related to probability theory. We now give details on the model in order to understand and implement the objective function being optimized. Let $\mathbf{X}$ be the $D \times N$ matrix of training data, as before, so that $\mathbf{x}_j$ is a column vector formed by the jth column of $\mathbf{X}$. Let $\mathbf{W}$ be the $L \times D$ weight matrix, with $\mathbf{w}_i$ the row vector formed by the ith row of $\mathbf{W}$. The probability of category label i given the data $\mathbf{x}_j$ is

$$p_{i,j} = \frac{1}{Z_j} \exp\left(\mathbf{w}_i \mathbf{x}_j\right). \tag{19.1}$$

The constant Z_j is the sum of the exponentiated inner products for all the category labels,

$$Z_j = \sum_{i=1}^{L} \exp\left(\mathbf{w}_i \mathbf{x}_j\right), \tag{19.2}$$

so that $\mathbf{p}_j$ represents a properly normalized probability distribution over labels with $\sum_{i=1}^{L} p_{i,j} = 1$.

[1] Available from http://www.cs.grinnell.edu/~weinman/code.

The training examples must have category labels assigned to the data. Let $\mathbf{y}$ be a $N \times 1$ column vector with entries $y_j \in \{1, \ldots, L\}$. With all of this, the objective function is the conditional log-likelihood of the labels given the observed data,

$$\begin{aligned}\mathcal{L}(\mathbf{W}; \mathbf{y}, \mathbf{X}) &\equiv \sum_{j=1}^{N} \log p_{y_j,j} \\ &= \sum_{j=1}^{N} \left(\mathbf{w}_{y_j} \mathbf{x}_j - \log Z_j \right), \end{aligned} \tag{19.3}$$

where $p_{y_j,j}$ indicates the probability of the correct label y_j for the jth instance and $\mathbf{w}_{y_j}$ is the vector of weights for the correct label of the instance. Typically, a regularization term is added to the function in order to prevent overfitting [4, 7]. We omit the details of this practice and point out where in the parallel implementation they are handled. For a particular label i and feature k, the gradient of the objective function, needed by the optimization routine, is given by

$$\begin{aligned} g_{i,k} &\equiv \frac{\partial}{\partial w_{i,k}} \mathcal{L}(\mathbf{W}; \mathbf{y}, \mathbf{X}) \\ &= \sum_{j=1}^{N} \left(\delta_{i,y_j} x_{k,j} - p_{i,j} x_{k,j} \right) \\ &= \sum_{j:y_j=i} x_{k,j} - \sum_{j=1}^{N} p_{i,j} x_{k,j}, \end{aligned} \tag{19.4}$$

where $\delta_{a,b}$ is the Kronecker delta.

19.2.2 Base Algorithm

Seeing the individual values used in the system, such as $p_{i,j}$ and $g_{i,k}$, is helpful for understanding the meaning of the computations. However, for economy many of these computations are calculated as matrix products. The inner products of Eq. 19.1 necessary to calculate the objective of Eq. 19.3 can be calculated at once with a matrix product. In addition, the gradient can also be calculated as two matrix products. In summary, the major computations involve

1. calculating a large matrix product $\mathbf{U} = \mathbf{WX}$;
2. log-normalizing the columns of $\mathbf{U}$ to create an $L \times N$ matrix of log probabilities, $\mathbf{L}$;
3. summing one entry from each column of $\mathbf{L}$ to find the objective function value and exponentiating the entire matrix to produce the probabilities $\mathbf{P}$; and then finally
4. calculating the gradient as a matrix product $\mathbf{G} = \left(\mathbf{XP}^\top\right)^\top$.

We note that the first term in the gradient Eq. 19.4 is a category-specific sum over the columns of $\mathbf{X}$. Much like the second term of the gradient can be implemented as a matrix product between the data $\mathbf{X}$ and the probabilities $\mathbf{P}$, the first term may be implemented as a sparse matrix product. Instead of $\mathbf{P}$,

Algorithm 1: Core value and gradient calculation for training a maximum entropy model in MATLAB.

```
function [V, G] = value(Y, X, W, G0)
% Calculate the raw energy (LxN)
U = W*X;
% Find the largest energy for each instance (1xN)
umax = max(U);
% Repeat for all labels (LxN)
umax = repmat(umax, [size(W,1) 1]);
% Log normalization (1xN)
logZ = umax + log(sum(exp(U-umax)));
% Repeat for all labels (LxN)
logZ = repmat(logZ, [size(W,1) 1]);
% Normalized log probability (LxN)
L = U - logZ;
% Indices in L of labels Y (1xN)
index = sub2ind(size(L), 1:length(Y), Y');
% Value: sum of log probabilities (1x1)
V = sum(L(index));
% Probabilities (LxN)
P = exp(L);
% Gradient (LxD)
G = G0 - (X*P')';
```

a sparse $L \times N$ matrix containing a 1 (one) at the appropriate row (the given label) for each column (instance) is used to calculate the first gradient term en masse. Because one factor of the matrix product is very sparse, the default MATLAB implementation for this calculation is sufficiently fast. Furthermore, this product needs to be calculated only once at the outset of training because it does not depend on the weights $\mathbf{W}$.

For numerical stability in calculating Z_j, the largest entry in the energy (for each instance) is found and subtracted before exponentiating in Eq. 19.2. All of these computations can be easily "vectorized" (i.e., written without any explicit loops) in MATLAB. An example implementation is shown in Algorithm 1. The first term of the gradient is precalculated and given as the input `G0`. In practice, to conserve memory the same variable can be used for `U`, `L`, and `P`, because they are the same size and are never needed simultaneously. We use separate variable names in this example to emphasize their connection to the mathematics outlined earlier in this chapter.

19.3 GPU ALGORITHM AND IMPLEMENTATION

Our program has been implemented as a back end to a MATLAB-based learning object. There are four calls to CUDA kernels, which are labeled `energy`, `probabilities`, `values`, and `gradient` in Figure 19.1. Each of these implements one of the four major computations listed in Section 19.2.2

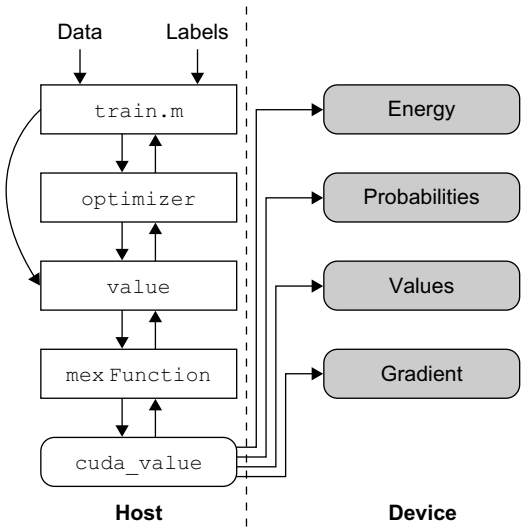

FIGURE 19.1

Overall program architecture. MATLAB routines and MEX functions are shown in square boxes, while CUDA functions are shown in rounded boxes. Shaded boxes indicate kernels that run exclusively on the CUDA device.

for a small piece of the training data. These kernels are called by another main procedure, `cuda_value`, that manages the division of the overall training data on the host system into pieces that are transferred to the CUDA device(s) for processing. The `cuda_value` wrapper procedure returns the total objective function value and gradient to a MATLAB `mexFunction`. In turn, the MEX gateway routine passes the result off to a MATLAB function `value` called by the overall optimization routine. It is the `optimizer` that uses the objective function value and gradient returned by `value` to calculate an updated weight matrix for subsequent evaluation, repeating the learning process as it maximizes the objective. After giving an overview of the communication patterns and module responsibilities, we explain each of these operations in more detail.

19.3.1 Training Overview

The training data $\mathbf{X}$ and category labels $\mathbf{y}$ are given to `train`, a MATLAB object method, along with any optional parameters relating to the learning process. Since $\mathbf{X}$ and $\mathbf{y}$ are fixed during the optimization process, these are cached in the `value` function, which calculates the objective function value and gradient when the `optimizer` passes in a weight matrix $\mathbf{W}$ for evaluation. The `value` function then forwards the request to a MEX gateway routine to be handled by our parallel CUDA implementation.

It is common to prevent overfitting by adding a regularization term for the weights to the objective function. This can easily be done in the `value` method. Because the regularization depends solely on $\mathbf{W}$ and a few other other higher-level parameters given to the training method, no parallelization is needed. The regularization term's value and gradient can be quickly calculated in `value` on the host system and added to the objective function value and gradient determined by the data.

19.3.2 Host-Device Interface

The interface between the host system and the CUDA device(s) is the `cuda_value` function. It is responsible for allocating memory on the device for the weights, the data, and intermediate results (i.e., the probabilities and the gradient). Host memory is much larger than the device memory, so we must partition the training data into smaller slices (several columns of $\mathbf{X}$), transferring each to the device and calculating their contributions to the objective function value and gradient one slice at a time. All four steps (energy through gradient calculation) must be completed on the device for the given slice of instances before the next slice is processed.

Because the value and gradient are sums over the N instances, this is a straightforward process of adding results to accumulator variables. The partial sum for the objective function value is calculated on the device and then accumulated on the host. The partial sums for the gradient matrix are accumulated on the CUDA device and copied back to the host when all the data has been processed. This is because the gradient is calculated using the CUBLAS library, which makes it easy to accumulate matrix products in parallel.

We assume that N will be the dominating term of the data over D and L. The number of labels L is unlikely to be extremely large because the number of instances N required to train an accurate classifier would need to be (nonlinearly) much larger. It is possible for the number of features D to be rather large, although overfitting may result. In some cases using fewer features can yield better prediction performance (e.g., [7]). In any case, we assume device memory will hold the full $L \times D$ weight and gradient matrices along with an intermediate $L \times N'$ probability matrix and the partial $D \times N'$ data matrix for several columns ($1 \ll N' < N$) of the full $D \times N$ data matrix $\mathbf{X}$.

For the most common ranges of L and D, the slice size N' will almost always exceed the number of independent streaming multiprocessors (SMs) on the device. Therefore, in the hopes of maximizing parallelism in the most common cases, we take the approach of processing the N' instances on the device in parallel, rather than maximally parallelizing the operations for processing each instance serially.

19.3.3 Calculating the Energy

The major computation of the MaxEnt model is the inner product between the weights and the data, called the energy. When viewed as matrices, these terms may easily be related via the matrix product $\mathbf{U} = \mathbf{WX}$. The weights and part of the data have been copied to the CUDA device, so the product $\mathbf{U}$, which is kept on the device, is easily calculated via a CUBLAS library call (`cublasSgemm`), the first exploitation of the device's parallelism. In order to calculate the objective function value, these raw energies must be converted into log probabilities, a normalization process that is performed in the next kernel function.

19.3.4 Normalizing Probabilities

Recall that the original data matrix has already been divided into slices, with one slice copied to the CUDA device. A column of the slice represents a training instance, each of which is assigned a block of threads for an SM. This partitioning of data is illustrated in Figure 19.2.

To calculate the probabilities used for the value and the gradient, the log-normalization term $\log Z_j$ must be calculated. This process requires three passes: finding a maximum, totaling exponentiated energies, and finally normalizing.

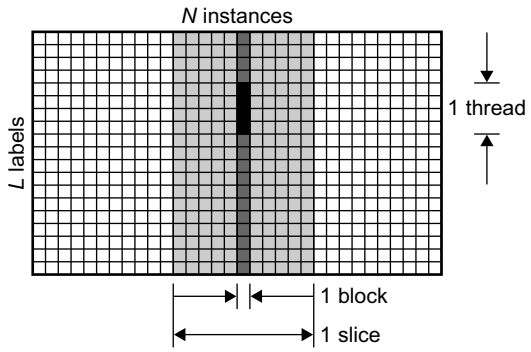

FIGURE 19.2

Partitioning of energy/probability matrix among blocks and threads.

```
__shared__ __device float mxarr[MAX_THREADS_PER_BLOCK];

...

float local_umx = P[start];

for (unsigned int i = start+1 ; i < end ; i++)
  local_umx = fmaxf(local_umx, P[i]);

mxarr[threadIdx.x] = local_umx;
```

FIGURE 19.3

Local maximum computation for a thread. The array `P` holds the instance's raw energies for the block, while `start` and `end` indicate the thread's partition bounds.

In the first pass, the the maximum value of the raw energy $u_{i,j}$ is found for all labels i of the instance j. This is ultimately to avoid exponentiating large values, which can give unreliable results, especially with the single precision floating-point numbers used in this implementation. We have tested two methods for finding the maximum `float` value in the vector. One makes use of comparisons and conditional assignment, while the other uses an order-preserving bijection between `unsigned int` and `float` types, so that comparisons can be made with the CUDA `atomicMax` function. First, we describe the simpler method and then an enhancement of it using the bijection and atomic functions.

Finding the Maximum Energy Without Atomics

We can use a simple butterfly reduction to find the maximum energy [5, 2.4.1]. An array is shared among the threads in the block. Because there may be more labels than threads, the labels are divided evenly among the block's threads, and each thread then finds the max among its partition of the data. This process is demonstrated in Figure 19.3.

```
unsigned int i;
for (i = powf(2,floorf(log2f(blockDim.x-1))); i > 0; i >>= 1)
{
  if (threadIdx.x < i && threadIdx.x + i < blockDim.x)
    if (mxarr[threadIdx.x] < mxarr[threadIdx.x + i])
      mxarr[threadIdx.x] = mxarr[threadIdx.x + i];

  _syncthreads();
}
float umax = mxarr[0];
```

FIGURE 19.4

Max-reduction butterfly code for a thread.

The butterfly reduction then proceeds by allocating one thread to two entries in the shared array and assigning one of the entries to the greater of the two. With the number of entries effectively halved, the process is repeated until the maximum entry is found.

One subtle detail that must be handled is when the number of threads is not a power of two. Let T be the number of threads, which is also the number of valid values in the butterfly. The value $2^{\lfloor \log_2(T-1) \rfloor}$ is thus the highest power of two strictly less than T. To ensure that no invalid data is accessed, we also check that the second index used belongs to a valid thread. This is demonstrated in Figure 19.4.

Finding the Maximum Using Atomics and Bijection

Rather than using conditional assignment, we may try using the CUDA `atomicMax` function to atomically assign the maximum value to one of the entries. This also performs assignment only as necessary. Because all threads in a butterfly must complete before advancing to the next level, a synchronization barrier is still required. However, the `atomicMax` function accepts only integral values. Fortunately, we can use an order-preserving bijection between `unsigned int`, a type supported by `atomicMax`, and `float`, the type of values in our vector.

As Herf reports,[2] the binary representation of a `float` number (regardless of denormalization) has two primary issues that prevent direct bitwise comparison, as would be done with `unsigned int`s. First and foremost, negative numbers use a leading sign bit, so that they are interpreted as larger than any positive number. Second, the signed-magnitude representation means the smaller (i.e., more negative) the number, the larger the bit interpretation. Because inverting the bits of both the exponent and mantissa reverses their ordering, we will want to do this for any negative number, but not positive numbers. To solve the first problem, we always flip the sign bit. The pair of methods for doing such a conversion in both directions is given in Figure 19.5.

With this conversion in hand, we can now re-tool our previous local max and butterfly algorithms. The only changes to Figure 19.3 are that the type of `mxarr` would be defined as `unsigned int`, and the final assignment to the array uses the forward bijection conversion:

```
mxarr[threadIdx.x] = floatFlip(local_umx);
```

[2] `http://stereopsis.com/radix.html` (accessed June 9, 2010).

```
_device_ unsigned int floatFlip(float theFloat)
{
  unsigned int mask = (_float_as_int(theFloat) >> 31) | 0x80000000;
  return _float_as_int(theFloat) ^ mask;
}

_device_ float invFloatFlip(unsigned int theUint)
{
  unsigned int mask = ((theUint >> 31) – 1) | 0x80000000;
  return_int_as_float((int)(theUint ^ mask));
}
```

FIGURE 19.5

A pair of routines for a bijection between `float` and `unsigned int` types. Note that the `__float_as_int_` and `__int_as_float` procedures are needed for doing bit-preserving conversions and that the leading (i.e., sign) bit is copied on a shift.

```
unsigned int i;
for (i = powf(2,floorf(log2f(blockDim.x–1))); i > 0; i >>= 1)
{
  if (threadIdx.x < i && threadIdx.x + i < blockDim.x)
    atomicMax(&(mxarr[threadIdx.x]), mxarr[threadIdx.x + i]);
  _syncthreads();
}
float umax = invFloatFlip(mxarr[0]);
```

FIGURE 19.6

Max-reduction butterfly code for a thread using CUDA atomics.

The butterfly reduction is changed to replace the conditional assignment with calls to `atomicMax`, and the final assignment of the max inverts the bijection, as shown in Figure 19.6.

While we have found only a very slight improvement in performance using atomics, we include this approach for completeness. If the design changed so that the partition spread an instance across multiple blocks, maximum values may need to be calculated in global memory, requiring greater use of atomics for `float` values.

Calculating the Normalizer: A Butterfly Sum Reduction

The second pass computes the sum for calculating Z_j, as in Eq. 19.2, subtracting the largest energy found in the previous pass before exponentiating. The same basic approach for calculating the maximum is used to calculate a sum involving the same data. Each thread computes a total for its partition, and then these subtotals are accumulated in a butterfly pattern to yield Z_j. We can finally compute the log of this sum and add back the maximum energy to give $\log Z_j$. This is demonstrated in Figure 19.7.

```
_shared_ _device_ float Z[MAX_THREADS_PER_BLOCK];

...

unsigned int i;

float local_Z = 0;

for (i = start; i < end; i++)
  local_Z += expf(P[i] – umax);

Z[threadIdx.x] = local_Z;

_syncthreads();

for (i = powf(2,floorf(log2f(blockDim.x–1))); i > 0; i >>= 1)
{
  if (threadIdx.x < i && threadIdx.x + i < blockDim.x)
    Z[threadIdx.x] += Z[threadIdx.x + i];

  _syncthreads();
}

float logZ = umax + logf(Z[0]);
```

FIGURE 19.7

Sum-reduction with thread-local computations and the butterfly for a thread. The array `P` holds the instance's raw energies for the block, while `start` and `end` indicate the thread's partition bounds. The maximum value `umax` has already been computed.

Calculating the Probability

The final pass uses the previous result to log-normalize the column of probabilities by subtracting $\log Z_j$ from every row (label) of the raw energy $u_{i,j}$. Because the raw energy is no longer needed, all of this computation is done in place to avoid extensive memory use. This helps to reduce the number of host-device data transfers because more space is available on the device for each slice.

In addition, we need to add the log probability of the correct label for each instance. Because there is only one for each block/instance, we allow a leader thread to make this calculation, which stores the result in a shared array that has one element for each block.

The final calculation is to normalize the log probability using the previously totalled log normalizer `logZ` (for $\log Z_j$ in Eq. 19.3) and exponentiate the result. This is done for every label assigned to the thread's partition, as shown in Figure 19.8.

19.3.5 Summing the Values

The third step is to total the value of the instances in the current slice. This is also a simple sum reduction. Because the number of instances in a slice is typically much smaller than the total memory on the device, this step is unlikely to be a bottleneck unless L and D are *very* small. Thus, we use a simple trick that is slightly suboptimal, but easy to implement. Because probabilities are between

```
for (unsigned int i = start ; i<end ; i++)
  P[i] = expf(P[i] – logZ);
```

FIGURE 19.8

Calculating the final normalized probability in one thread.

0 and 1, the log probability should be nonpositive. We can thus use the `cublasSasum` procedure to total the absolute values of the entries (which always flips the sign), and negate the result. The total for the slice is added to an accumulator on the host.

19.3.6 Calculating the Gradient

Like the first and third kernels, the final kernel is a straightforward CUBLAS call. To calculate the first term of the gradient Eq. 19.4, we use MATLAB's built-in sparse matrix multiplication routine with no significant speed loss. At each call to `cuda_value`, the gradient accumulator on the device is initialized to that pre-calculated matrix; subsequent gradient calculations simply add to it.

Because the number of features is typically much larger than the number of category labels ($D \gg L$), we avoid taking the transpose of the large slice of the data matrix $\mathbf{X}$ and instead prefer to transpose $\mathbf{P}$. Therefore, the product between the data and the probabilities is formatted as $\mathbf{G} = \left(\mathbf{X}\mathbf{P}^\top\right)^\top$, rather than the mathematically equivalent $\mathbf{G} = \mathbf{P}\mathbf{X}^\top$, to minimize data-processing overhead. Our implementation requires the outer transpose to occur only once on the host device after the gradients for all N instances have been accumulated. Despite this, there is a substantial performance improvement in avoiding the transpose of $\mathbf{X}$.

19.4 IMPROVEMENTS OF PERFORMANCE

To establish a baseline, we compare our CUDA-based GPU implementation of the training algorithm with a previous implementation using MATLAB (R2009a) matrix multiplications and other vectorized operations on a 64-bit platform with dual Intel Xeon Quad-Core 2.26 GHz CPUs and 48 GB of 1066 MHz DDR3 RAM. The parallel version is currently designed for a single GPU, but more could be utilized with relatively little effort. Our experiments are based on the same host system using an NVIDIA C1060 Tesla having 240 streaming processors (SPs) running at 1.3 GHz arranged in 30 streaming multiprocessors (SMs) with 4 GB DDR3 RAM.

MATLAB uses the extremely efficient LAPACK/BLAS routines, many of which are automatically multithreaded over a host system's available multicore processors (eight cores in this case). The vectorized operations $+$, $-$, `exp`, and `log` are also parallelized. Because many of our algorithm's most computationally intensive operations are parallelized, the comparison is reasonable. The notable exceptions are `sum` and `max`.

19.4.1 General Benchmarks

To measure the speedup of our parallel implementation, we use a variety of configurations of N, L, and D on random data, shown in Figure 19.9. We achieve a maximum speedup of $205\times$ when $N = 2^{15}$ and

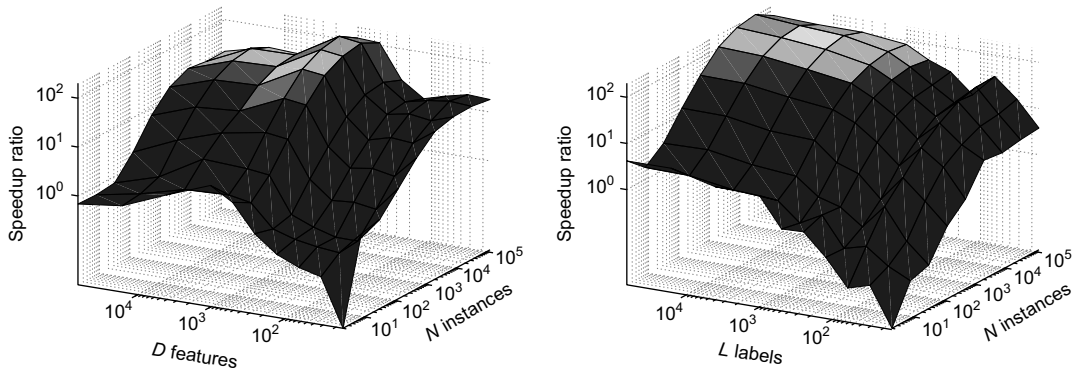

FIGURE 19.9

Speedup of GPU vs. CPU implementations when the number of labels is $L = 1024$ (left) and the number of features of the input is $D = 1024$ (right). The ratio of the average times over 10 runs are shown.

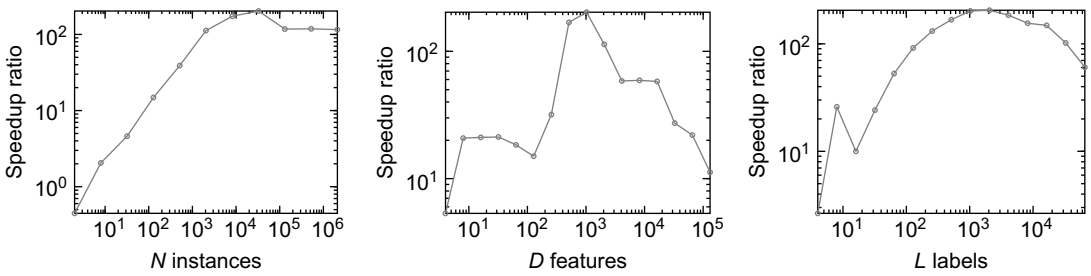

FIGURE 19.10

Saturation of improvement when the number of labels is $L = 1024$ and the number of features of the input is $D = 1024$ (left), $L = 1024$ and $N = 32768$ (center), and $D = 1024$ and $N = 32768$ (right).

$D = 2^{10}$ with $L = 2^{11}$. This simply measures the ratio of the time to run a single iteration of training (i.e., calculation of the objective function value and its gradient). There are likely to be more complex interactions between the optimizer and the amount of training data. We do not explore those interactions because they will depend heavily on the nature of the data, making it difficult to generalize. Our experience suggests that if additional training data requires more training iterations, the growth is more likely to be additive than multiplicative.

Additional experiments indicate that after the peak along the N axis there is a slight drop-off that then flattens out, as shown in Figure 19.10. There is a similar peak performance with respect to L, which gradually tapers. This is the dimension where our GPU implementation is more parallelized than MATLAB. The `sum` and `max` functions over the L labels in Algorithm 1 are not parallelized. Thus, we expect the parallelization of the kernel snippets in Figures 19.3, 19.6, and 19.7 to be most apparent when the matrix multiplication for the energy is not the dominant term, as shown in Figure 19.10.

After the peak along the D axis, performance drops more rapidly. This is because, for sufficiently large L, as the inner dimension of the matrix products grows (D in our case), the matrix multiplication

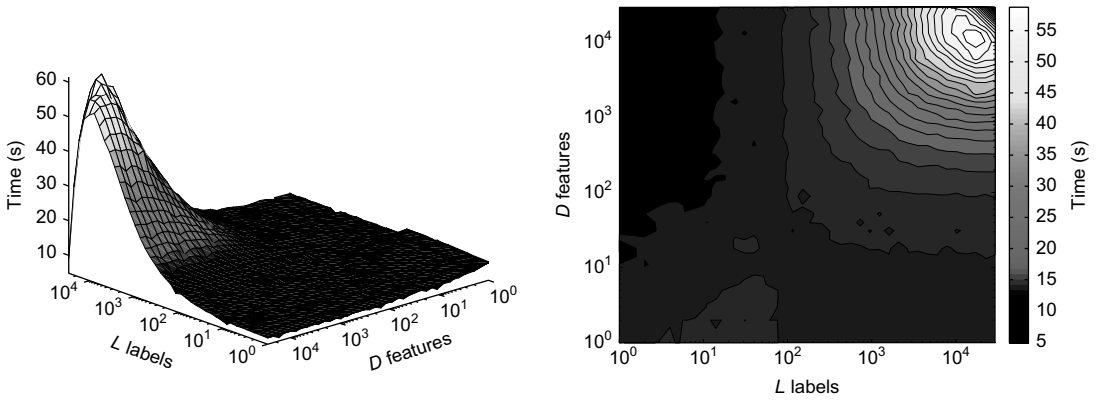

FIGURE 19.11

Two views of the time to calculate a matrix product between $L \times D$ and $D \times N$ matrices using CUBLAS SGEMM. The total amount of memory required is held constant so that N is determined by L and D.

(in CUBLAS) takes longer, even with a concomitant drop in the outer dimension N. We demonstrate this in Figure 19.11, where we measure the time to calculate the product of an $L \times D$ matrix and a $D \times N$ matrix using the CUBLAS SGEMM call. As we vary L and D, the remaining dimension N is set so that the matrix factors and product occupy as much of the device memory as possible. The product time increases with D until N is forced to be close to one, degenerating to a much faster operation.

19.4.2 Character Recognition Application

The parallel training algorithm is now a part of a larger character recognition system [10]. A character image is processed by a wavelet filter bank, whose responses undergo post-processing for invariance and numerical stability. In addition, a simple binarization algorithm is applied to the image. In this context, the number of features is $D = 32 \text{ pixels} \times 32 \text{ pixels} \times 25 \text{ images} = 25600 \approx 2^{14.64} \approx 10^{4.41}$. Our classifier must recognize 63 character categories (upper- and lower-case letters, digits, and space) in one of 7 quantized character widths, as well as a "null," non-text category. This gives rise to $L = 63 \times 7 + 1 = 442 \approx 2^{8.79} \approx 10^{2.65}$ categories. We use examples of characters from 1600 fonts rendered with various distortions, neighboring characters, and other contextual elements modeled on the intended deployed application environment. With 1600 base fonts, four distortions for each of the 62 characters and 3 spaces (of varying width), we have $N = 409600 \approx 2^{18.64} \approx 10^{5.61}$ training instances. Only N is determined by our system capability (host memory); L and D are driven by the application.

In these circumstances, our GPU implementation measures a sustained speedup of 30.6× over the MATLAB implementation (the ratio of average times over 5 runs). With these values of N, L, and D, the bottleneck of the computation is the energy calculation, a large matrix multiplication. Presumably, both the MATLAB and CUDA implementations of the BLAS function SGEMM are highly optimized. Despite the large difference in core processor speed (2.26 GHz on the host vs. 1.3 GHz GPU), the approximately 30× speedup here can largely be attributed to the use of 30 times as many cores for processing on the GPU (240 cores) over the host (eight cores).

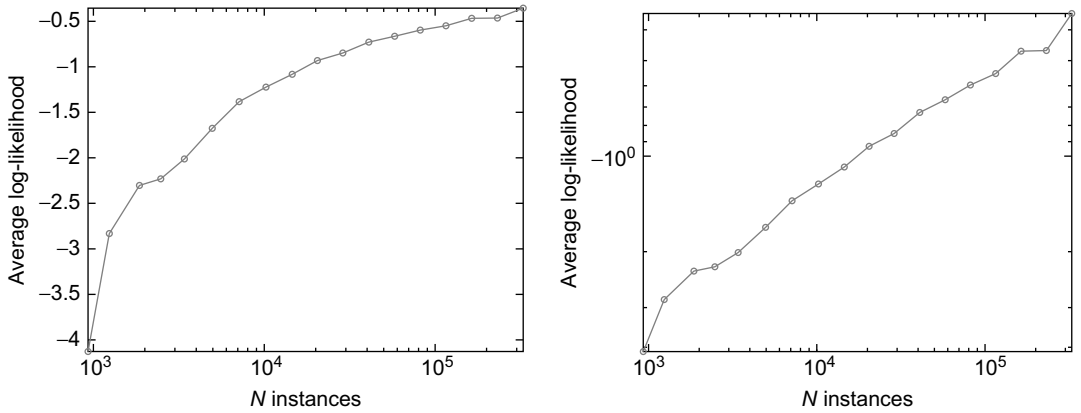

FIGURE 19.12

Performance of a character recognition model on a held-out dataset of over 100,000 characters. The size of the training set is gradually increased and the average log-likelihood of the validation data is measured. The right-hand figure is a log-log plot of the data on the left, which is plotted log-linear.

In real terms, the time drops from 56.6 minutes per iteration to 111 seconds per iteration. With just over 1000 iterations necessary to complete training with parameter validation, the CUDA implementation allows us to perform experiments in just over a day that would have taken more than a month otherwise.

The rapid turnaround time means that more data can be used to generate improved results. To measure the impact of increased training data in the character recognition domain, we used a simpler problem having $L = 62$ categories (only the digits and characters) with the same $D = 25600$ features. We assess the average log-likelihood of the model (the objective function) on a *different*, held-out dataset as N varies. Since the recognition model is used as part of a larger system where the relative probabilities of the various categories are important, the log-likelihood is a more useful metric than raw accuracy. Figure 19.12 shows a logarithmic improvement in log-likelihood as the training data increases in size exponentially.

19.5 CONCLUSIONS AND FUTURE WORK

Although MATLAB is highly optimized for the kinds of computations that must be performed in training a maximum entropy classifier, its host platform does not support the degree of parallelism necessary for learning from large datasets. We have therefore implemented such a learner that utilizes the parallelism of a GPU for the most common scenarios (large numbers of training instances and a moderate number of features). It is likely that in the cases where matrix multiplication is not the bottleneck, performance can be improved even further by optimizing the max and sum reductions at the heart of the other kernels. Unrolling loops and/or initializing extraneous values for non-power of two block sizes so that special cases can be eliminated should yield improved runtimes.

Acknowledgments

Augustus Lidaka and Shitanshu Aggarwal received support from Grinnell College and an HHMI Undergraduate Science Education Award for this work. The authors wish to thank NVIDIA corporation for a hardware grant, Michael Herf for the original float/unsigned integer bijection, Colin Sprinkle for his tutorial on atomics, Gernot Ziegler for early assistance with CUDA, and anonymous reviewers for suggestions that have improved this manuscript.

References

[1] A.L. Berger, S.A. Della Pietra, V.J. Della Pietra, A maximum entropy approach to natural language processing, Comput. Linguist. 22 (1) (1996) 39–71.

[2] P.W. Buchen, M. Kelly, The maximum entropy distribution of an asset inferred from option prices, J. Financ. Quant. Anal. 31 (1) (1996) 143–159.

[3] R.H. Byrd, J. Nocedal, R.B. Schnabel, Represenations of quasi-Newton matrices and their use in limited memory methods, Math. Program. 63 (1994) 129–156.

[4] S. Chen, R. Rosenfeld, A Gaussian Prior for Smoothing Maximum Entropy Models, Technical Report CMU-CS-99-108, Carnegie Mellon University, 1999.

[5] I.T. Foster, Designing and Building Parallel Programs, Addison-Wesley, Reading, MA, 1995.

[6] E.T. Jaynes, Information theory and statistical mechanics, Phys. Rev. 106 (4) (1957) 620–630.

[7] B. Krishnapuram, L. Carin, M.A.T. Figueiredo, A.J. Hartemink, Sparse multinomial logistic regression: fast algorithms and generalization bounds, IEEE Trans. Pattern Anal. Mach. Intell. 27 (6) (2005) 957–968.

[8] J.K. Hong-Kwang, Y. Gao, Maximum entropy direct models for speech recognition, IEEE Trans. Audio. Speech. Lang. Processing 14 (3) (2006) 873–881.

[9] S.J. Phillips, M. Dudík, R.E. Schapire, A maximum entropy approach to species distribution modeling, in: Proceedings International Conference on Machine Learning, Banff, A., Canada, July 4–8, 2004, New York: Association for Computing Machinery, p. 83.

[10] J.J. Weinman, E. Learned-Miller, A. Hanson, A discriminative semi-Markov model for robust scene text recognition, in: 19th International Conference on Pattern Recognition, Tampa, Florida, 2008, IEEE Computer Society, pp. 1–5.

[11] J.J. Weinman, E. Learned-Miller, A. Hanson, Scene text recognition using similarity and a lexicon with sparse belief propagation, IEEE Trans. Pattern Anal. Mach. Intell. 31 (10) (2009) 1733–1746.