Efficient GPU implementation of convolutional neural networks for speech recognition

Ewout van den Berg\textsuperscript{1}, Daniel Brand\textsuperscript{2}, Rajesh Bordawekar\textsuperscript{2}, Leonid Rachevsky\textsuperscript{1}, Bhuvana Ramabhadran\textsuperscript{1}

\textsuperscript{1}IBM Watson
\textsuperscript{2}IBM T.J. Watson Research Center

\{evandenberg,danbrand,bordaw,rachevs,bhuvana\}@us.ibm.com

1. Introduction

Over the past couple of years the use of neural networks in acoustic models has models has developed standard practice, with convolutional neural networks (CNNs) [1, 2, 3] being on of the most widely used network types. The training of CNNs remains a computationally intensive task, however, in part because of their expensive convolution layers. Improvements in the performance enables the training of larger models, the use of more training data, an increase in the number of training iterations, or simply the training of a larger number of different network configurations; each of which can help improve the overall model accuracy. In this paper we investigate different approaches for the implementation of the convolution operations on GPU hardware. We compare our native implementation with those based on the NVIDIA cuDNN [4, 5] and cuFFT [6] libraries, and compare the end-to-end performance on a 3-hour transcribed Lithuanian data set from the Babel very limited language pack.

2. Background

The convolution layers extensively use convolution and cross-correlation operations. Here we give the exact definitions used in the implementation. Given an $m_1 \times n_1$ matrix $A$ and an $m_2 \times n_2$ matrix $B$, the convolution $M = A \odot B$ gives matrix $M$ of size $(m_1 + m_2 - 1) \times (n_1 + n_2 - 1)$, with entries

$$M[i, j] = \sum_{s=1}^{m_2} \sum_{t=1}^{n_2} A[i - m_1 + s, j - n_1 + t] \cdot B[s, t], \quad (1)$$

with the convention that $A[i, j] = 0$ for $(i, j) \notin [1, m_1] \times [1, n_1]$. A restricted version of the convolution excludes the first and last $m_2 - 1$ rows and $n_2 - 1$ columns and is denoted by $A \circ B$. The cross-correlation operator $M = A \circ B$ outputs a matrix of size $(m_1 - m_2 + 1) \times (n_1 - n_2 + 1)$ with entries

$$M[i, j] = \sum_{s=1}^{m_2} \sum_{t=1}^{n_2} A[i + s - 1, j + t - 1] \cdot B[s, t]. \quad (2)$$

We extend the convolution to three-dimensional tensors $A$ and $B$ with equal depth $c$ (number of feature maps) as

$$A \circ B = \sum_{i=1}^{c} A[\ldots, i] \circ B[i, \ldots, \ldots], \quad (2)$$

and likewise for cross-correlation. Using this notation we can define the forward operator in each CNN layer as the cross-correlation of the three-dimensional input tensor $X$ with filters $F_1, \ldots, F_K$, resulting in an output tensor $Y$, as follows:

$$Y[\ldots, k] = X \circ F_k, \quad k = 1, \ldots, K. \quad (3)$$

An illustration of this is shown in Figure 1.

Given a loss function $f$ we can compute the gradient with respect to each of the filters by means of the chain rule\textsuperscript{1}:

$$\frac{\partial f}{\partial F_k} = \frac{\partial Y}{\partial F_k} \cdot \frac{\partial f}{\partial Y} \quad (4)$$

\textsuperscript{1}By redefining the partial derivative in transpose form, we can reverse the order of the chain rule.
We now give a brief sketch on how this is done on two-dimensional input with a single filter. The extension of this to the tensor case with multiple filters is relatively straightforward, however the necessary notation would overly complicate exposition and we therefore omit the details.

For the restricted convolution $A \circ B$, denote the discrete 2D Fourier transformation by the complex matrix $F$, and let $a$ and $b$ be vectorized versions of $A$ and $B$, then

$$A \circ B = R F^{-1} \left( \text{diag}(Fa) \cdot (F \cdot Ec) \right),$$

where $R$ and $E$ are appropriate linear restriction and embedding operators, respectively. In case there are several filters or input matrices (batches in neural network training) we need to compute the Fourier transformation of each of the filters and input matrices once, then compute the elementwise product and compute an inverse Fourier transform for each of the outputs. For the tensor case all we need to do is take the elementwise product between pairs of matrices and summing these up before applying the inverse Fourier transformation. Without going into details we mention that the error backpropagation and gradient with respect to the filter coefficients can be derived by repeated use of the chain rule.

### 3. Algorithms

There are several ways in which we can compute the three basic operations: the forward operator (3), the backward error propagation (5), and the gradient with respect to the filters (4). In this section we discuss two different approaches.

#### 3.1. Expansion of the input matrix

As shown in Figure 1, we can evaluate the cross-correlation by shifting a window over the input data and computing at each location the inner-product between the filter and the input entries within the window. This is then repeated for each of the filters to obtain the output tensor. For each filter we perform a complete swipe over the input data, thereby repeatedly accessing the same blocks of input data. One way to improve the memory access pattern is to swap the window over the input data once, and at each location vectorize the corresponding entries of the input and append these to a matrix, say $D$. The cross-correlation can then be represented as a matrix-matrix product $FD$, where $F$ is a matrix whose rows contain the vectorized filter tensors $F_k$. Denoting by $\mathcal{L}$ the linear operator that expands the input tensor $X$ to matrix $D$, we have $Y = F \cdot \mathcal{L}(X)$, which matches (3) after reordering and reshaping $Y$ or by simply defining the appropriate strides in memory to obtain a tensor representation.

From this formulation, we easily find that

$$\frac{\partial f}{\partial F} = Z[\mathcal{L}(X)]^T, \quad \text{and} \quad \frac{\partial f}{\partial X} = \mathcal{L}^*(F^TZ),$$

where $Z = \partial f/\partial Y$, and $\mathcal{L}^*$ is the adjoint of $\mathcal{L}$. The adjoint can be computed by initializing an empty tensor $B$ with the same dimension as $X$ and swapping a window across this tensor just like in the cross-correlation step. Instead of copying each of the entries in the window in $B$ we now add the scalar entry in $Z$ corresponding to the window location to each of these entries and repeat this for each of the filters.

#### 3.2. Fourier-based approach

It is well known that convolution can be done efficiently in the frequency domain [7]. We now give a brief sketch on how this is done on two-dimensional input with a single filter. The extension of this to the tensor case with multiple filters is relatively straightforward, however the necessary notation would overly complicate exposition and we therefore omit the details.

For the restricted convolution $A \circ B$, denote the discrete 2D Fourier transformation by the complex matrix $F$, and let $a$ and $b$ be vectorized versions of $A$ and $B$, then

$$A \circ B = R F^{-1} \left( \text{diag}(Fa) \cdot (F \cdot Ec) \right),$$

where $R$ and $E$ are appropriate linear restriction and embedding operators, respectively. In case there are several filters or input matrices (batches in neural network training) we need to compute the Fourier transformation of each of the filters and input matrices once, then compute the elementwise product and compute an inverse Fourier transform for each of the outputs. For the tensor case all we need to do is take the elementwise product between pairs of matrices and summing these up before applying the inverse Fourier transformation. Without going into details we mention that the error backpropagation and gradient with respect to the filter coefficients can be derived by repeated use of the chain rule.

### 4. Implementation

For the discussion of the implementation it will be convenient to first introduce some terminology used by the cuDNN library. The main data structure used is a four-dimensional tensor containing $N$ batches, each consisting of $C$ feature maps of height $H$ and width $W$. Tensors consist of a pointer to device memory along with a descriptor containing the dimensions and the strides between neighboring elements along these dimensions. The general memory layout can be abbreviated by ordering the strides from large to small; a typical ordering is NCHW.

#### 4.1. Native implementation

The native implementation in the IBM Attila speech recognition toolkit [8] uses the expansion approach discussed in Section 3.1 with input of the form NHCW. Given a filter of size $f_h \times f_w \times C$, there will be $p_h = H - f_h + 1$ window positions in the vertical direction and $p_w = W - f_w + 1$ in the horizontal direction. The output of the localizer is therefore a matrix with $N \cdot p_h \cdot p_w$ columns and $C \cdot f_h \cdot f_w$ rows. For the GPU implementation we assign to each thread the expansion of a single two-dimensional window of size $f_h \times f_w$ for a given feature map and window location. Each block contains the threads for all positions and as many feature maps as possible. After localization we perform a matrix-matrix product with the filter matrix $F$ using the cuBLAS gemm function [9]. A bias term is then added to the result, followed by the application of a (sigmoid) nonlinearity. The resulting tensor has an NHWC ordering, and the role of the subsequent pooling operator is therefore two-fold: first it applies max-pooling to shifting windows over the HW dimension, and second it ensures that the output is in the NHWC format expected by the next layer.

#### 4.2. Implementation using cuDNN

The cuDNN library provides functions for convolution, bias, activation, and pooling. Convolution is also implemented based on the matrix product reformulation, with the difference that cuDNN performs on-the-fly localization while doing the matrix multiplication. The benefit of the dynamic approach is a much reduced memory footprint. On the other hand, in our implementation we can reuse the localized matrix for the gradient...
computation, which helps improve performance. The backward and gradient operations in cuDNN require input in the NCHW format, which means that the output of the forward operation is restricted to the same format.

4.3. Alternative cuDNN implementation

As we will see in the next section, the backward and gradient operations provided by cuDNN turn out to be unexpectedly slow on our particular convolution settings. The forward convolution operator does have a good performance, and it therefore makes sense to use the expressions for the gradient (4) and back-propagation of error (5) and implement these using the forward convolution function of cuDNN instead.

The forward convolution and cross-correlation operators in cuDNN completely decouple along the N-dimension (number of batches). For each batch, the convolution operation is equal to (2), and similar for cross-correlation. The gradient with respect to the filter coefficients is given by the cross-correlation of the input data with the back-propagated error signal (4). Unlike in the regular forward pass, we now need to sum over the contributions of each of the batches while retaining separate information for each of the feature maps. This can be achieved by exchanging the roles of N and C in the error data. Ideally this should be as simple as exchanging the dimensions and stride information, but unfortunately cuDNN only accepts filters in contiguous NCHW format, which means that we first need to copy the data in transposed format before applying the forward cross-correlation operator.

The implementation for the backward error propagation is slightly more complicated due to the fact that cuDNN does not implement the full convolution (1) but only the restricted version. In order to implement the full convolution $A \ast B$ with matrices $A$ and $B$ defined in Section 2 we first need to pad $A$ with $m_2 - 1$ rows of zeros at the top and bottom and $n_2 - 1$ columns on the left and right, resulting in a matrix of size $(m_1 + 2(m_2 - 1)) \times (n_1 + 2(n_2 - 1))$. This operation is best done by creating a zero matrix of desired size and then embedding matrix $A$. Just like for the gradient computation, we need to transpose the tensor by exchanging the C and N dimensions. Both operations can be done at once by setting up two tensor descriptors with appropriate strides and performing a data transformation with an offset added to the output data pointer. After this we form a new descriptor that simply has the N and C dimensions and strides exchanged and takes into account the increase H and W dimensions (this step is logical only and does not require any additional manipulations of the underlying data). We then apply the forward operator for (restricted) convolution to obtain the desired result.

4.4. Implementation using cuFFT

The two main parts of the frequency-based implementation of the convolution block are the transformation between time and frequency space, and the pairwise multiplication and addition of the coefficients. We further require embedding and restriction (6), both of which can be done by setting up appropriate descriptors and using the tensor transformation provided by cuDNN. The Fourier transformations in the cuFFT library are done by first generating a transformation plan which describes the fastest transformation based on the dimensions of the data, their strides, as well as the number of individual transforms and the stride in memory between successive input matrices. Once a plan has been generated it can be executed as many times as needed.

### Table 1: Description of the first two convolution layers and subsequent linear layers.

<table>
<thead>
<tr>
<th>Layer</th>
<th>Stage</th>
<th>N</th>
<th>C</th>
<th>H</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>#1</td>
<td>Input</td>
<td>256</td>
<td>3</td>
<td>11</td>
<td>40</td>
</tr>
<tr>
<td></td>
<td>Filter layer</td>
<td>128</td>
<td>3</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>Convolution</td>
<td>256</td>
<td>128</td>
<td>3</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td>Pooling</td>
<td>256</td>
<td>128</td>
<td>3</td>
<td>11</td>
</tr>
<tr>
<td>#2</td>
<td>Input</td>
<td>256</td>
<td>128</td>
<td>3</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>Filter layer</td>
<td>255</td>
<td>128</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>Convolution</td>
<td>256</td>
<td>256</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>Pooling</td>
<td>256</td>
<td>256</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>#3</td>
<td>Input</td>
<td>256</td>
<td>–</td>
<td>1</td>
<td>2048</td>
</tr>
<tr>
<td>#4/#5</td>
<td>Output/input</td>
<td>256</td>
<td>–</td>
<td>1</td>
<td>1024</td>
</tr>
<tr>
<td>#5</td>
<td>Output</td>
<td>256</td>
<td>–</td>
<td>1</td>
<td>1000</td>
</tr>
</tbody>
</table>

5. Experiments

For our experiments we use a CNN with two convolution layers and three linear layers, each with a sigmoid non-linearity, except for the output layer, which has a softmax output. The network parameters are summarized in Table 1. The input to the first layer comes in batches of size 256 (this remains constant for all subsequent layers) with 40-dimensional log-MEL features (width) and 11 frames of context (height) along with $\Delta$ and $\Delta\Delta$ (giving a total of three feature maps). For filter descriptors, the number of filters N (denoted by K in Section 2) is equal to the number of feature maps C in the output; for the first layer we use 128 filters. The output of the convolution has height $11 \times 9 + 1 = 3$ and width $40 \times 9 + 1 = 32$. The bias and activation have the same output size as the convolution. For the max-pooling we use a window size of $1 \times 3$ with shift 1 in the vertical direction and 3 in the horizontal direction. When the overlap between the window and the input data is more than half the window size (both horizontally and vertically) we pool over all valid entries within the window. For our first layer this means that for the input width of 32 we have 11 pools in the horizontal direction. In cuDNN such overlap is not supported and we instead apply zero padding in the output of the activation function by changing the horizontal stride from 32 to 33. The only caveat of this approach is that the explicit zero can mean selection of this value as the maximum. For the sigmoid activation function alternative padding values could be used. For the second layer we have convolution with 256 filters each of size $3 \times 4$. In our native implementation we apply max pooling with pool size $1 \times 1$ to convert the data to NCHW format. For the cuDNN implementation the data in the second layer are already in the correct format after activation so we can omit pooling. The output of the second layer has 256 feature maps of size $1 \times 8$. For the third layer we reinterpret this as feature vectors of size 2048. Subsequent layers transform this to size 1024, 1024, and finally to 1000 output HMM states.

5.1. Stand-alone experiment

Our first experiment considers the forward, backward, and gradient operations of the convolution block. Aside from our native implementation we implemented convolution classes based on the cuDNN library (versions 1 and 2 with implicit expansion) and the alternative approach using cuDNN forward convolution for each of the three operations. The convolution uses NCHW input and generates output in the NHWC format for the native
For our end-to-end evaluation we use the full network given in Table 1 to train a 3-hour Lithuanian system using stochastic gradient descent on a cross-entropy loss function for 30 epochs.

Table 2: Average runtime in milliseconds of the forward, backward, and gradient operations of the convolution block on the K40 using different methods. The version numbers refer to the cuDNN library used.

<table>
<thead>
<tr>
<th>Method</th>
<th>Fwd</th>
<th>Bwd</th>
<th>Grad</th>
<th>Fwd</th>
<th>Bwd</th>
<th>Grad</th>
</tr>
</thead>
<tbody>
<tr>
<td>Native v.1</td>
<td>1.79</td>
<td>2.09</td>
<td>3.42</td>
<td>2.16</td>
<td>2.38</td>
<td>1.93</td>
</tr>
<tr>
<td>Implicit v.1</td>
<td>2.61</td>
<td>3.00</td>
<td>3.06</td>
<td>4.34</td>
<td>10.71</td>
<td>22.97</td>
</tr>
<tr>
<td>Implicit v.2</td>
<td>2.12</td>
<td>2.49</td>
<td>2.84</td>
<td>3.11</td>
<td>19.09</td>
<td>22.87</td>
</tr>
<tr>
<td>Precomp. v.2</td>
<td>1.61</td>
<td>2.50</td>
<td>2.84</td>
<td>2.05</td>
<td>19.91</td>
<td>22.87</td>
</tr>
<tr>
<td>Explicit v.2</td>
<td>2.11</td>
<td>2.49</td>
<td>2.84</td>
<td>2.18</td>
<td>19.90</td>
<td>22.87</td>
</tr>
<tr>
<td>Alternative v.2</td>
<td>1.61</td>
<td>9.43</td>
<td>7.26</td>
<td>2.02</td>
<td>4.76</td>
<td>1.98</td>
</tr>
</tbody>
</table>

The performance of the different methods on the first layer convolution are similar except for the alternative implementation, which is notably slower. The fastest forward operator is cuDNN version 2 with precomputed indices. The backward and gradient operators in version 2 outperform those in version 1. The native implementation has a slight edge for the forward operator but is somewhat slower on the forward and gradient operations.

The performance for the second layer is an altogether different story. From Table 2 we see that cuDNN performs very poorly on the backward and gradient operations for this particular convolution setting. On the three operations combined, our native implementation is 6.9 times faster than the best cuDNN method. Our alternative approach, which was inspired by the discrepancy in performance between the forward and backward steps, performs well despite the overhead of the tensor transposition and embedding.

For the frequency-based implementation of the convolution block we performed a series of experiments using cuFFT. For the first layer we need to apply forward Fourier transformation on 256 × 3 input data blocks of size 11 × 40, and 128 × 3 filter blocks of the same size after embedding the 9 × 9 filter coefficients. The output requires the inverse Fourier transformation on a grid of 256 × 128 blocks, again of size 11 × 40. To evaluate the backward and gradient operations we need to compute each of these transformations in reverse order exactly once. The 256 × 128 grid of matrices can be transformed as 256 batches of size 128; 128 batches of size 256; or as a single batch of 32768 elements (provided that the tensor is in NCHW format). The fastest of these, the single-batch option, took 9.09 ms for the forward transform and 9.66 ms for the inverse. When using the native NHCW format these times increased to at least 14.63 ms and 16.01 ms, respectively. Given that 11 is a prime, the natural question to ask is whether using a height of 12 or 16, or a width of 48 or 64 would yield better result. In general we found a larger runtime for larger matrices, despite the more favorable factorization of these sizes. Given the results we already had with the native and alternative cuDNN implementation, we decided that the cuFFT based approach was not worth further consideration for the given problem sizes. Recent work has shown, however, that Fourier-based convolutions can be highly efficient on a range of other problem sizes [10].

5.2. End-to-end training task

We have provided a detailed comparison between our native CNN implementation and those based on the cuDNN and cuFFT libraries. On both the K10 and K40 we found that our native implementation outperforms all cuDNN-based methods. Based on the performance of the Fourier transformations alone, we concluded that cuFFT is not a viable alternative for our network configuration. We are currently working to further improve the performance of our system by utilizing streams and overlaying the data ingestion on the CPU with computations on the GPU, which reduces the overhead time. Preliminary results show that end-to-end training time on our given network can be reduced further by at least a factor of two, thereby making the improvements in convolution performance even more prominent.

6. Conclusions

We have provided a detailed comparison between our native CNN implementation and those based on the cuDNN and cuFFT libraries. On both the K10 and K40 we found that our native implementation outperforms all cuDNN-based methods. Based on the performance of the Fourier transformations alone, we concluded that cuFFT is not a viable alternative for our network configuration. We are currently working to further improve the performance of our system by utilizing streams and overlaying the data ingestion on the CPU with computations on the GPU, which reduces the overhead time. Preliminary results show that end-to-end training time on our given network can be reduced further by at least a factor of two, thereby making the improvements in convolution performance even more prominent.

7. References


