Binarized Convolutional Neural Networks for Efficient Inference on GPUs

Mir Khan, Heikki Huttunen, Jani Boutellier
Tampere University of Technology
Tampere, Finland
Email: Mir.Khan@tut.fi, Heikki.Huttunen@tut.fi, Jani.Boutellier@tut.fi

Abstract—Convolutional neural networks have recently achieved significant breakthroughs in various image classification tasks. However, they are computationally expensive, which can make their feasible implementation on embedded and low-power devices difficult. In this paper convolutional neural network binarization is implemented on GPU-based platforms for real-time inference on resource constrained devices. In binarized networks, all weights and intermediate computations between layers are quantized to +1 and -1, allowing multiplications and additions to be replaced with bit-wise operations between 32-bit words. This representation completely eliminates the need for floating point multiplications and additions and decreases both the computational load and the memory footprint compared to a full-precision network implemented in floating point, making it well-suited for resource-constrained environments. We compare the performance of our implementation with an equivalent floating point implementation on one desktop and two embedded GPU platforms. Our implementation achieves a maximum speed up of $7.4\times$ with only 4.4% loss in accuracy compared to a reference implementation.

Keywords: model compression, binarized convolutional neural networks, optimization, image classification

1. Introduction

In the recent years, convolutional neural networks (CNNs) have presented impressive performance in image classification [16][4], face recognition [17][19], audio classification [14], and speech recognition [7]."

Large neural network models can be computationally expensive, making them unsuitable for deployment to small resource-constrained mobile devices. To this extent, contemporary CNN-based solutions often acquire the input data on a mobile device, but transmit the data to a remote server for CNN-based processing. However, performing the CNN-based processing on the mobile device (a.k.a. edge computing) would reduce the overall system complexity and enable real-time applications.

The emerging CNN subfield of *model compression* aims to retain the accuracy of the neural network while minimizing redundant network parameters and reducing computational load. Many such techniques have already been proposed.

One technique [9] is based on *pruning* of parameters, where majority of the parameters of the network are removed without significantly impacting accuracy. Reduction of parameters initially leads to a significant drop in accuracy; however, retraining (fine-tuning) of the parameters restores most of the network's accuracy. The authors report $13\times$ reduction of memory requirements with no loss in accuracy [9].

Another approach, low-rank approximation of convolutional kernels [13], approximates 2D convolutions with convolutions by vectors. The separable kernels can be obtained either by training the network with separable filters [1] or by posing it as an optimization problem to minimize the reconstruction error of the feature maps. Depending on the approach [13][1], speedups between $2\times$ to $4\times$ have been reported on CPU implementations.

Binarized neural networks (BNN) have been first introduced in [11], where their performance was demonstrated on the CIFAR-10 dataset. The weights and activations for intermediate computations are binarized to $+1$ and -1. The authors present a speed up of $7\times$ on a network for the MNIST dataset. In a further work [21] the approach was refined for CPU implementation and evaluated on the ImageNet dataset.

In this paper, an approach for the implementation of binarized convolutional neural networks (BCNNs) on GPU platforms is presented. A recent work (to appear in ICLR 2018) [20] presents similar BCNN optimizations on GPUs. Our work was developed simultaneously with [20] but differs in several respects. We provide a more detailed (layer-wise) performance comparison with state of the art full precision GPU implementations such as cuDNN, cuBLAS, and the ARM Compute libraries. Also, instead of concentrating only on the desktop GPU, we test and evaluate the performance of our GPU-optimized BCNNs (written in both CUDA and OpenCL) on several different platforms, including embedded ones. Additionally, we present our implementation with an application use case of vehicle type classification [12]. Finally, we employ alternative input binarization strategies and study the impact of each approach on the classification accuracy. Our results show significant speedups in real-time inference compared to a floating point version of an equivalent neural network.

As a summary, the contributions of this work are as follows:

- Detailed presentation of efficiently implementing CNN binarization, including the convolutional layers, on GPU-based platforms.
- Study and comparison of different approaches for binarizing input data, and how each approach impacts the classification accuracy.
- Performance (execution time) measurements and comparisons on several platforms against state of the art full-precision GPU implementations.

The source code for our CUDA implementation is publically available [1].

2. Experimental Setup

2.1. Binarizing the network

Our binarized network architecture is based on the original vehicle classifier network presented in [12]. We implement a binarized version of the same architecture in several steps. We do not use any ReLU [6] activations in the binarized version. In the original binarization work [5], the authors suggest two approaches for binarization: stochastic and deterministic. For binarizing the weights and intermediate computations, we use the deterministic *sign* function, which is defined as

$$sign(x) = \begin{cases} -1 & \text{if } x \leq 0 \\ +1 & \text{if } x > 0 \end{cases} \qquad (1)$$

For training the BNN, following [10], we explicitly define the gradient of the *sign* function to be the identity function in the backward pass, such that $\frac{\partial sign(x)}{\partial x} = x$.

The non-binarized network is trained with the RMSprop optimizer [23], while the binarized version is trained with the ADAM [15] optimizer. After training, only the binarized weights are used for inference for the binarized network.

The network is trained with a dataset set consisting of 6555 images of vehicles that have been captured by a camera and manually categorized into four different classes: *bus*, *normal*, *truck*, and *van*. Each image has size 96×96 and are in full color. The data has been split into a training set (90%) and a test set (10%). We augment the training set using flipping and filtering with a 2D Gaussian filter with $\sigma = 0.5$, resulting in a total training set size of 14,108 images, 20% of which are used for validation. Throughout this text, our accuracy reports are for the performance of the network on the test set that corresponds to the best validation set accuracy.

2.2. Testing pipeline

For obtaining runtime results, we use the built-in GPU timers to measure the runtime of the kernels for our CUDA

1. github.com/Valentin4869/BinCNN

and OpenCL programs. Our kernel execution time measurements do not include memory transfer times to/from the GPU, as they can be affected by various factors, some of which are hardware-dependent, for example, on the NVidia Jetson host and device memory are shared. The correctness and accuracy of the profiling results generated have been verified by the Nvidia Visual Profiler for the same CUDA programs.

For each test run, 1000 images are randomly generated and fed to the network one at a time. The timer begins after the memory is copied, and the timer ends after the last kernel's computation is completed. Our final result is the total accumulated time per sample averaged over all 1000 samples.

2.3. Input binarization

In this section we describe our methods for binarizing the inputs to the first layer of our BNN. We pre-process the data set using these techniques and evaluate the accuracy of the BNN on the pre-processed data set.

Thresholding A constant threshold T can be subtracted from the input $\mathbf{X}$ before binarizing it. We simply substitute the input $\mathbf{X}$ to the first layer with $sign(\mathbf{X} + T)$, for $\mathbf{X} \in \mathbb{R}^{H \times W \times C}$, and for $T \in \mathbb{R}^{1 \times 1 \times C}$. The motivation is to shift the range of values taken by $\mathbf{X}$ such that binarization with the *sign* function produces meaningful results, as opposed to all zeros for standard pixel-value ranges do not include negative numbers. The network is trained as before but in two stages: first, the network is trained for 50 epochs and the loss is minimized with respect to all network parameters except for T. Then a second stage of tuning is entered where we minimize the loss with respect to the parameter T and the validation set. We repeat this process for several thousand training epochs until the performance on the validation set no longer improves.

Figure 1. Input binarization with RGB Thresholding (first row) and LBP (second row).

Local Binary Patterns (LBP) A well-known technique called *local binary patterns* for extracting multi-resolution and scale-invariant features from images has been introduced in [18]. We use a similar approach in our application for image binarization, but with a slight modification: we operate on the grayscale image and process each pixel by examining its neighborhood at a radius of 1 pixel, generate

3 artificial color channels and select 3 pixels at a clockwise stride of 3 in the neighbourhood to distribute to these channels. Then the value of these pixels are set to 1 if they exceed the value of the center pixel and 0 otherwise. An example of this transformation on an image from the dataset is demonstrated in the second row of Figure 2.3.

2.4. Packing binary-valued vectors

To avoid confusion with terminology, we denote by *packing* the encapsulation/conversion of an array of 1-bit values into an individual 32-bit unsigned integer. Formally, for a binary-valued vector $\mathbf{x} \in \{-1, +1\}^D$, assuming D is divisible by B, then the packed representation of $\mathbf{x}$, $\mathbf{x}_p \in \{0, 1, 2, 3, ..., 2^{B-1}\}^{D/B}$ for a packing bitwidth $B \leq 32$ (assuming 32-bit word) and positive D, is given by

$$
\mathbf{x}_p = \begin{bmatrix}
\sum_{i=1}^{B}(1+x_i)2^{B-2-mod(i-1,B)} \\
\sum_{i=B+1}^{2B}(1+x_i)2^{B-2-mod(i-1,B)} \\
\sum_{i=2B+1}^{3B}(1+x_i)2^{B-2-mod(i-1,B)} \\
\vdots \\
\sum_{i=D-B+1}^{D}(1+x_i)2^{B-2-mod(i-1,B)}
\end{bmatrix}. \quad (2)
$$

3. Implementation

In this section, we present the details of our CUDA implementation of the binarized neural network architecture described in Section 2. We use CUDA terminology throughout this section.

3.1. Convolutional layers

The convolutional layer in a neural network can significantly improve image classification accuracy compared to standard multi-layer perceptrons. Given a kernel $H \in \mathbb{R}^{K \times K \times C}$ and an image $X \in \mathbb{R}^{H \times W \times C}$, an output feature map $F \in \mathbb{R}^{H \times W}$ is given by the expression

$$
F[i,j] = \sum_{c=0}^{C-1}\sum_{l=-R}^{R}\sum_{k=-R}^{R} H[R+l, R+k, c]X[i+k, j+l, c], \quad (3)
$$

for odd K, and the kernel radius $R = \frac{K-1}{2}$. It should be noted that equation (3) in fact computes cross-correlation (not convolution), which is the convention in deep learning. A common approach for computing convolutions efficiently is through matrix multiplication [2], where the weights and image tensors are reshaped into 2-dimensional matrices, which will then allow us to compute the convolution through a single matrix multiplication. The reshaping for the weights is trivial, and this step can often be skipped if the weights are already stored in this layout; however, the process of arranging the input image into the matrix of columns used for computing the convolution can be difficult to optimize. This is due to inefficient access patterns, complicated index

calculations that involves many division and modulo operations, and the overhead of storing the large output matrix to global memory.

A straightforward approach for avoiding inefficient access patterns is to load regions from the image into shared memory (on-chip memory) and then extract the patches from shared memory [3]. For an image with dimensions $H \times W \times C$ corresponding to height, width, and channels respectively, and a $K \times K \times C$ kernel with a radius of $R = \frac{K-1}{2}$, we use threadblock dimensions of $S \times W$ ($S = 2$ in our case), which covers the entire width of the image, eliminating the need to redundantly load the horizontal non-zero halo regions which are difficult to load with an efficient access pattern. Then each thread-block loads an image region of dimensions $(S + 2R) \times W$ into a region in shared memory in three steps, starting by loading the top vertical halo region, the middle part, then the bottom vertical halo region (except when loading from the bottom of the image). The shared memory buffer is zero-initialized in order to implicitly handle horizontal zero-padding. Loading vertical halo regions can be done very efficiently since all threads in the threadblock load from contiguous regions in the image array.

In the second stage, the patches of size $K \times K \times C$ are extracted from shared memory. We avoid division and modulo operations in the patch-extraction stage by using an integer counter register. This results in a $2\times$ performance boost in our case. Since the network is binarized, the packing and patch-extraction step can be fused into one step to avoid redundant accesses to global memory, reducing global memory stores by $K \times K$. The algorithm for the combined step of extracting the patches and packing them is shown in Algorithm 1.

Algorithm 1 Patch-extraction and packing

1: **function** ExtractPacked(sh_block):
2:      $v \leftarrow 0$
3:      $k \leftarrow 0$
4:      **for** $i = 0$ to $B-1$ **do**
5:          **if** $(i - kK = K)$ **then**
6:              $k{+}{+}$
7:          $idx = (W + 2R)(\_\mathtt{t}_y + k) + \_\mathtt{t}_x + i - kK$
8:          $s = \mathrm{sh\_block}[idx] > 0$
9:          $v = \mathrm{bitOR}(\, v, s << B - 1 - i)$
10:      **return** v

In Algorithm 1, `sh_block` is the region of the image loaded into shared memory using the previously described steps, including the halo regions. $\_\mathtt{t}_x$ and $\_\mathtt{t}_y$ are the thread indices for the x and y dimensions of the thread block corresponding to the CUDA `threadIdx.x` and `threadIdx.y` variables. B is the packing bitwidth, chosen to be 25 in our case, $<<$ is the left bit-shift operator, and v is the packed extracted patch.

For computing the convolution we implement a standard matrix multiplication subroutine in a manner similar to [22], where tiles from each matrix are loaded successively into shared memory and used to compute a submatrix of the

TABLE 1. Runtime of the network on each platform

Implementation Method	GTX1080	Mali T860	Tegra X2
cuDNN (full-precision)	$401.83\mu s$	N/A†	2.27 ms
Arm CL (full-precision)	N/A†	29.61 ms	N/A†
BCNN	$102.39\mu s$	23.63 ms	0.53 ms
BCNN with binarized inputs	**$55.63\mu s$**	**17.58** ms	**0.41** ms

†Library not compatible with this platform.

TABLE 2. Runtime per-layer (GTX1080)

Layer	cuDNN	Binarized	Speed-up
Im2col3d $(96, 96, 3)$	$21.63\ \mu s$	$3.17\mu s$	$6.82\times$
GEMM-convolution $(32, 5, 5, 3)$	$37.54\mu s$	$8.61\mu s$	$4.36\times$
Max-Pooling $(96, 96, 32)$	$5.22\mu s$	$8.26\mu s$	$0.63\times$
Im2col3d $(48, 48, 32)$	$65.41\mu s$	$5.50\mu s$	$11.89\times$
GEMM-convolution $(32, 5, 5, 32)$	$69.28\mu s$	$8.10\mu s$	$8.55\times$
Max-Pooling $(48,48,32)$	$5.38\mu s$	$2.66\mu s$	$2.02\times$
Fully-Connected $(100, 24 \times 24 \times 32)$	$200.03\mu s$	$6.28\mu s$	$31.85\times$

output, such that each thread computes a single element in the output matrix, but instead of computing multiplications, we compute xnors and bit-counts following an approach similar to what was suggested in [11] as

$$\mathbf{a}\cdot\mathbf{b} = \mathtt{W} - 2 \times \mathtt{popcount}(\mathtt{xor}(A,B)), \quad (4)$$

where A and B are both 32-bit unsigned integer registers containing the packed representations of vectors $\mathbf{a}$, $\mathbf{b}$ $\in \{-1,+1\}^\mathtt{W}$ respectively. We denote by $\cdot$ the real-valued dot product. The operation $\mathtt{xor}$ is the bit-wise xor operation, and $\mathtt{popcount}$ is a function for computing the number of bits set to 1. The packing bitwidth $\mathtt{W}$ is the number of elements that are packed together in a single unsigned integer register.

3.2. Fully connected layer

For the fully-connected layer, we follow a slightly different approach from standard matrix multiplication. For a packed weights matrix $\mathbf{W} \in \mathbb{R}^{\mathbf{L}\times\mathbf{D}}$, and a packed vector $\mathbf{x} \in \mathbb{R}^{\mathbf{D}\times\mathbf{1}}$, we divide the process of computing the dot product of each weight vector and $\mathbf{x}$ into 64 segments, such that each of 64 threads handling a weight vector compute the partial sum of the dot product between a weight vector and $\mathbf{x}$ through xnor operations, and stores the results in shared memory. The partial sums are then combined in a parallel reduction sum that does not require synchronization (for a warp size of 32 on the target platform).

4. Results

In this section we present our results for the impact of input binarization on classification accuracy and the performance improvement achieved.

Input binarization in Table 3 we report the classification accuracy results we obtained using each different input binarization scheme for our binarized version of the vehicle classifier [12]. We can observe that accuracy is best retained when the first layer is not binarized; however, only a moderate loss in accuracy occurs when using LBP and RGB Thresholding. Considering that RGB Thresholding is much simpler to implement and results in almost no additional computational overhead, we choose this approach for our final binarized architecture, for which we report the speed up results in the following section.

Performance Boost We time our binarized implementation on 3 different hardware platforms: Nvidia GTX 1080,

Nvidia Jetson (Tegra X2), and the Mali-T860. We derive an OpenCL version of our implementation for testing on the Mali-T860, which is a straightforward process. We compare the performance of our implementation against an equivalent full precision version of the same network implemented with highly optimized libraries on each target platform, in our case these are cuDNN on Nvidia platforms, and the ARM Compute Library on the Mali-T860. We list in Table 1 the average execution times of the full network on each platform. We can see that our binarized implementation can achieve up to $7.5\times$ speed up on the GTX1080 and about $5.5\times$ on the Tegra X2. We also notice that the relative performance improvement on Mali GPU is much smaller at about $1.7\times$ for the fully binarized version. In our optimizations, we heavily take advantage of using local memory (in OpenCL terms) which resides on-chip in most workstation GPUs and the Nvidia Tegra X2, but this does not offer any performance benefits on Mali GPUs since local memory is allocated in global memory. It should be noted that cuDNN is optimized for batch processing and that our results are for one sample at a time which means these results may not necessarily be reflective of the full potential of cuDNN; however, batch processing is not a suitable option for real-time applications where a single input is processed at a time. Additionally, we note that for our cuDNN implementations, we use the explicit GEMM convolution algorithm, which can be slightly slower than the implicit GEMM algorithm. For example, cuDNN with implicit GEMM can run at $316\mu s$ for the first convolutional layer in our network on the GTX1080.

For a more detailed comparison, we present the execution times for each individual layer in Table 2. Each layer's name is followed with the dimensions of the input, except for the convolution layers where the dimensions are for the kernels, and the input dimensions can be inferred from the previous layer. This table compares the execution time of our binarized implementations with the full-precision versions of the same layer in cuDNN on the GTX1080. We omit from the table the computation times for ReLU activations, which are present in the full-precision version of the network, but are absent from the binarized version. We also omit the last 2 fully-connected layers since they are too small and in most practical applications it would be more efficient to implement them on the CPU. We include the computation time for packing the outputs of the previous layer in the binarized version of the fully-connected layer for

TABLE 3. IMPACT OF DIFFERENT INPUT-BINARIZATION SCHEMES ON CLASSIFICATION ACCURACY

Method	Accuracy
LBP	92.06%
Thresholding Grayscale	89.16%
Thresholding RGB	92.52%
No input binarization	94.20%
Full-precision network	97.09%

a fair comparison. The results in Table 2 have been obtained directly from the Nvidia Visual Profiler.

It should be noted that the runtime for the fully-connected layer for full-precision cuDNN in Table 2 includes a matrix transposition. The run time excluding matrix transposition is about $100\mu s$; however, it is a necessary step for evaluating this layer. Our full-precision matrix multiplication kernel is in fact $2\times$ slower than cuBLAS (as measured in this network), yet a significant speed-up is still achievable through binarization.

5. Conclusion and Future Work

We presented an efficient implementation of a binarized convolutional neural network on GPUs that can achieve a significant decrease in runtime while reasonably preserving classification accuracy. In the future we wish to restructure our algorithms to achieve a similar performance improvement on other embedded platforms. We are also planning to extend this work to alternative convolution algorithms such as implicit GEMM, which can be faster than explicit GEMM. Finally, we plan to extend our study of how input binarization impacts classification accuracy on larger datasets with more difficult classification tasks.

Acknowledgment

References

[1] Alvarez J, Petersson L. Decomposeme: Simplifying convnets for end-to-end learning. arXiv preprint arXiv:1606.05426. 2016 Jun 17.

[2] Chellapilla K, Puri S, Simard P. High performance convolutional neural networks for document processing. In Tenth International Workshop on Frontiers in Handwriting Recognition 2006 Oct 23.

[3] Chetlur S, Woolley C, Vandermersch P, Cohen J, Tran J, Catanzaro B, Shelhamer E. cudnn: Efficient primitives for deep learning. arXiv preprint arXiv:1410.0759. 2014 Oct 3.

[4] Ciregan D, Meier U, Schmidhuber J. Multi-column deep neural networks for image classification. In Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on 2012.

[5] Courbariaux M, Bengio Y, David JP. Binaryconnect: Training deep neural networks with binary weights during propagations. In Advances in neural information processing systems 2015 (pp. 3123-3131).

[6] Glorot X, Bordes A, Bengio Y. Deep Sparse Rectifier Neural Networks. In Aistats 2011 (Vol. 15, No. 106, p. 275).

[7] Graves A, Jaitly N. Towards End-To-End Speech Recognition with Recurrent Neural Networks. In ICML 2014 (Vol. 14, pp. 1764-1772).

[8] Gregor K, Danihelka I, Graves A, Rezende DJ, Wierstra D. DRAW: A recurrent neural network for image generation. arXiv preprint arXiv:1502.04623. 2015.

[9] Han S, Pool J, Tran J, Dally W. Learning both weights and connections for efficient neural network. InAdvances in neural information processing systems 2015 (pp. 1135-1143).

[10] Hinton, G. Neural networks for machine learning. Coursera, video lectures. (2012).

[11] Hubara I, Courbariaux M, Soudry D, El-Yaniv R, Bengio Y. Binarized neural networks. In Advances in neural information processing systems 2016 (pp. 4107-4115).

[12] Huttunen H, Yancheshmeh FS, Chen K. Car type recognition with deep neural networks. In Intelligent Vehicles Symposium (IV), 2016 IEEE 2016 Jun 19 (pp. 1115-1120).

[13] Jaderberg M, Vedaldi A, Zisserman A. Speeding up convolutional neural networks with low rank expansions. arXiv preprint arXiv:1405.3866. 2014 May 15.

[14] Kanda N, Takeda R, Obuchi Y. Elastic spectral distortion for low resource speech recognition with deep neural networks. In Automatic Speech Recognition and Understanding (ASRU), 2013 IEEE Workshop on 2013 (pp. 309-314).

[15] Kingma DP, Ba J. Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980. 2014 Dec 22.

[16] Krizhevsky A, Sutskever I, Hinton GE. Imagenet classification with deep convolutional neural networks. In Advances in neural information processing systems 2012 (pp. 1097-1105).

[17] Lawrence S, Giles CL, Tsoi AC, Back AD. Face recognition: A convolutional neural-network approach. IEEE transactions on neural networks. 1997;8(1):98-113.

[18] Ojala T, Pietikäinen M, Mäenpää T. Multiresolution gray-scale and rotation invariant texture classification with local binary patterns. IEEE Transactions on pattern analysis and machine intelligence. 2002 Jul; 24(7):971-87.

[19] Parkhi OM, Vedaldi A, Zisserman A. Deep Face Recognition. In BMVC 2015 (Vol. 1, No. 3, p. 6).

[20] Pedersoli F, Tzanetakis G, Tagliasacchi A. Espresso: Efficient Forward Propagation for BCNNs, ICLR 2018 (to appear).

[21] Rastegari M, Ordonez V, Redmon J, Farhadi A. Xnor-net: Imagenet classification using binary convolutional neural networks. In European Conference on Computer Vision 2016 Oct 8 (pp. 525-542).

[22] Tan G, Li L, Triechle S, Phillips E, Bao Y, Sun N. Fast implementation of DGEMM on Fermi GPU. In Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis 2011 Nov 12 (p. 35).

[23] Tieleman T, Hinton G. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. COURSERA: Neural networks for machine learning. 2012;4(2).