Espresso: Efficient Forward Propagation for BCNNs

Fabrizio Pedersoli
Department of Computer Science
University of Victoria
fpeder@uvic.ca

George Tzanetakis
Department of Computer Science
University of Victoria
gtzan@uvic.ca

Andrea Tagliasacchi
Department of Computer Science
University of Victoria
ataiya@uvic.ca

Abstract

There are many applications scenarios for which the computational performance and memory footprint of the prediction phase of Deep Neural Networks (DNNs) needs to be optimized. Binary Neural Networks (BDNNs) have been shown to be an effective way of achieving this objective. In this paper, we show how Convolutional Neural Networks (CNNs) can be implemented using binary representations. *Espresso* is a compact, yet powerful library written in C/CUDA that features all the functionalities required for the forward propagation of CNNs, in a binary file less than 400KB, without any external dependencies. Although it is mainly designed to take advantage of massive GPU parallelism, Espresso also provides an equivalent CPU implementation for CNNs. Espresso provides special convolutional and dense layers for BCNNs, leveraging *bit-packing* and *bit-wise* computations for efficient execution. These techniques provide a speed-up of matrix-multiplication routines, and at the same time, reduce memory usage when storing parameters and activations. We experimentally show that Espresso is significantly faster than existing implementations of optimized binary neural networks ($\approx$ 2 orders of magnitude). Espresso is released under the Apache 2.0 license and is available at `http://github.com/fpeder/espresso`.

1 Introduction

Convolutional Neural Networks have revolutionized computer vision, pushing the task of object recognition beyond human capabilities [18, 24, 26]. Deep Neural Networks (DNN), have also been successfully applied in other fields, such as speech recognition [11, 13] and automated translation [2, 25]. Despite achieving impressive classification accuracy results, DNNs require too much memory and power to be used effectively on embedded or low-power devices. Many networks consume a considerable amount of memory. Memory remains a very limited resource on mobile platforms, and, for example, the popular AlexNet [18] architecture consumes 250MB. Even when memory is not an issue, DNNs remain very computationally intensive, and can quickly drain the battery. Reducing the computational load does not only improve energy efficiency, but can also enable further applications. For example, when processing real-time object classification on mobile, being able to perform faster predictions frees up computational resources that can be spent on tasks such as speech recognition and analysis. Therefore, there is a substantial interest in reducing the computational and memory requirements of DNNs.

31st Conference on Neural Information Processing Systems (NIPS 2017), Long Beach, CA, USA.

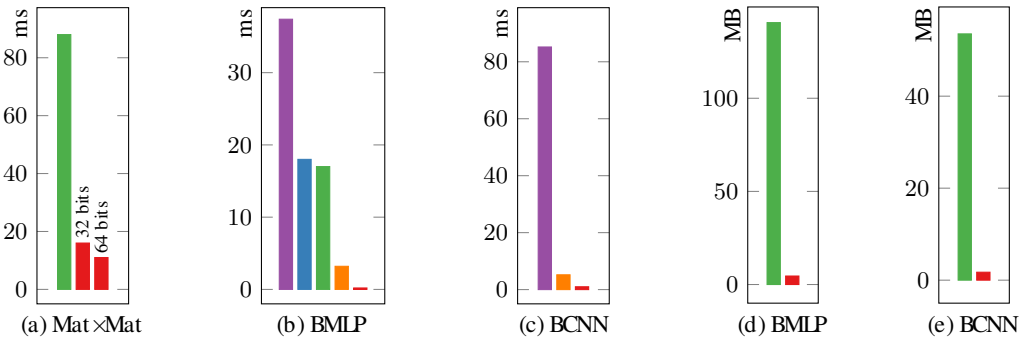


Figure 1: We introduce *Espresso*, a framework providing forward-propagation of Binary Deep Neural Networks (BDNN) on CPU, GPU, and optimized GPU* with substantially enhanced computational efficiency and memory requirements when compared to existing GPU implementations: BinaryNet [14] or Intel Nervana/neon [21]. Our experiments measured: (a) a speedup of $\approx 8\times$ for GPU-based (32 or 64 bits) dense binary matrix multiplications of size 8192^2 – the core operation for BDNNs [14]; (b) a speedup of $\approx 68\times$ for the forward propagation of Binary MultiLayer Perceptron (BMLP) architectures [14]; (c) the first highly efficient implementation of Binary Convolutional Neural Networks (BCNN). (d,e) A reduction in memory consumption of $\approx 32\times$ compared to the existing implementations of BMLPs and BCNNs. Average execution time and memory consumption are reported in milliseconds (ms) and megabytes (MB).

Efficient deep neural networks. One way to achieve this target is to use specialized hardware for DNNs. Another strategy is to tune the network to reduce its memory footprint, hence increasing its efficiency. Such solutions are preferable as they can be implemented in software without requiring specialized hardware. In our research we follow the software approach, and focus our attention to *quantized* networks. In particular, we consider the *binary* deep neural networks (BDNN or BinaryNet) proposed by Hubara et al. [14]. At the expense of a relatively small decrease in accuracy, BDNNs can considerably reduce memory usage, and result in faster execution time (i.e. forward propagation). Further, note that potential hardware implementation of BDNNs would also be cheaper due to the reduced number of required FPUs. While these results are highly promising, currently only *proof-of-concept* implementations of BinaryNets have been published [14]. The availability of a flexible end-to-end framework, with particular emphasis placed on computational efficiency, can enable further research on BDNNs, as well as its application to practical scenarios.

Contributions. With *Espresso* we aim at filling this gap and provide an optimized framework for BDNNs capable of achieving state-of-the-art run-time *performance* with minimal *memory* footprint; see Figure 1. In particular, Espresso has achieved up to a $\mathbf{68}\times$ speedup compared to currently available implementations of BDNN, and its memory footprint is $\approx \mathbf{32}\times$ smaller than those of existing solutions. While our work is a necessary stepping stone towards optimization of training routines, in this paper we focus on the optimization of forward-propagation (i.e. testing), rather than back-propagation (i.e. training). Current state-of-the-art optimized-BDNNs implementations are limited to fully connected layers and ignore other networks, such as CNNs, that can take advantage of binary optimizations. Espresso executes binary-optimized *dense* neural networks (e.g MLP) with state-of-the-art computational performance, while also pioneering the efficient forward-propagation of binary *convolutional* neural networks (BCNN). Our framework was designed to have no external dependencies. This not only results in a highly optimized implementation of BDNNs, but also substantially simplifies its deployment in practical applications, such as those executing on mobile or embedded devices.

2 Related Works

Improving the performance of DNNs can be achieved at either the hardware or software level. At the hardware level, chipsets that are dedicated to DNN execution can outperform general-purpose CPUs/GPUs [16]. At the software level, the network can be simplified to increase performance, where one common approach is to penalize the total number of non-zero weights (i.e. connections)

via a modified loss function [5]. Another recently proposed approach is to *quantize* the network [6], such that dense linear algebra operations can be executed more efficiently.

Quantized networks. In quantized networks, the objective is to train DNNs whose (quantized) weights do not significantly impact the network's precision (i.e. classification accuracy). For example, Courbariaux et. al. [6] shows that 10-bits are enough for Maxout Networks, and how more efficient multiplications can be performed with fixed-point arithmetic. Continuing this trend, Hwang et. al. [15] proposed fixed-point DNN with *ternary* weights $\{-1, 0, +1\}$. Their training leveraged an optimized backtracking procedure for fixed-point data, obtaining precision very close to that of the floating-point baseline.

Binary Deep Neural Networks (BDNN). Recently, Courbariaux et al. [7] showed that a network with *binary* $\{-1, +1\}$ weights can achieve near state-of-the-art results on several standard datasets. Binary DNNs (BDNNs) were shown to perform effectively on datasets with relatively small images, such as the permutation-invariant MNIST [19], CIFAR-10 [17] and SVHN [22]. Recently, Rastegari et. al. [23] show that binarized CNNs can perform well even on massive datasets such as ImageNet [9] using binarized versions of well-known DNN architectures such as AlexNet [18], ResNet-18 [12], and GoogLenet [26]. Similarly interesting results can be achieved by binarizing both DNN weights and activations as showed by Courbariaux et. al. [14]. In this work, the authors introduce *BinaryNet*, a technique to effectively train DNNs where both weights and activations are constrained to $\{-1, +1\}$. *BinaryNet* achieves nearly state-of-the-art precision for MLP training on MNIST and CNN training on CIFAR-10. Their optimizations result in $7\times$ faster performance than the base-line kernel, and, almost $2\times$ faster than Theano [4]. Their core contributions, namely to replace Floating-point Multiply and Add operations (FMAs) with *XNORs* and *bit-counts*, represent the cornerstone over which we build our research.

3 The Espresso Framework

Espresso provides the user with the necessary tools for executing forward-propagation of DNNs, with particular emphasis placed on convolutional neural networks, due to their ubiquitousness in computer vision applications. As the complexity of these networks is cubic to the size of the problem, they are less memory efficient and more computationally intensive than traditional machine-learning algorithms. Identifying the memory and computational bottlenecks of DNNs is therefore essential to enable their practical application. In particular, our primary focus is *GPU-optimized* BDNN architectures, which we refer to as **GPU***, but we also support the equivalent floating-point counterparts on heterogeneous architectures, which in our discussion we simply identify as **CPU** and **GPU**.

Hybrid DNNs. The Espresso's implementations of tensors and layers come in three variants $\{\text{CPU}, \text{GPU}, \text{GPU}^*\}$. A CPU-tensor is allocated in CPU memory, and is processed on the CPU using sequential code. A GPU-tensor is allocated on GPU main memory and is processed by CUDA kernels. Espresso provides functions for *converting* tensors and layers from one variant to the other, and different variants can also be interconnected with each other. Consequently, Espresso enables the design of hybrid DNNs consisting of a combination of $\{\text{CPU}, \text{GPU}, \text{GPU}^*\}$ layers.

The computational bottleneck: dot products. Dense linear algebra is at the heart of deep-learning as deep networks can be viewed as a composition of *matrix-matrix*, *matrix-vector* and *elementwise matrix-matrix or vector-vector* multiplications. The implementation of these dense linear algebra operations relies heavily on the efficient computation of the *dot-product*. The execution of this operator consists of (single precision) *Floating-point Multiply and Add* (FMA) operations. In modern architectures, floating-point multiplications executing on the FPU dominate the complexity of FMAs, and BDNNs address these concerns by replacing FMAs with simpler *bitwise* operations; see Section 4.

Technical highlights. The superior computational performance of Espresso derives from three main technical contributions: (1) the use of bit-packing in network layers, (2) better memory layout, and (3) our optimized CUDA kernels. Through the use of bit-packed layers, Espresso can execute a forward operation without the need for expensive memory re-arrangements employed by existing implementations. As dynamic memory allocation on GPUs is a performance bottleneck, Espresso pre-allocates all resources during initialization, including the *scratch* memory used for intermediate computations. Finally, matrix multiplications are performed with CUDA kernels that have been adapted to bit-packing, and only resort to XNORs and bit-counts.

4 Binary Deep Neural Networks (BDNN) – Hubara et al. [14]

In this section, we overview the fundamental characteristics of BDNNs that inform the basics of Espresso's design. In Binary DNNs, computationally intensive FMA operations are replaced by *XNOR* (for multiplications) and *bit-count* (for additions), enabling significant computational speed-ups. In particular, XNOR is a simpler machine instruction compared to floating point multiplication, and therefore achieves much higher throughput on many architectures. More importantly, a single XNOR step can execute multiple 64bits-wide blocks of dot-products, further increasing the overall computational efficiency. In what follows, we describe how a network is binarized, detail a compressed memory layout enabling efficient execution of dot-products, show how to re-interpret input data to allow execution on fixed-precision input (e.g. images), and provide a few notes regarding the training procedure.

4.1 Network binarization

A BDNN is composed of a sequence of $k = 1, \ldots, L$ layers whose weights W_k^b and activations a_k^b are binarized to the values $\{-1, +1\}$. The superscript b in the notation indicates binary quantities. Weights and activations are $\{-1, +1\}$, but at the hardware level they must be encoded as $\{0, 1\}$. Our convention is to encode $-1 \rightarrow 0$ and $+1 \rightarrow 1$. Amongst many possible choices, e.g. stochastic binarization [7], we employ the following activation function due to its efficient implementation:

$$x^b = \text{sign}(x) = \begin{cases} +1 & x \geq 0 \\ -1 & \text{otherwise} \end{cases} \tag{1}$$

4.2 Bit-packing

The weights of a BDNN can be stored in the bits of a 64-bit word. One immediate advantage of bit-packing is to drastically reduce the memory usage by a $32\times$ factor. An even more significant advantage is the ability to process multiple values at the same time using registers. This is particularly useful for dot-products: with bit-packing we can compute a dot-product of 64 element vectors by using just one XNOR and one bit-count. Furthermore, modern computer architectures provide a hardware instruction for counting the number of bits set to 1 in a given word. Assuming binary vectors $a, b \in \mathbb{B}^{1\times N}$ where N is a multiple of 64, the dot-product is then equivalent to:

$$a \cdot b \;\equiv\; N - \left(\sum_{i=1}^{N/64} \text{bitcount}(\text{XNOR}(a_i, b_i)) \right) \ll 1 \;\triangleq\; a \odot b \tag{2}$$

where $\ll$ represents the bit-shift operator. This simple computation becomes the building block of optimized BDNNs as binary matrix-matrix or matrix-vector operations are computed in this fashion.

4.3 Input data binarization

BDNNs require binary input data, which is not typically available at the first layer of the network. However, the input data usually comes in a fixed precision format (e.g. 8-bit/channel in RGB images). Therefore, the optimized computation of dot-products can still be applied if we split the input data according to bit-planes, and then sum back each contribution according to the corresponding weight. For instance, if with $\langle a \rangle_n$ we indicate the n-th bit of a fixed precision vector, and with i the corresponding bit-plane, we obtain:

$$a \cdot b \equiv \sum_{i=0}^{n-1} 2^i \langle a \odot b \rangle_i \tag{3}$$

4.4 Training

When training a BDNN, it is important to note that the gradient is computed with the binary weights, but is accumulated with floating point precision [14]. That is because the optimizer needs sufficient precision to make a reliable update. In addition, the derivative of the sign function, which is zero almost everywhere, cannot be used for back-propagation. To overcome these issues, the *straight-through estimator* [3] is employed, where 1 is back-propagated if the floating point argument $|x| \leq 1$,

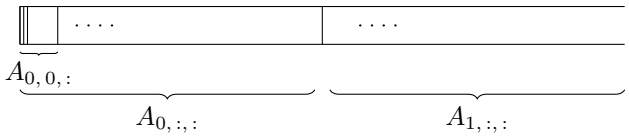
and 0 otherwise. Finally, during training weights are clipped to $[-1, 1]$ to avoid a large growth of the floating point weights that would not have an impact on the binary weights.

5 Espresso architecture

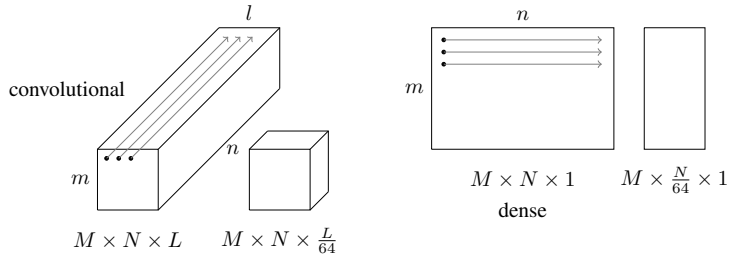
The principal components of our framework are *tensors*, *layers* and the *network*. These components are organized as a hierarchy. Tensors are n dimensional matrices used for storing *inputs*, *weights* and *activations* (outputs). A layer processes an input tensor and produces an output tensor, while a network consists of a concatenation of layers.

5.1 Tensors

In Espresso, each element of a tensor $A \in \mathbb{R}^{M \times N \times L}$ is identified by the triplet m, n, l, where $m \in [0, M)$ indicates the row, $n \in [0, N)$ indicates the column, and $l \in [0, L)$ indicates the channel. A tensor is stored in memory using row-major order with interleaved channels. Therefore, according to this layout, the element $A_{m,n,l}$ is found at position $(mN + n)L + l$ in linear memory.



We use the notation $A_{m,n,:}$, to indicate all the channels of the (m, n)-th element. Using the same storing scheme Espresso also defines bit-packed tensors for GPU* implementations but with the following changes to further increase its performance. Bit-packing is performed according to the number of channels: when $L > 1$ bit-packing is done along the l dimension; when $L = 1$ bit-packing is done along the n dimension. For *convolutional* layers this packing direction enables efficient



memory access when unrolling/lifting a tensor, which would have not been possible if either m or n had been chosen instead. More specifically, this layout is optimal for retrieving a pixel neighborhood as needed by convolution without requiring the layout to be changed. Further, typically a large number of filters are used resulting in an increase of tensor dimension in the l direction, while the m and n dimensions are progressively shrunk by pooling layers. For other layer types, n is the most efficient packing direction, as neurons are stored along rows, and their number decreases as we move toward later stages in the network.

5.2 Layers

Espresso provides the following layer types: *Input*, *Convolutional*, *Pooling*, *Dense* (i.e. fully connected) and *Batch-normalization*. Each layer is characterized by its size, tensor parameters and output. The Espresso API defines for each layer a *forward* function that computes the output of a layer given an input tensor, and a function for applying *non-linearity* to the outputs of convolutional and dense layers. Moreover, the convolutional layer features additional functions for *pooling* and *unrolling*.

Convolutional layers. In our framework, 2D convolutions are computed through matrix multiplications – an operation involving a very high reuse of data. For both CPU and GPU, this computation is

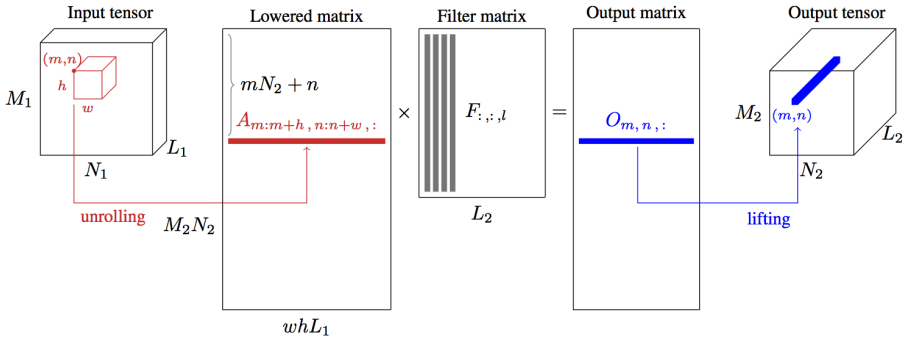


Figure 2: *unrolling* and *lifting* operations for CNN layers

performed by sectioning data in amounts that are cache-friendly [10], resulting in implementations attaining close to peak computational performance. However, in order to express convolution as matrix multiplication we need to re-organize the input memory appropriately. This is achieved through the *unrolling* procedure; see Figure 2. It consists of transforming a tensor into a matrix where each row is formed by unrolling the tensor data contained in each convolution sliding volume. The unrolled matrix is then multiplied by the filter matrix. Finally, the result of the convolution is reordered back to tensor by using the *lifting* procedure. In Espresso we do need to manually lift the convolution result in order to undo the unrolling: thanks to our tensor representation this happens automatically and at zero cost. Espresso provides CUDA kernels for the *unrolling* and *pooling* of tensors for both GPU and GPU* implementations.

Efficient Matrix multiplication. Matrix-vector multiplications are fundamental operations of both dense and CNN layers. For the CPU architecture, we use the OpenBLAS library [27] to implement these operations. For GPU and GPU* architectures, the CUDA kernels are based on *Magma(sgemm)* [1], modified to make it compatible with our binary data representation. These kernels for matrix multiplication feature *register blocking* optimization: since the introduction of *Fermi* architectures the number of registers have been increased, while register access latency has been substantially reduced compared to shared-memory; hence caching at the register-memory level results in considerably faster throughput [20]. Espresso first fetches the tiles of the two matrices into shared-memory and then process sub-tiles using registers. In the GPU* variant, we modify the code by replacing blocks of 64 (or blocks of 32 for GPU*32) single precision multiply and add (FMA) operations with XNOR and bit-count using packed tensors. We also re-tune the kernel block size parameters for improving the performance on reduced size matrices.

Zero-padding for convolutions. Typical CNN implementations apply a tensor convolution in a "same" configuration, where the sizes of input and output tensors matches. This is achieved by zero-padding input tensors, but in convolutional GPU* layers the *zero-padding* of the input introduces the side-effect of making the data ternary $\{-1, 0, +1\}$. We deal with this problem by treating the data as if it was binary (zero is considered a minus one) and fix the results of the convolution at these corner-cases in post-processing. This allows us to leave the convolution kernel code – the computational bottleneck of the code – untouched. The corner-cases are fixed using a highly efficient kernel which executes an element-wise sum between the results of the convolution and the correction matrix. The correction matrix is computed once, when the GPU* layer is loaded, and it simply consists of the convolution of the layer's weights with a $(+1)$-padded zero-tensor.

Training Espresso. A DNN in Espresso is defined as a combination of layers, which is loaded at runtime by reading its parameters file. The parameters file specifies the storage format of all the layers, as well as their weights. Therefore, it completely specifies a DNN as layers are stored sequentially. Training of the network is done by BinaryNet [14]; the resulting parameters are converted to the Espresso format by utility script distributed together with our sources.

6 Evaluation

The performance of our framework is evaluated in terms of average computational time needed to perform a particular task. The execution times, averaged over 100 experiments, are obtained on a machine equipped with an NVIDIA GeForce GTX 960 with 2GB of RAM, and a Intel® dual-Xeon® X5660 @ 2.80 GHz. In CPU mode, we configure the OpenBLAS library for matrix multiplication to use all the 24 available cores.

Experimental design. We perform three quantitative evaluations: (Section 6.1) matrix multiplications of two dense square matrices of size 8192×8192; (Section 6.2) forward-propagations of a Multi-Layer Perceptron (MLP) trained on the MNIST dataset [19]; (Section 6.3) forward-propagations of a Convolutional Neural Network (CNN) trained on the CIFAR-10 dataset [17]. We compare Espresso with: (1) the author provided implementation of BinaryNet [7]; (2) the optimized BDNN implemented in the Intel Nervana *neon* framework [21]; (3) a self-comparison across $\{\text{CPU}, \text{GPU}, \text{GPU}^*\}$ as no binary-optimized implementations of convolutional layers are publicly available.

Public datasets. The MNIST dataset [19] consists of 60K instances for training and, 10K instances for testing. Each instance is a 28×28 grayscale image that depicts digits ranging from 0 to 9. The CIFAR-10dataset [17], consists of 50K training instances and 10K testing instances of $32 \times 32 \times 3$ color images. Images are subdivided into 10 classes (airplanes, automobiles, birds, cats, deers, dogs, frogs, horses, ships and trucks). Since our interest is to asses the real-time performance of binary optimized DNNs, in those experiment we use a batch-size of one, and measure the averaged forward time for each image of the testing-sets for each dataset.

6.1 Dense matrix multiplication – Figure 1a

In computing dense matrix multiplication, Espresso outperforms BinaryNet by a $\approx \mathbf{8}\times$ factor. Much of the gain can be attributed to our optimized kernels, and the use of register blocking: by fetching bigger data from main memory and shared memory, our kernel increases the bandwidth utilization by decreasing the number of memory fetch instructions. The use of 64-bit packing instead of the 32-bit (such as that of BinaryNet), introduces an additional performance improvement. The 64-bit kernel achieves a memory DRAM throughput of 40GB/s for reads and 5GB/s for writes, while the 32-bit kernel obtain 29.6GB/s for reads and 3.6GB/s for writes. This translates into the resulting $\approx 25\%$ speed improvement.

6.2 Multi-layer perceptron on MNIST – Figure 1b and Figure 1d

We evaluate the average classification execution time over the MNIST database, where we trained the MLP architecture from [8, Sec 2.1] with author-provided sources, and then converted it to Espresso's format. In Figure 1b, Espresso achieves a consistent speed-up of $\mathbf{68}\times$ when compared to BinaryNet. As the Nervana/neon implementation of binary network is a BinaryNet derivative, it is affected by the same drawbacks of BinaryNet, and hence achieves comparable performance. Both alternatives have the additional cost of running CUDA through Python/Theano which may introduce further latency in the process. In Figure 1b, the evaluation over the three variants of Espresso shows the expected outcome, with the GPU* implementation leading the ranking. Note that we are able to achieve a speedup of $\approx \mathbf{12}\times$ on an NVIDIA GTX 960 (≈ 2.5 TFLOPs), although this device has only roughly four times more throughput than the Xeon X5660 (≈ 500 GFLOPs without turbo-boost). Through binary optimization, we are able to further increase the performance to $\approx 15\times$ with respect to the GPU implementation. We attribute our computational gains to (1) the use of *binary-optimized* layers, (2) our use of *optimized kernels* for matrix multiplication and (3) Espresso's ability to perform binary optimization of the first layer.

Binary optimized layers. An evident drawback of Binary-Net is the need for binarizing/packing the layer's parameters *every time* a forward method is called. In the case of binary optimized networks, the cost of packing the parameters is closely related to the cost of multiplication itself. Therefore, the reduction of bit-packing function calls leads to a consistent improvement. This motivates our choice of designing specific layers, where bit-packing is done once during network loading.

Optimized kernels. BinaryNet employs two bit-packing kernels: one for row-packing, the other for column-packing. Although BinaryNet's pack-by-rows kernel is slightly slower than ours (8%), the pack-by-columns kernel is significantly slower ($4\times$) due to non-coalesced accesses to global

memory. An additional performance gain of $\approx 15\%$ is achieved by swapping matrix-vector in favour of matrix-matrix multiplication kernels when appropriate (i.e. Dense layers with batch size=1); for this reason, Espresso also includes the binary-optimized MAGMA(sgemv) kernel.

First-layer binary optimization. Another important advantage offered by Espresso is the ability to leverage binary optimization in the *first* layer. Since the first stage of a network processes non-binary data, BinaryNet does not feature binary optimization for this layer. However if the input data is split into its constituent bit-planes, binary optimization can still be applied. In particular, we split the input vector in a matrix of 8 rows, and recombine the result after multiplication by a weighted sum. Our experimental results report an overall $\approx \mathbf{3}\times$ performance boost when comparing the full binary optimized network with one in which the first layer is not binary optimized.

6.3 Convolutional Neural Network on CIFAR-10 – Figure 1c

To the best of our knowledge, no BDNN implementation of *binary-optimized* CNN layers is publicly available. Our self-evaluation implements the *VGGNet*-like CNN architecture from Hubara et al. [14, Sec. 2.3], and evaluates it across our three modalities: as expected the GPU* implementation achieves significantly better performance.

Unrolling and pooling. Note how the GPU implementation offers a slightly better improvement over CPU with respect to the MLP test, with an $\approx \mathbf{16}\times$ speed-up. In this experiment, the inherent parallelism of unrolling and pooling, and the GPU higher memory throughput explain the behavior. Gains are marginal as FMA still represents the computational bottleneck.

Bit-packing. The GPU* implementation results in a $\approx \mathbf{5}\times$ performance gain with the respect to GPU. These gains, to binary optimizations, are slightly smaller than those discussed for MLP in Section 6.2. The output of convolutional layers is significantly larger than those of MLP's dense layers, therefore, the computation of bit-packing sign-activation requires more computational effort.

7 Conclusions

In this paper we presented Espresso, a highly optimized forward-propagation framework for both traditional DNNs as well as BCNNs, that supports heterogeneous deployment on CPU and GPU. While BinaryNet and Nervana/neon BDNN implementations are limited to MLP networks, our framework also supports the popular CNN while simultaneously outperforming state-of-the-art implementations of MLP networks. Espresso is highly-efficient, light-weight and self-contained. Computation on the GPU side is done though specifically designed CUDA kernels, which, combined with a more careful handling of memory allocation and bit-packing, allows us to obtain considerable performance improvements. In future work we would like to add training capabilities, and perform additional performance comparisons on larger standard datasets.

References

[1] Ahmad Abdelfattah, Azzam Haidar, Stanimire Tomov, and Jack Dongarra. Performance, design, and autotuning of batched GEMM for GPUs. In *International Conference on High Performance Computing*, pages 21–38. Springer, 2016. `http://icl.cs.utk.edu/magma/` (Link verified on May 11th, 2017).

[2] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[3] Yoshua Bengio, Nicholas Léonard, and Aaron Courville. Estimating or propagating gradients through stochastic neurons for conditional computation. *arXiv preprint arXiv:1308.3432*, 2013.

[4] James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. Theano: A CPU and GPU math compiler in python. In *Proc. 9th Python in Science Conf*, pages 1–7, 2010.

[5] Y-lan Boureau, Yann L Cun, et al. Sparse feature learning for deep belief networks. In *Advances in neural information processing systems*, pages 1185–1192, 2008.

[6] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Training deep neural networks with low precision multiplications. *arXiv preprint arXiv:1412.7024*, 2014.

[7] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Binaryconnect: Training deep neural networks with binary weights during propagations. In *Advances in Neural Information Processing Systems*, pages 3123–3131, 2015.

[8] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1. *arXiv preprint arXiv:1602.02830*, 2016.

[9] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pages 248–255. IEEE, 2009.

[10] Jack J Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain S Duff. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software (TOMS)*, 16(1):1–17, 1990.

[11] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. Speech recognition with deep recurrent neural networks. In *Acoustics, speech and signal processing (icassp), 2013 ieee international conference on*, pages 6645–6649. IEEE, 2013.

[12] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 770–778, 2016.

[13] Geoffrey Hinton, Li Deng, Dong Yu, George E Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N Sainath, et al. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine*, 29(6):82–97, 2012.

[14] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks. In *Advances in Neural Information Processing Systems*, pages 4107–4115, 2016. `https://github.com/MatthieuCourbariaux/BinaryNet`.

[15] Kyuyeon Hwang and Wonyong Sung. Fixed-point feedforward deep neural network design using weights+ 1, 0, and- 1. In *Signal Processing Systems (SiPS), 2014 IEEE Workshop on*, pages 1–6. IEEE, 2014.

[16] Norm Jouppi. Google supercharges machine learning tasks with TPU custom chip, 2017.

[17] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. The CIFAR-10 dataset. `https://www.cs.toronto.edu/~kriz/cifar.html`, 2009. Link verified on May 11th, 2017.

[18] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[19] Yann LeCun, Corinna Cortes, and Christopher J.C. Burges. The MNIST database of handwritten digits. `http://yann.lecun.com/exdb/mnist`, 1998. Link verified on May 11th, 2017.

[20] Rajib Nath, Stanimire Tomov, and Jack Dongarra. An improved magma gemm for fermi graphics processing units. *The International Journal of High Performance Computing Applications*, 24(4):511–515, 2010.

[21] Intel NervanaSystems. The Neon deep learning framework. `https://github.com/NervanaSystems/neon`. Link verified on May 11th, 2017.

[22] Yuval Netzer, Tao Wang, Adam Coates, Alessandro Bissacco, Bo Wu, and Andrew Y Ng. Reading digits in natural images with unsupervised feature learning. In *NIPS workshop on deep learning and unsupervised feature learning*, 2011.

[23] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European Conference on Computer Vision*, pages 525–542. Springer, 2016.

[24] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[25] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

[26] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–9, 2015.

[27] Zhang Xianyi, Wang Qian, and Werner Saar. OpenBLAS: An optimized BLAS library. `http://www.openblas.net`. Link verified on May 11th, 2017.