FAST QUANTIZED NEURAL NETWORK WITH 4 BITS COMPRESSION

Ladislas de Naurois, Matteo Turchetta, Luca Corinzia

Department of Computer Science
ETH Zürich
Zürich, Switzerland

ABSTRACT

Neural Networks are a class of models for making inference over complex non-linear functions that have established the state of the art for several machine learning tasks. Despite their success, their diffusion in the world of embedded devices is limited by their memory and computational requirements. These requirements stem from the high number of parameters, usually stored as floats, that is required to represent a Neural Network. To overcome this, quantization compresses these parameters to a representation that uses a predefined number of bits, k. The benefits in terms of memory are evident. However, the computational benefits of this representation have not been explored extensively. In this work we introduce an optimized implementation of a 4-bits quantized neural network. Such level of compression is challenging because byte addressability of computer memories forces even our vanilla implementation to work on multiple data simultaneously. Our optimizations concern ops count reduction, memory optimization and vectorization. Our implementation is able to achieve a speed-up up to $\times 14$ for some functions and a $\times 2.5$ overall speed-up.

1. INTRODUCTION

In recent years we are witnessing an exponential increase in the amount of data available for analysis in almost all scientific disciplines. As a result, there is a high interest in machine learning methods to conduct such analysis. Among these, Neural Networks (NNs) are regarded as one of the most promising techniques. They have been successfully applied to a wide range of tasks including medical applications [1], image recognition [2] and robotics [3].

One of the main drawbacks of NNs is their high number of parameters. As a consequence, NNs require a lot of memory resources for storage and a lot of computational resources for training and forward prediction. While the training phase is usually performed on parallel computing architectures where there is an abundance of both computational and memory resources, the platforms where trained networks are deployed usually have more limited capabilities (e.g. mobile phones). This problem has steered attention of the research community toward reducing the memory and computation requirements for trained NNs.

A promising research direction is the one of quantized neural networks (QNNs). The central idea to QNNs is to compress the parameters of the network from their float representation to a light-weight one based on quantization bins. The parameter space is divided into a predefined number of bins and each parameter float value is mapped to a bin. The number of bins trades-off the accuracy versus the gain in memory and computation requirements.

In this work we present an optimized implementation of a QNN for the forward prediction on the MNIST data set that makes use of 4-bits quantization.

Related work. The research regarding NNs compression focuses mostly on memory requirements. Thus the performance benefits that can be obtained as a by-product of compression are almost unexplored. Hence contributions in the literatue are mostly on the algorithmic side rather than on the code optimization one. For example, [4] use k-means clustering to reduce the size of a convolutional NN. The work of [5] exploits the over-parametrization of NNs by randomly grouping parameters by means of a hashing function. In [6] the authors reduce the size of the convolution filters of a NN using low-ranking approximation methods. More closely related to our method are the works that explicitly reduce the number of bits used to represent the weights in a NN. Among these [7] propose a high-accuracy 4-bits quantization scheme for recurrent NNs, a type of NN that is notorious for low prediction performance when quantized. However, their open source implementation is not optimized for performance. In [8] present a high-performance implementation of one bit QNNs optimized for Graphical Processing Units (GPUs) is presented. The authors of [9] introduce an implementation of an 8-bits QNN for speech recognition optimized for CPUs.

We present a high-performance implementation of 4-bits QNN optimized for CPUs. While this level of quantization can yield substantial improvements in memory and computation requirements, it presents implementation challenges due to the byte addressability of most computer memories and to the lack of built-in data type for 4-bits integers.

2. BACKGROUND: NNS AND QNNS

In this section we formally introduce NNs and QNNs and relative notation.

Artificial neural networks. An artificial neural network (NN) is non linear map from an input vector $\mathbf{x} \in \mathbb{R}^{d_i}$ to an output vector $f(\mathbf{x}) = \mathbf{y} \in \mathbb{R}^{d_o}$. A one-layer NN implements this mapping by composing a linear transformation $\mathbf{a} = \mathbf{W}\mathbf{x} + \mathbf{b}$ with a non linear one $\mathbf{y} = \phi(\mathbf{a}) = \phi(\mathbf{W}\mathbf{x}+\mathbf{b})$. The matrix $\mathbf{W}$ is called *weight matrix*, the vector $\mathbf{b}$ is called *bias vector* and the non-linear transformation $\phi(\cdot)$ is called *activation function*. In a multi layer NN the mapping f is implemented by composing a sequence such layers, i.e. by feeding the output of a layer as input to the subsequent one:

$$\mathbf{y} = \phi_1(\mathbf{W_1}\phi_\mathbf{2}(\mathbf{W_2}\phi_\mathbf{3}(\cdots) + \mathbf{b_2}) + \mathbf{b_1}). \qquad (1)$$

Quantized neural network. A quantized neural network (QNN) is a NN that uses low precision weight matrix and bias vector. Formally, given a one-layer NN with parameters $\{\mathbf{W}\}, \{\mathbf{b}\}$ and activation functions $\phi(\cdot)$, its quantized implementation applies the linear transformation $\mathbf{a} = \mathcal{Q}(\mathbf{W})\mathcal{Q}(\mathbf{x}) + \mathcal{Q}(\mathbf{b})$ and the non-linear transformation $\mathbf{y} = \phi(\mathbf{a})$, where $\mathcal{Q}(\cdot)$ is the quantization function that we introduce in the following section. Similarly to the standard case, multi-layer QNNs are implemented by composition of individual layers as in eq. (1).

Matrix quantization. For a matrix $\mathbf{A}$, the function $\mathcal{Q}(\mathbf{A})$ returns a low precision encoding of the matrix $\mathbf{A}$. It first computes the minimum entry (mn) and the maximum entry (mx) of the matrix $\mathbf{A}$. Then, given k bits it builds a linear binning of the continuous interval $[mn, mx]$ into 2^k bins. The bin size of the quantization $\Delta(\mathbf{A})$ is then

$$\Delta(\mathbf{A}) = \frac{mx - mn}{2^k}$$

To insure that the value 0 is represented exactly as a bin value, its index is computed as

$$z(\mathbf{A}) = sat([-mn/\Delta(\mathbf{A})])$$

where the brackets $[\cdot]$ stand for the rounding to the closest integer and the $sat(\cdot)$ function saturates an integer value into the integer value representable with k bits, hence it reads $sat(n) = \max(0, \min(n, 2^k))$. The bin values are then $\{(i - z(\mathbf{A}))\Delta(\mathbf{A}),\ i = 0, \ldots, 2^k - 1\}$. Then every entry A_{ij} is quantized to the closest bin value. The quantize matrix $\mathcal{Q}(\mathbf{A})$ and the quantized integer matrix $\tilde{\mathcal{Q}}(\mathbf{A})$ have respectively the bin value and the bin index as their ij entry. Note that the matrix $\mathcal{Q}(\mathbf{A})$ is a real-valued matrix, while $\tilde{\mathcal{Q}}(\mathbf{A})$ is k-bit integer valued, and also that the following holds:

$$\mathcal{Q}(\mathbf{A}) = (\tilde{\mathcal{Q}}(\mathbf{A}) - z(\mathbf{A})\mathbf{J})\Delta(\mathbf{A}) \qquad (2)$$

where $\mathbf{J}$ is a matrix with all entries equal to one. The algorithm to compute $\tilde{\mathcal{Q}}(\mathbf{A})$ is showed in algorithm 1.

Algorithm 1 Quantize

```
1: compute $mn = \min A_{ij}$ and $mx = \max A_{ij}$
2: $\Delta = \frac{mx-mn}{2^k}$.
3: $z = -mn/\Delta$
4: for $i, j = 1, \ldots N$ do
5:     $\tilde{\mathcal{Q}}(\mathbf{A})_{ij} = saturate([A_{ij}/\Delta + z])$
```

Quantized Matrix-Matrix Multiplication. Given two matrices $\mathbf{L}$ and $\mathbf{R}$, we want to compute the product $\mathcal{Q}(\mathbf{L})\mathcal{Q}(\mathbf{R})$. Using eq. (2) and we write the product as

$$\begin{aligned}&\mathcal{Q}(\mathbf{L})\mathcal{Q}(\mathbf{R}) = \\ &\Delta(\mathbf{L})\left(\tilde{\mathcal{Q}}(\mathbf{L}) - z(\mathbf{L})\mathbf{J}\right)\left(\tilde{\mathcal{Q}}(\mathbf{R}) - z(\mathbf{R})\mathbf{J}\right)\Delta(\mathbf{R}).\end{aligned} \qquad (3)$$

Inverting the equation 2 we can then obtain the k-bit integer valued product matrix as

$$\tilde{\mathcal{Q}}(\mathbf{LR}) = sat([\frac{1}{\Delta(\mathbf{LR})}\mathcal{Q}(\mathbf{L})\mathcal{Q}(\mathbf{R}) + z(\mathbf{LR})\mathbf{J}]) \qquad (4)$$

The algorithm for Quantized Matrix Matrix Multiplication (QMMM) is shown in algorithm 2.

Algorithm 2 QMMM

```
1: compute $\mathcal{Q}(\mathbf{L})\mathcal{Q}(\mathbf{R})$ as in eq. (3)
2: compute the k-bit integer matrix $\tilde{\mathcal{Q}}(\mathbf{LR})$ as in eq. (4)
```

3. PERFORMED OPTIMIZATION

In this section we propose an optimized implementation of the quantization and Quantized Matrix-Matrix Multiplication (QMMM) functions introduced in section 2 that uses four bits compression. We start by presenting the data structure that we use for our baseline implementation. We continue by analysing the bottlenecks of this implementation and by proposing a set of solutions to achieve a higher performance.

Baseline implementation. We presented the algorithm at the base of a straight forward implementation of a simple QNN using a k-bits compression scheme in section 2. Here we introduce the data structure necessary to a naive implementation in case $k = 4$. Using a 4-bits compression scheme presents challenges due to the byte addressability of the computer memory and to the lack of a built-in 4-bits integer data type. As a consequence, we have to define our custom data structure. One way of operating on entities that require less than a byte for storage is to use structs in combination with bit fields. Nevertheless, byte addressability of the memory does not make it possible to load or store less

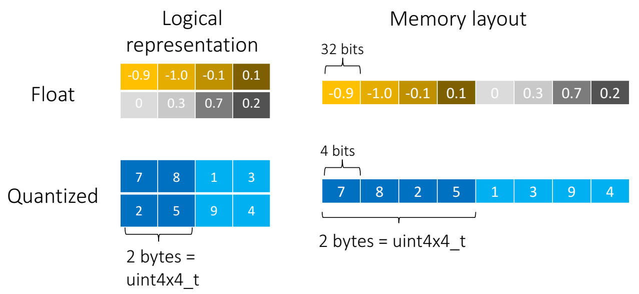

Fig. 1: Logical and memory layouts of a weight matrix and its corresponding 4-bits quantized version stored as a matrix of $uint4x4\_t$. Cells that share the same color are stored in the same data structure.

than one byte at a time. Hence, using bit fields to define a custom data type that stores a single 4-bits integer is wasteful. In the proposed solution we define the $uint4x4\_t$ data structure. This is a 2 bytes struct that exploits bit fields to store four integers of 4 bits each. Given a $n \times m$ weight matrix $\mathbf{A}$ of floats, its 4-bits quantized counterpart $\tilde{\mathcal{Q}}(\mathbf{A})$ is stored as an $\frac{n}{2} \times \frac{m}{2}$ matrix of $uint4x4\_t$. Because of this design choice, we assume both n and m to be even. In particular, the element (i, j) of $\tilde{\mathcal{Q}}(\mathbf{A})$ contains the quantized representation of the elements of $\mathbf{A}$ at the following indices: $(2i, 2j)$, $(2i, 2j+1)$, $(2i+1, 2j)$, $(2i+1, 2j+1)$. The relation between the logical layout and the memory layout for the original matrix $\mathbf{A}$ and its quantized version that uses $uint4x4\_t$ data structure, $\tilde{\mathcal{Q}}(\mathbf{A})$, can be seen in fig. 1. The difference between the memory layout of the $\mathbf{A}$ and $\tilde{\mathcal{Q}}(\mathbf{A})$ will play an important role in the vectorization of the code. The reason why we pack four integers in one struct instead of two is that it allows us to define a data type that is more convenient for QMMM. Indeed the definition of MMM applies as well for a matrix of $uint4x4\_t$, where the product and the sum of $uint4x4\_t$ elements are those of a 2x2 matrix. Then all the optimization techniques available for float MMM can be applied in $uint4x4\_t$ MMM.

Operation count optimization. The first optimization we present concerns the reduction of redundant computation. Using simple linear algebra, we can rewrite eq. (3) as:

$$\mathcal{Q}(\mathbf{L})\mathcal{Q}(\mathbf{R}) = \Delta(\mathbf{L})\Delta(\mathbf{R})(\tilde{\mathcal{Q}}(\mathbf{L})\tilde{\mathcal{Q}}(\mathbf{R}) - z(\mathbf{L})\mathbf{J}\tilde{\mathcal{Q}}(\mathbf{R}) + -z(\mathbf{R})\tilde{\mathcal{Q}}(\mathbf{L})\mathbf{J} + z(\mathbf{L})z(\mathbf{R})\mathbf{J}\mathbf{J}). \quad (5)$$

This formulation of the QMMM makes it easy to see its redundant computation. The result of the product $\mathbf{J}\tilde{\mathcal{Q}}(\mathbf{R})$ contained in the second term inside the parenthesis is a matrix that has all rows equal to each other. In particular the i^{th} term of any such row is the sum of the elements of the i^{th} column of $\tilde{\mathcal{Q}}(\mathbf{R})$. As a consequence, one can compute such row only once and save computation. In particular, this means we move ops from the inner most loop of the QMMM to the second inner most loop. Thus, the count of such ops grows quadratically with the size of the weight matrix rather than cubically. A similar reasoning can be applied to the third and fourth term inside the parenthesis.

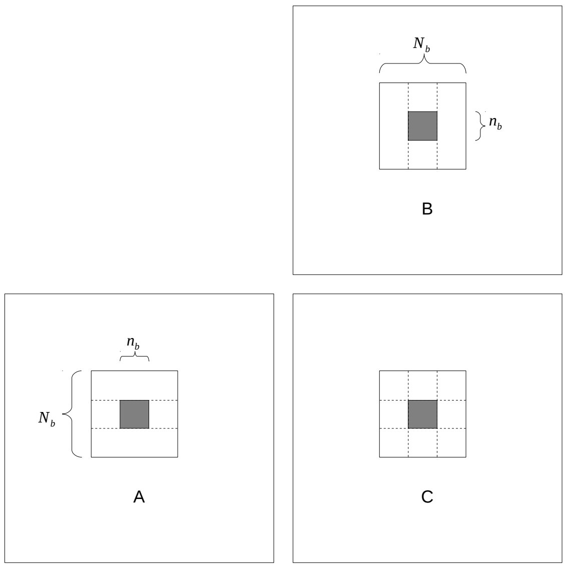

Fig. 2: Blocking parameters for the QMMM

Memory optimization. Another important optimization of our implementation is related to the efficient use of the memory. To increase the temporal locality of the QMMM kernel, we exploit the well known blocking strategy, both for cache and for register. The parameter of the blocking are the cache-block size N_b and the register-block size n_b, as outlined in fig. 2

Vectorization. Finally, a critical aspect of our implementation concerns the vectorization of the code. The gain in performance that can be expected vary depending on what type of data a function operates on. For example, if we consider the functions needed to implement the quantization step, we can expect to observe a gain in performance up to $\times 8$ because an AVX register can operate on 8 floats simultaneously. On the other hand, when we consider the QMMM, in principle we can expect a gain in performance up to $\times 16$. This is because, in order to have the representation power that is necessary to compute correctly the dot products that appear in the term $\tilde{\mathcal{Q}}(\mathbf{L})\tilde{\mathcal{Q}}(\mathbf{R})$ of eq. (5), we use 16-bits integers as accumulators. As a consequence, our implementation of quantization and QMMM is divided into sub-functions that try to keep as separate as possible the integer computation from the float computation. Among them, the most important are:

- *round-sat*: computes the round and saturation operation used in line 5 of algorithm 1 and line 2 of algorithm 2.

Function	Ops count (int or float)
round-sat	$5N^2$ FLOPS
quantize	$7N^2$ FLOPS
qmmm_kernel	$2N^3$ IOPS
trick	$2N^2 + 2N$ IOPS
add_trick	$3N^3$ IOPS

Table 1: Cost analysis

- *quantize*: given a weight matrix $\mathbf{A}$ performs the operation necessary to compute $\tilde{\mathcal{Q}}(\mathbf{A})$ except for rounding and saturating (e.g. computing mn, mx, $\Delta(\mathbf{A})$).
- *qmmm_kernel*: computes the $uint4x4\_t$ Matrix Matrix Multiplication $\tilde{\mathcal{Q}}(\mathbf{L})\tilde{\mathcal{Q}}(\mathbf{R})$.
- *trick*: computes the sum over columns and rows of $\tilde{\mathcal{Q}}(\mathbf{R})$ and $\tilde{\mathcal{Q}}(\mathbf{L})$ respectively. As explained before, this reduces the overall ops count of QMMM.
- *add_trick*: adds the appropriate element of the row and the columns computed by the *trick* function to the result of the dot product computed by *qmmm_kernel*.

Table 1 summarizes the cost in FLOPS or IOPS of each of the functions above.

Among the numerous advantages in terms of performance that come from using 4-bits compression, there are also drawbacks due to the memory layout of the $uint4x4\_t$ data structure shown in fig. 1. For example, listing 1 shows the overhead in terms of masking, shifting and blending that is required to load two rows of the quantized weight matrix.

```
__m256i tmp = _mm256_loadu_si256((__m256i const *)a);

// Mask for odd indices in memory
__m256i odd = _mm256_and_si256(tmp,_mm256_set1_epi8
      (15));

// Mask for even indices in memory
__m256i even = _mm256_and_si256(tmp,_mm256_set1_epi8
      (240));

// Shift and blend to recover rows
b_mask = _mm256_set1_epi16(32768);
*b1 = _mm256_blendv_epi8(odd,_mm256_slli_epi64(even,
      4),b_mask);
*b2 = _mm256_blendv_epi8(_mm256_srli_epi64(odd, 8),
      _mm256_srli_epi64(even, 4),b_mask);
```

Listing 1: Load of two $uint4x4\_t$ rows.

4. EXPERIMENTAL RESULTS

In this section we evaluate empirically the optimizations outlined in section 3. Every code version is tested for correctness on a small size example with hand-computed output and on several large-size random instances with output given by the naive implementation.

Experimental setup. For the empirical evaluation of the code, we use a Skylake processor (3.5 GHz, L1 cache 128 KB, L2 cache 1 MB, L3 cache 6 Mb). The compiler used is g++ with flags ”-O3 -fno-tree-vectorize -march=native -mavx”. The matrix size varies in the range $[30, 1000]$.

Results: operational count optimization. We first evaluate the gain the in runtime due to the optimization in the operational count performed with the trick as in eq. (5). From fig. 3 we can see the decrease in performance of the trick version with respect to the naive implementation. Figure 4 shows that the operation count optimization gives an overall speed-up of 15% (this is the difference in run time between *naive* and *naive_trick*). In fig. 5 we can see the contribution of each function of the vanilla implementation of the pipeline to the overall runtime. In the following the trick version is further optimized.

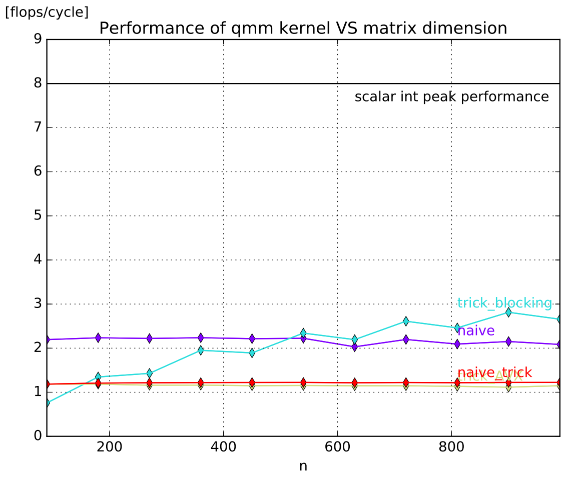

Fig. 3: Performance plot for the QMM kernel

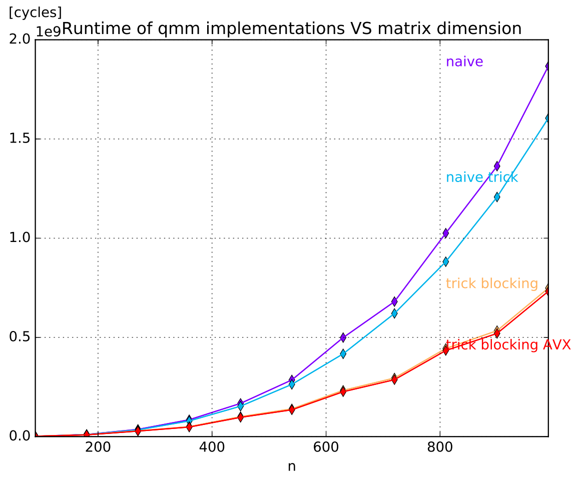

Fig. 4: Runtime plot of the overall pipeline

Results: blocking for MMM. The blocking parameter used is $N_b = 30$ for cache blocking and $n_b = 3$ for register

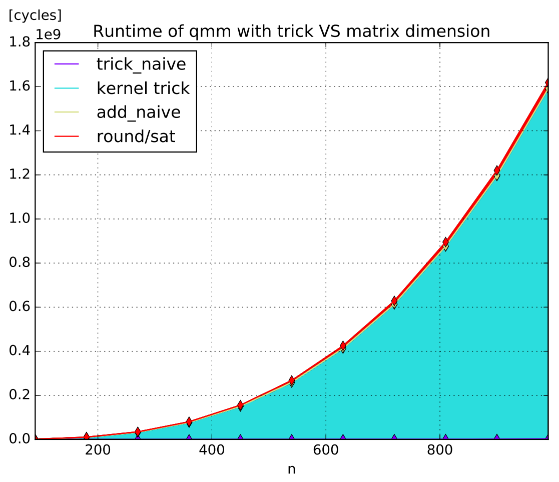

Fig. 5: Contribution of each naive-version sub-function in the overall runtime.

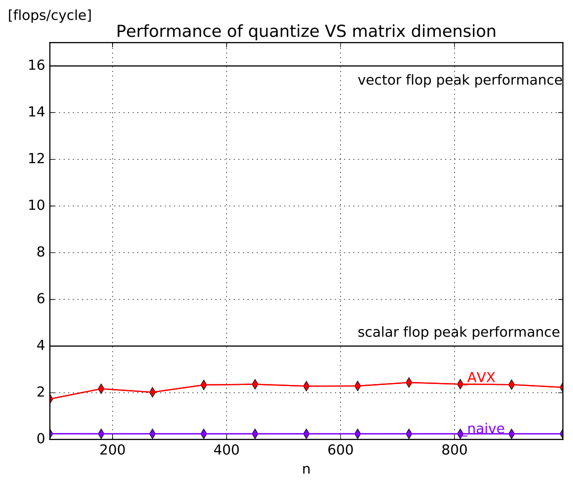

Fig. 7: Performance plot of the function *quantize*

blocking, as described in section 3. Figure 3 shows the gain in performance with respect to the *naive_trick* implementation. The performance gain for large large n is approximately 2X. Note that the blocking parameter for cache and for register are not fine tuned, so a further speed-up could be possible.

Results: vectorization. In this paragraph we evaluate the performance gain thanks to vectorization for *trick_vector*, *quantize*, *add_trick_vector* and *round_saturation* as outlined in section 3. Figure 6 and fig. 8 show a linear increase of the performance for small instances. This is due to a border effect. The instance size in the performance plot are not in general a multiple of the vector size (16X16 $Bits$), hence for small instance size the scalar computation is be non-negligible. Moreover the scalar contribution decreases linearly with n.

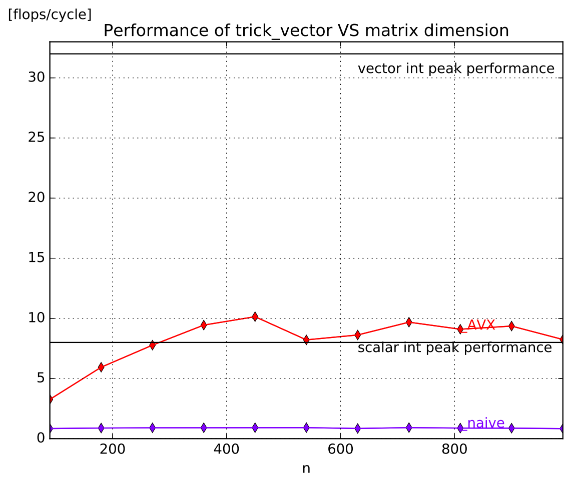

Fig. 8: Performance plot of the function *trick_vector*

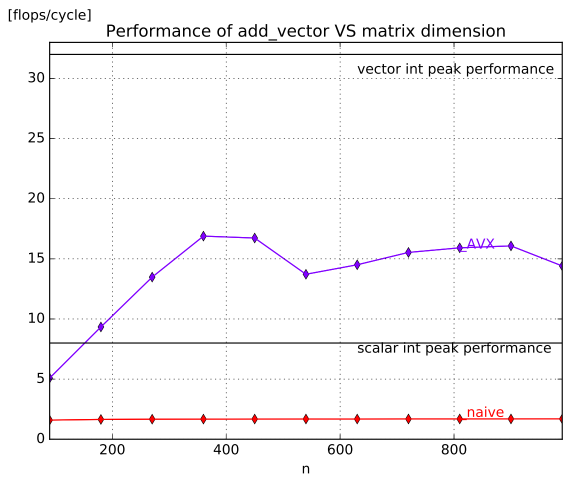

Fig. 6: Performance plot of the function *add_vector*

The maximal gain in performance that the vectorization can allow for the functions *trick_vector* and *add_trick_vector* is 16X, that is the size of the accumulator vector for 16 bit integers. The measured performance gain are respectively 9.8X and 8.5X.

Figure 7 and fig. 9 show the performance plot for the functions *quantize* and *round_saturation*. The measured performance gain are respectively 9.2X and 14.2X.

In fig. 4 we can see the runtime plot for the whole pipeline. The implementation *trick_blocking_AVX* is obtained combining the *QMM_kernel_blocking* with the vectorized implementation of all the other sub-functions. The overall speedup is 2.5X.

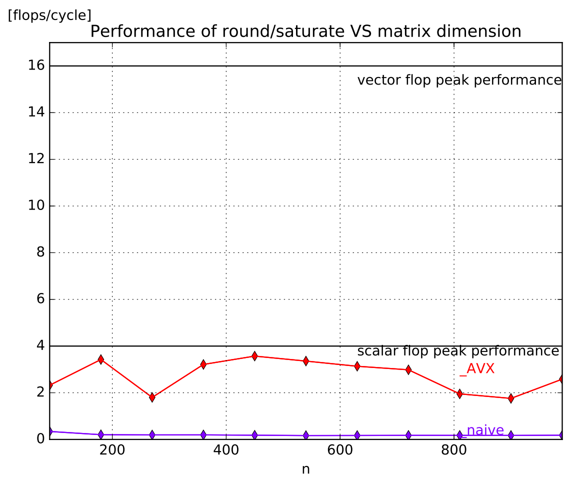

Fig. 9: Performance plot of the function *round_saturation*

5. CONCLUSIONS

The experimental results presented in the previous sections outline the benefits of performing vectorization and matrix blocking to achieve a significant speedup over a naive implementation. The key ingredient in the approach outlined above lies in the decision to map a tile of 2×2 floats to a single 16-bit integer (using a struct of 4 members with 4 reserved bits respectively). Through this construction, the algorithms outlined in the previous are accessible in a convenient form that lends itself to simply vectorize the load, quantize, round and saturate functions, while also enabling a simplified blocking scheme for quantized matrix multiplication.

6. REFERENCES

[1] Filippo Amato, Alberto Lpez, Eladia Mara Pea-Mndez, Petr Vahara, Ale Hampl, and Josef Havel, "Artificial neural networks in medical diagnosis," *Journal of Applied Biomedicine*, vol. 11, no. 2, pp. 47–58, 2013.

[2] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," in *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds., pp. 1097–1105. Curran Associates, Inc., 2012.

[3] Shixiang Gu, Ethan Holly, Timothy Lillicrap, and Sergey Levine, "Deep Reinforcement Learning for Robotic Manipulation with Asynchronous Off-Policy Updates," *arXiv:1610.00633 [cs]*, Oct. 2016, arXiv: 1610.00633.

[4] Yunchao Gong, Liu Liu, Ming Yang, and Lubomir Bourdev, "Compressing Deep Convolutional Networks using Vector Quantization," *arXiv:1412.6115 [cs]*, Dec. 2014, arXiv: 1412.6115.

[5] Wenlin Chen, James T. Wilson, Stephen Tyree, Kilian Q. Weinberger, and Yixin Chen, "Compressing Neural Networks with the Hashing Trick," *arXiv:1504.04788 [cs]*, Apr. 2015, arXiv: 1504.04788.

[6] Emily L Denton, Wojciech Zaremba, Joan Bruna, Yann LeCun, and Rob Fergus, "Exploiting Linear Structure Within Convolutional Networks for Efficient Evaluation," in *Advances in Neural Information Processing Systems 27*, Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, Eds., pp. 1269–1277. Curran Associates, Inc., 2014.

[7] Qinyao He, He Wen, Shuchang Zhou, Yuxin Wu, Cong Yao, Xinyu Zhou, and Yuheng Zou, "Effective Quantization Methods for Recurrent Neural Networks," *arXiv:1611.10176 [cs]*, Nov. 2016, arXiv: 1611.10176.

[8] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio, "Binarized Neural Networks," in *Advances in Neural Information Processing Systems 29*, D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, Eds., pp. 4107–4115. Curran Associates, Inc., 2016.

[9] Vincent Vanhoucke, Andrew Senior, and Mark Z. Mao, "Improving the speed of neural networks on CPUs," 2011.