Skipping CNN convolutions through efficient memoization

Rafael Fão de Moura[1], Paulo C. Santos[1], João Paulo C. de Lima[1], Marco A. Z. Alves[2], Antonio C. S. Beck[1], Luigi Carro[1]

[1] Informatics Institute - Federal University of Rio Grande do Sul - Porto Alegre, Brazil
[2] Department of Informatics - Federal University of Paraná - Curitiba, Brazil

Abstract. Convolutional Neural Networks (CNNs) have become a *de-facto* standard for image and video recognition. However, current software and hardware implementations targeting convolutional operations still lack embracing energy budget constraints due to the CNN intensive data processing behavior. This paper proposes a software-based memoization technique to skip entire convolution calculations. We demonstrate that, by grouping output values within proximity-based clusters, it is possible to reduce by hundreds of times the amount of memory necessary to store all the tables. Also, we present a table mapping scheme to index the input set of each convolutional layer to its output value. Our experimental results show that for a YOLOv3-tiny CNN, it is possible to achieve a speedup up to 3.5× while reducing the energy consumption to 22% of the baseline with an accuracy loss of 7.4%.

Keywords: Convolutional Neural Networks · Computation reuse · Memoization.

1 Introduction

Supported by advancements in machine learning algorithms, Convolutional Neural Networks (CNNs) have been broadly used in modern applications, such as image classification [16], speech recognition [6], and natural language processing [3] tasks. In addition to the algorithmic advances, efforts in architectural research have also contributed to make feasible the employment of CNNs, ranging from General Purpose Processors (GPPs) and Graphics Processing Units (GPUs) to custom accelerator designs [19, 2, 18]. Meanwhile, an increasing interest in migrating the execution of CNNs to embedded systems is leveraged by the Internet of Things (IoT) era, as already seen in self-driving vehicles, audio, and image recognition software running into mobile devices. However, current hardware designs used to accelerate CNNs execution still have concerns related to energy consumption and do not entirely fulfill the energy constraints of embedded systems [9].

A significant part of the execution time of a CNN, as well as its energy consumption, is spent performing convolution operations [10]. At their core, a convolution is composed fundamentally by multidimensional dot product operations on a data streaming. Thereby, the most costly operations involved in a convolution are the data movements and the floating-point multiplications and additions [9]. Therefore, such

convolutional kernels must be accelerated, either by reducing the time of such operations or by skipping them, to improve the overall CNNs execution time.

Meanwhile, CNNs present characteristics of data compression along with their convolutional layers. For instance, to perform inference over one input image in *YoloV3-tiny* CNN [16], nearly 10 GB of temporary data is generated and processed between different layers. However, one can observe that, for a universe of N input images, the classifier vector output will be represented into 320 Bytes. Since the amount of data generated and processed is thousands of times higher than the output size, it is expected that there would be high levels of redundancy in the intermediate data. Thereby, the data redundancy can be exploited to improve performance and energy efficiency. Such behavior makes CNNs suitable for applying memoization techniques.

Memoization is an old technique, which has been used to shorten the runtime of applications [20]. This technique works by keeping in storage results of previous execution for future reuse, thus saving time and energy if the algorithmic way demands more than a lookup search. In the past years, memoization approaches have been proposed at different levels of implementation and abstraction, varying from function-level in software to Functional Unit (FU)-level in hardware [20, 5]. Employing the memoization technique can be suitable for accelerating CNNs since there are many opportunities for computing reuse. Moreover, by using memoization, both energy and performance improvements can be achieved as already presented in recent literature [9, 5]. Nevertheless, memoization-based approaches have a trade-off between performance savings and data storage size. The memoization overheads come due to storing previous results in a table and looking up that table at every next execution. Since the cost per lookup table access is proportional to its size, identifying the most suitable memoization technique is essential to achieve the best relation between performance and table access time.

To overcome the limitations mentioned above, we propose a software-based memoization technique to replace the entire convolution computations by a single lookup table search implemented as a hash table. Although there are reuse opportunities in the kernel of CNNs, we demonstrate that measuring the table sizes necessary to keep all the different output values for each convolutional layer in a *YOLOv3-tiny* CNN [16] reveals a costly amount of memory. Further, we present a methodology to reduce the output table sizes by grouping the output values into range-based clusters according to their proximity values, which drastically reduces the amount of memory necessary to keep the convolution values.

To ensure the effectiveness and correctness of computing reuse and memoization, we propose an indexing table scheme to map each set of inputs in each convolutional layer to its corresponding output value. By implementing these techniques in the open-source *Darknet* framework [15], it is possible to achieve a performance speedup up to 3.5×, while reducing the energy consumption by a factor of 4.5× by allowing an accuracy loss of 7.4% due to output values clustering.

The main contributions of this paper are listed as follows:

1. We explore computation reuse and memoization as a software-based technique to improve CNNs execution.

2. We propose a technique to reduce the amount of memory necessary to store the lookup tables by grouping output values into range-based clusters making memoization feasible for the CNNs realm.
3. In contrast to previous works, we replace entire convolution computations by lookup table search with low accuracy loss.

The rest of this paper is organized as follows: Section 2 presents a general overview of CNNs layout and processing and also discusses the state-of-art hardware and software techniques to accelerate CNNs execution. The proposed technique and their implications are presented in Section 3. Section 4 introduces the methodology, experimental setups, and results used to evaluate this work. Finally, a brief conclusion and future works are drawn in Section 5.

2 Background and related work

In this section, the basics of CNNs are presented. After, a review of the state-of-art researches regarding CNNs acceleration is done.

2.1 CNN basics

A CNN performs feature extraction using series of *convolutional layers* and then it is followed by *classification layers* that analyze features and classify input images. Figure 1 shows an example of a convolutional layer. Each layer takes as input RxC values distributed along N *channels* (or *input feature maps*) and *convolves* it with M sets of $NxKxK$ filters generating M output feature maps with $R'xC'$ values each. The filters represent the weights that are previously obtained in the training phase using a learning algorithm such as back-propagation. For each filter of the M sets, the convolution is performed by sliding the filter across the input feature map according to a stride value. At each overlapping of the filter over the input feature map, the values are multiplied and accumulated together, giving one value to the output feature map. Generally, after a convolution, an activation function is applied on the output feature maps, and, occasionally, it is also followed by a subsampling operation[1].

In general, the layout of a CNNs is composed of many layers in which the output of the previous layer is the input of the subsequent layer. As the depth of the network

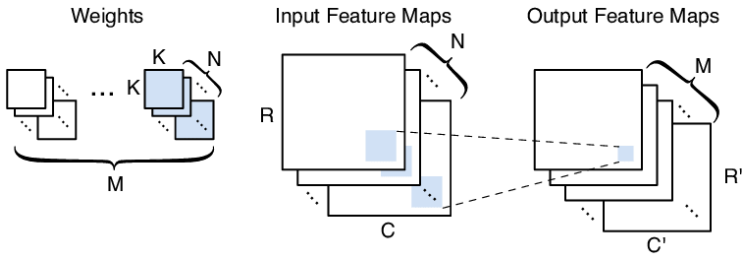

Fig. 1. Parts of a convolutional layer. Adapted from [1]

increases, higher recognition accuracy can be achieved. For instance, state-of-the-art CNNs, such as GoogLeNet [21] and YOLOv3 [16], have 20 and 75 convolutional layers, respectively.

2.2 Software-based techniques for efficient CNN execution

Past works have shown different approaches for obtaining small networks by shrinking, factorizing, or compressing pre-trained networks [4], as well as for reducing the number of computation or memory access by pruning, quantizing, or decreasing the precision of a network [8]. Although novel CNN architectures have been proposed to provide smaller and faster models for mobile and embedded vision applications, such as MobileNet [7], we focus on the optimization of existing CNN architectures to further improve performance under the same hardware.

Recently, [22] has proposed an unified framework to compress and accelerate CNN inference through product quantization. This technique exploits opportunities of replacing inner product computation by addition operation if inner products between the input and quantized weights have been computed in advance. In another study, [12] proposes modifications to Winograd-based CNN architecture to reduce the number of multiplications by combining two methods: Winograd's minimal filtering algorithm and network pruning. The authors have moved *ReLU* operation into Winograd domain to increase weight sparsity, and then pruned the weights in the Winograd domain to exploit weight sparsity.

Afterward, previous works indicate that reuse opportunities can deliver efficient CNN models by exploiting data redundancy and similarity if we allow a little accuracy loss [22, 12]. Though, most of the past works have a common drawback since they require retraining or modifying the network topology. Also, most of these characteristics related to data redundancy and operation reuse have been explored in hardware accelerators, as described in the next section.

2.3 Computation reuse-based accelerators

In the past years, several accelerators were proposed to explore data redundancy, weight/input similarity, and reuse opportunities in convolutional neural networks. The study presented by [17] reuses computation from one execution of the CNN to the next and applies the reuse taking into account the similarity found in the inputs of each layer. The authors have applied linear quantization of inputs to increase the redundancy, which favors the efficiency of the proposed mechanism, with a minimal impact on accuracy. [5] exploits weight repetition to reuse CNN sub-computations (dot product results), to reduce off-chip memory reads and also to compress network model size. They propose the Unique Weight CNN Accelerator, which unifies two opportunities to eliminate multiplication and memory reads: by factorizing dot products as sum-of-products-of-sum expression, and later by reusing the partial product when the filters slide.

In another proposal, [14] introduces a methodology to replace CNN multiplication with lookup searches in an associative memory. The authors provide a theoretical analysis of the additive error and present an algorithm to minimize it. However,

the gains depend on the associative memory blocks to reduce the power and execution time of floating point units. Instead, our proposal to reuse a coarse-grained computation (a full convolution) can leverage the use of GPPs.

Following the same idea of [14], [9] explores computation reuse through a reconfigurable Bloom filter unit that supports approximate set membership queries with a tunable rate of errors to store frequent computation patterns and return the products without executing the operands on energy-intensive Floating-Point Units (FPUs). The authors implemented the Bloom filter unit with resistive memory elements to provide energy efficient storage of common multiplication patterns in each layer of a Neural Network (NN).

All these approaches that employ memoization for convolution products depend on custom hardware, and the improvement provided may not justify the costs to produce them. The main drawback of multiplication reuse resides on increased energy consumption, since a 16-bit fixed-point multiplication in 32nm is 0.4 pJ, whereas the corresponding table lookup costs 2.5 pJ, considering a 32K-entry 16-bit SRAM [5, 13]. Instead, our proposal to reuse a coarse-grained computation can provide energy savings by reducing the amount of table lookups even on GPPs. Therefore, when we master the technique of reusing convolutional operations, the benefits come virtually for free, and it is a much easier way than try to brute force optimize the hardware.

3 Proposed approach

In this section, we explore computation reuse opportunities in CNNs by replacing the execution of entire convolution calculations by memory lookups. Based on this idea, we propose a hash table scheme to associate an arbitrary input with its corresponding output value for each layer's filter. Further, we measure the amount of memory necessary to store all the tables, and we develop a proximity range-based clustering mechanism to reduce the table sizes making feasible the employment of memoization in the CNN realm.

3.1 Replacing CNN calculations with memory accesses

Since data redundancy and data similarity are found in the inputs, weights and feature maps for inferring a batch of images, the memoization technique can be considered as an alternative for the convolutional execution model. To avoid costly multiplications and additions, we associate every pair of input (I_{i}) and weight (W_{k}) producing a single output (O_{ik}) of a feature map in lookup tables. The implementation of a CNN with a reduced number of convolution operations relies on an offline memoization mechanism that generates these tables to be used at running time.

Figure 2 illustrates both the offline analysis and online execution mechanisms. The flowchart of the memoization *Offline analysis flow* described in Figure 2 comprises two main execution blocks: the *CNN execution*, and the *Memoization profiler*. First, we run all the memoization training set of images, and we store the inputs and their corresponding outputs for all convolutional layer's filters.

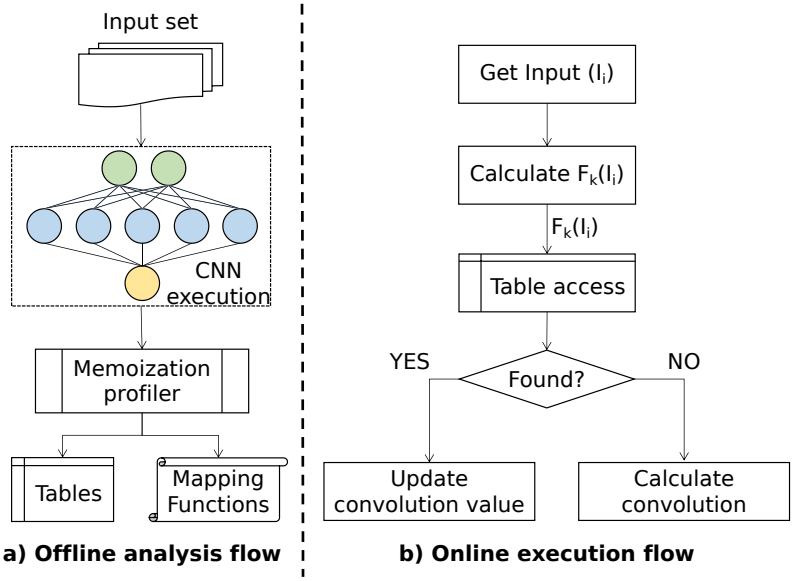

Fig. 2. The proposed approach depicted in two flowcharts: a) offline analysis and online execution flow.

As soon as we have gathered all the convolutional values $\{I_i, W_k, O_{ik}\}$ from the execution of a memoization set, the *Memoization profiler* processes the collected data and generates two output files: the output *Tables* and the *Mapping Functions* files. The output *Tables* are hash tables that enclose all the convolutional values generated during the *CNN execution*. The *Mapping Functions* are indexing functions in charge of coordinating the appropriate mapping for any input to its corresponding output. From the set theory perspective, the convolution operation regarding each weight set and input can be seen as a bijective function $F(I_i, W_k)$ that convolves an input I_i over the filter weight W_k. To ensure the correct index mapping for any input to its corresponding output, we create a function that reflects the same bijective function behavior in the convolution operation.

Now that we have the hash tables with convolution values in memory, the execution flow of each convolutional layer follows the *Online execution flow* described in Figure 2. First, the input is read, and we apply a Mapping Function to calculate a table index. As soon as we have a table index associated with the input value, we try to retrieve the entry for a convolution output from the hash table. If an entry is found in the table, the corresponding position of the feature map is updated with the value retrieved. Otherwise, we must calculate the convolution value associated with the current input.

Though, it is only viable to memoize a subset of a large dataset like MSCOCO, which provides a representative part of the convolutional operations. Even using a memoization set of 500 random images, the offline mechanism generates huge tables, which requires up 12 GB of storage memory.

3.2 Redundant data on CNNs

Storing all the outputs for any memoization set as presented in the previous section can be a limiting factor for this implementation since it is a one-to-one relationship of (I_i, W_k) and O_{ik}, which may introduce repeated entries in the lookup table. However, inputs tend to present redundancy within a single picture and even among different images that generate repeated convolution results. The goal of exploiting this inherent redundancy is not only to reduce the table size but also to improve spatial and temporal locality, thus benefiting the execution on CPUs.

To illustrate the above-mentioned data redundancy, Table 1 shows the computation reuse percentage to run different numbers of images from MSCOCO dataset [11] along YOLOv3-tiny [16] convolutional layers. For instance, by analyzing a single image (second column), the first convolutional layer presents up to 26 percent of the convolutions that can be reused from previous convolutions. That is, the same pattern of input I_i and weight W_k are repeatedly found in the first layer producing the output pattern of O_{ik}, which can be replaced by a single table search. Table 1 also indicates that, as the number of images in the batch increases, the reuse ratio for each convolutional layer grows too. Such behavior is essential to provide scalability to this CNN implementation. Thus, by eliminating the redundant entries in the offline analysis, we found that the 3 GB of storage memory is needed instead of the former 12 GB.

Table 1. Computation reuse percentage running different numbers of images in YOLOv3-tiny CNN.

		Number of Images				
		1	**10**	**100**	**1k**	
Convolutional Layer	**#1**	26	34	68	94	**Compute Reuse (%)**
	#2	23	29	52	88	
	#3	21	25	44	83	
	#4	15	19	32	71	
	#5	6	11	20	56	
	#6	0	1	6	35	
	#7	0	1	9	53	
	#8	0	0	3	22	
	#9	0	1	6	40	
	#10	0	1	11	57	
	#11	0	0	2	15	
	#12	0	1	10	54	
	#13	1	5	36	81	

3.3 Range-based clustering to reduce lookup tables

As described in the previous section, we measured the amount of memory required to store all the unique pairs $\{I_i, W_k\}$ and the corresponding output O_{ik} to create the

relationship $\{I_i, W_k\} \rightarrow O_{ik}$ in the table. However, the amount of memory to store different entries remains prohibitive to CPUs, indicating that we have to find a way to reduce even more the table size. To make the proposed memoization technique feasible, we introduce a range-based clustering mechanism that groups entries by its proximity and provides smaller lookup tables.

Our technique groups convolution values within ranges in a four-step procedure, which requires modification to the *Memoization profiler* presented in Figure 2. Firstly, all the output values for each layer in the CNN are sorted in ascending order. Then, we create clusters that minimize the distance among the entries. Each number in a cluster is enclosed within an interval defined by a bottom value and a top value. The top values are determined by multiplying the bottom values by a predefined range (typically a percentage). Thirdly, we replace all the convolution values inside each cluster by their average value. Finally, as a result of the clustering, both the *Tables* and the *Mapping Functions* in Figure 2 are updated to generate the correct output values.

As a result of the mechanism proposed to reduce the table sizes, Figure 3 presents a tunable accuracy correlation among the table sizes, and proximity values, and their respective accuracy levels. For the four range distances (0.1% to 1%) illustrated in Figure 3, the table sizes vary from 220 MB to 4.35 MB in contrast to 12 GB, when none table reduction technique is employed, and 3 GB, when repeated entries are removed. As we have approximated the convolution results by reducing the *Output set* size, one can observe an accuracy drop over the original CNN detection. The normalized accuracy over the original CNN execution varies from 98.9% to 92.5% when the range distance is increased from 0.1% to 1%.

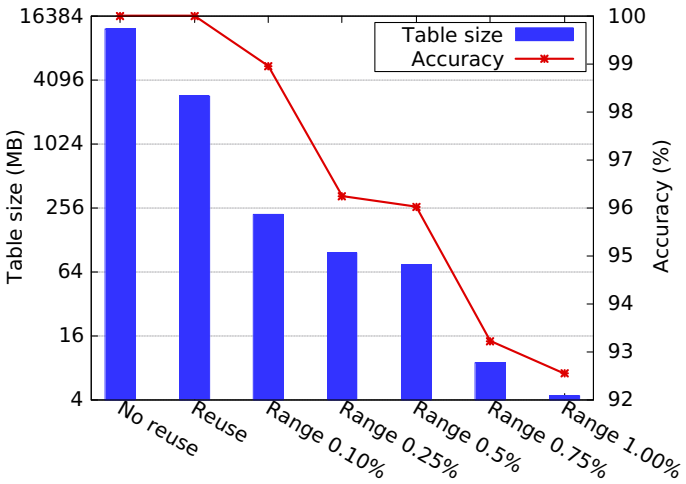

Fig. 3. Table sizes and accuracy loss.

4 Methodology and Evaluation

To evaluate the proposed mechanism, we implemented the algorithm flow described in Figure 2 in Darknet framework [15]. We used a random subset of the MSCOCO dataset [11] as input for the offline memoization technique of the YOLOv3-tiny model [16]. Then, another random subset of MSCOCO dataset was used in the execution, which runs the modified Darknet coupled with our memoization technique and YOLOv3-tiny.

Further, we ran all experimental scenarios in the system described in Table 2 to measure the execution time. We estimated the accuracy of our implementation by analyzing the *confusion matrix* of YOLOv3-tiny's predicted classes. To compare our experiments, we took the original Yolov3-tiny implementation and predictions accuracy as the baseline.

4.1 Performance and energy consumption evaluation

To obtain a significant performance on traditional CPUs, we have to find a sweet spot between CPU's memory performance and table size, as well as considering the trade-off between table size and accuracy previously illustrated in Figure 3. As our proposal of using memoization aims to replace multiplications and additions by lookup searches, it is expected an increase of main memory accesses when compared to the original CNN execution.

Figure 4 shows the absolute values for the amount of stored data and main memory access for each experiment. By storing all generated data, the ***No reuse*** experiment demands nearly 12 GB of memory space, while it requires 3.2 GB of data moved from main memory to the Last-Level Cache (LLC). On the other hand, by applying the proposed technique, the demand for memory space and data transferring decreases. By removing the redundant entries (***Reuse*** in Figure 4) we can reduce the amount of data placed in memory from 12 GB to 3 GB, though this approach still needs 2.8 GB of data transferring.

Further, by applying the proposed technique and clustering the values within the range of 1% (***Range 1.00%*** in Figure 4), the demanded amount of data drops to 4.2 MB. This aggressive reduction on the memory footprint allows more efficient

Table 2. System configuration.

CPU processor
Intel(R) Core(TM) i5-7500 CPU; 4 cores; 3.40GHz;
AVX2 Instruction Set Capable;
32KB IL1 cache; 32KB DL1 cache;
1MB L2 cache;
6MB L3 cache;
Total Power - 65W;
DRAM
DDR4 2133MHz;
Total DRAM Size 16GB;
Total Power - 6.4W;

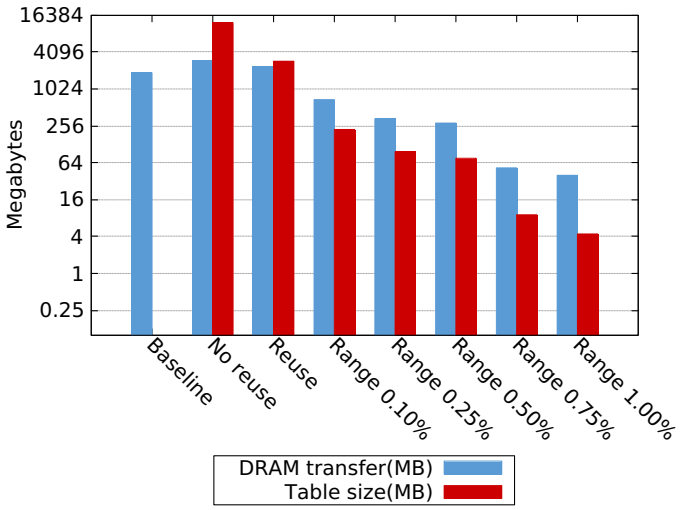

Fig. 4. Table sizes and DRAM transfers.

exploration of the cache memories, which directly reflects on the data movement behavior, thus reducing DRAM transfer from 3.5 GB to 52 MB.

Influenced by the table size and the amount of data moved from the main memory, the performance and energy consumption are presented in Figure 5. Orthogonally, the accuracy of the experimented CNN is dependent on the representativeness of data in lookup tables.

In comparison to the baseline, the ***No Reuse*** experiment presents a negative impact due to the increasing data access, as presented in Figure 5. This impact leads to a performance reduction of 23%, although energy consumption has barely decreased. Considering the ***Reuse*** experiment, due to the elimination of repeated entries, the performance is improved by 2.3×. In both cases, ***No Reuse*** and ***Reuse*** achieves 100% of original CNN accuracy, which is possible due to the maintenance of all original values. However, the amount of memory demanded by this configuration becomes a significant drawback.

The experiments presented by ***Range 0.10%***, ***Range 0.25%***, ***Range 0.50%***, ***Range 0.75%***, ***Range 1.00%*** show a performance improvement of 2.3×, 2.8×, 3.1×, 3.4×, and 3.5×, respectively. Similarly, the energy consumption is reduced in 72%, 74%, 75%, 77%, and 78%, respectively. Despite the positive impact on performance and energy, the accuracy is reduced due to the lack of data representativeness in the tables. Thus, in Figure 5 it is possible to observe the impact of different *Ranges* on accuracy, which varies from 98.9% to 92.5% over the original YOLOv3-tiny accuracy.

5 Conclusion and future work

In this work, we exploit data redundancy and proximity in neural networks leveraged by a software-based memoization technique to skip entire convolution operations.

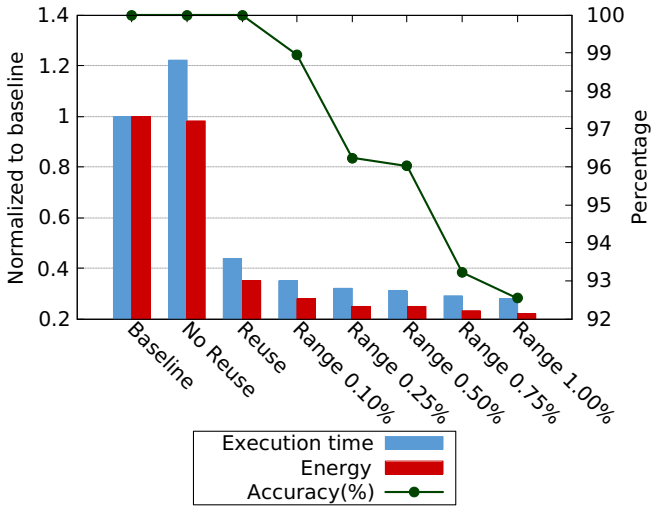

Fig. 5. Speedup and energy consumption normalized to baseline.

We designed a workflow that analyzes a training set and generates hash tables for future reuse in the running time. By using a range-based clustering, we make the size of the lookup tables smaller, at a cost to the accuracy of prediction. Our experimental results show that our implementation can improve the execution time by 3.5× over the baseline CNN running a YOLOv3-tiny model. We have shown how the performance improvement provided by our technique is strictly dependent on a tunable accuracy loss. Thus, the energy savings can be improved by a factor of 4.5×, at the cost of 7.4% of accuracy loss over the original CNN execution. In future works, we intend to investigate the proposed technique on larger CNN models, and also investigate whether the variances of data set and CNN models influence the opportunities for memoization.

References

1. Alwani, M., Chen, H., Ferdman, M., Milder, P.: Fused-layer cnn accelerators. In: The 49th Annual IEEE/ACM International Symposium on Microarchitecture. p. 22. IEEE Press (2016)
2. Choquette, J., Giroux, O., Foley, D.: Volta: performance and programmability. IEEE Micro **38**(2), 42–52 (2018)
3. Dauphin, Y.N., Fan, A., Auli, M., Grangier, D.: Language modeling with gated convolutional networks. In: Proceedings of the 34th International Conference on Machine Learning-Volume 70. pp. 933–941. JMLR. org (2017)
4. Han, S., Mao, H., Dally, W.J.: Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. arXiv preprint arXiv:1510.00149 (2015)
5. Hegde, K., Yu, J., Agrawal, R., Yan, M., Pellauer, M., Fletcher, C.W.: Ucnn: Exploiting computational reuse in deep neural networks via weight repetition. In: Proceedings of the

45th Annual International Symposium on Computer Architecture. pp. 674–687. IEEE Press (2018)
6. Hoshen, Y., Weiss, R.J., Wilson, K.W.: Speech acoustic modeling from raw multichannel waveforms. In: 2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP). pp. 4624–4628. IEEE (2015)
7. Howard, A.G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M., Adam, H.: Mobilenets: Efficient convolutional neural networks for mobile vision applications. arXiv preprint arXiv:1704.04861 (2017)
8. Hubara, I., Courbariaux, M., Soudry, D., El-Yaniv, R., Bengio, Y.: Quantized neural networks: Training neural networks with low precision weights and activations. The Journal of Machine Learning Research **18**(1), 6869–6898 (2017)
9. Jiao, X., Akhlaghi, V., Jiang, Y., Gupta, R.K.: Energy-efficient neural networks using approximate computation reuse. In: 2018 Design, Automation & Test in Europe Conference & Exhibition (DATE). pp. 1223–1228. IEEE (2018)
10. Jouppi, N.P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., Bates, S., Bhatia, S., Boden, N., Borchers, A., et al.: In-datacenter performance analysis of a tensor processing unit. In: 2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA). pp. 1–12. IEEE (2017)
11. Lin, T.Y., Maire, M., Belongie, S., Hays, J., Perona, P., Ramanan, D., Dollár, P., Zitnick, C.L.: Microsoft coco: Common objects in context. In: European conference on computer vision. pp. 740–755. Springer (2014)
12. Liu, X., Pool, J., Han, S., Dally, W.J.: Efficient sparse-winograd convolutional neural networks. International Conference on Learning Representations (ICLR) (2018)
13. Muralimanohar, N., Balasubramonian, R., Jouppi, N.P.: Cacti 6.0: A tool to model large caches. HP laboratories pp. 22–31 (2009)
14. Razlighi, M.S., Imani, M., Koushanfar, F., Rosing, T.: Looknn: Neural network with no multiplication. In: Proceedings of the Conference on Design, Automation & Test in Europe. pp. 1779–1784. European Design and Automation Association (2017)
15. Redmon, J.: Darknet: Open source neural networks in c. http://pjreddie.com/darknet/ (2013–2016)
16. Redmon, J., Farhadi, A.: Yolov3: An incremental improvement. arXiv preprint arXiv:1804.02767 (2018)
17. Riera, M., Arnau, J.M., González, A.: Computation reuse in dnns by exploiting input similarity. In: Proceedings of the 45th Annual International Symposium on Computer Architecture. pp. 57–68. IEEE Press (2018)
18. Shafiee, A., Nag, A., Muralimanohar, N., Balasubramonian, R., Strachan, J.P., Hu, M., Williams, R.S., Srikumar, V.: Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars. ACM SIGARCH Computer Architecture News **44**(3), 14–26 (2016)
19. Sodani, A.: Knights landing (knl): 2nd generation intel® xeon phi processor. In: 2015 IEEE Hot Chips 27 Symposium (HCS). pp. 1–24. IEEE (2015)
20. Suresh, A., Rohou, E., Seznec, A.: Compile-time function memoization. In: Proceedings of the 26th International Conference on Compiler Construction. pp. 45–54. ACM (2017)
21. Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., Rabinovich, A.: Going deeper with convolutions. In: Proceedings of the IEEE conference on computer vision and pattern recognition. pp. 1–9 (2015)
22. Wu, J., Leng, C., Wang, Y., Hu, Q., Cheng, J.: Quantized convolutional neural networks for mobile devices. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. pp. 4820–4828 (2016)