Proteus: Exploiting Numerical Precision Variability in Deep Neural Networks

P. Judd, J. Albericio, N. Enright Jerger, A. Moshovos
Department of Electrical and Computer Engineering
University of Toronto
Toronto, Canada
{juddpatr,jorge,enright,moshovos}@ece.utoronto.ca

T. Hetherington, T. Aamodt
Department of Electrical and Computer Engineering
University of British Columbia
Vancouver, Canada
{taylerh,aamodt}@ece.ubc.ca

***Abstract*—This work exploits the tolerance of Deep Neural Networks (DNNs) to reduced precision numerical representations and specifically, their ability to use different representations per layer while maintaining accuracy. This flexibility provides an additional opportunity to improve performance and energy compared to conventional DNN implementations that use a single, uniform representation for all layers throughout the network. This work exploits this property by proposing PROTEUS, a layered extension over existing DNN implementations that converts between the numerical representation used by the DNN execution engines and a shorter, layer specific fixed-point representation when reading and writing data values to memory be it on-chip buffers or off-chip memory. When used with a modified layout of data in memory, PROTEUS can use a simple, low-cost and low energy conversion unit.**

On five popular DNNs, PROTEUS can reduce data traffic among layers by 41% on average and up to 44% compared to a baseline that uses 16-bit fixed-point representation, while maintaining accuracy within 1% even when compared to a single precision floating-point implementation. When incorporated into a state-of-the-art accelerator PROTEUS improves energy by 14% While maintaining the same performance. When incorporated on a graphics processor PROTEUS improves performance by 1%, energy by 4% and reduces off-chip DRAM accesses by 46%.

I. INTRODUCTION

A Deep Neural Network, or DNN, is a state-of-the-art machine learning technique used for difficult tasks like speech recognition [1] and image classification[2]. DNNs are first *trained* over several known examples; then used to *classify* new inputs. Training is a more time consuming process, but is typically done once, and offline. This work focuses on Classification as it is of more interest to end-users especially for mobile applications.

Currently, classification quality and applications of classification are limited by compute performance, memory bandwidth and memory capacity [3], [4]. In the current energy-constrained semiconductor technology era, increasing the processing power of computing devices requires decreasing the energy per operation [5]. Communication accounts for 50% of the power consumption of a state-of-the-art DNN accelerator [3]. Accordingly, the goals of this work are to decrease the memory storage and overall energy needed to process DNNs.

While most general purpose software implementations use single-precision floating point [6], narrower fixed-point representations using 16 bits or fewer are often sufficient [3], [7], [8]. However, existing approaches tend to follow a *one-size-fits-all* approach by using a representation long enough to work well for *all* computations in a DNN.

In prior work, we found that the required representation length varies significantly both across networks and between the different layers of a single network [9]. As a result, the flexibility of choosing a per layer representation can open up new opportunities for energy and performance optimizations.

Another recent work also considers different precisions per-layer in to reduce the area of a custom circuit design [10]. However, this solution lacks the configurability to run different and larger networks.

We propose PROTEUS (PR) which exploits the aforementioned DNN property to reduce the data footprint of DNNs improving energy and processing capability. PR uses a different representation for each layer when storing both the inter-layer data and weights of a DNN. As a result, PR reduces on- and off-chip data traffic, improving energy, while enabling larger DNNs to run on a fixed memory budget. PR can be implemented as a dynamically configurable, layered extension over existing DNN compute engines. PR stores data in memory with the pre-selected per-layer representation length but performs computation in the native representation, using a translation layer to convert the between the two. In this work, we demonstrate PR over a GPU and a state-of-the-art DNN accelerator.

Simulation results suggest that PR can reduce memory traffic by 51% on average compared to a 16-bit baseline. For a state-of-the-art accelerator, PR reduces data traffic by 49% and overall energy by 14%. Moreover, for a conventional GPU and even without the optimized memory layout, PR is shown to reduce off-chip traffic by 46% and overall energy by 4% on average.

The rest of this work is organized as follows. Section II provides background, and summarizes our previous analysis of the precision tolerance of five neural networks. Section III presents the PR technique that reduces traffic to and from memory, leading to reduced energy consumption, higher performance and higher effective memory capacity. Section IV evaluates the proposed extension on both an accelerator and GPU architecture. Finally, Section V summarizes our findings.

II. Using A Per-Layer Numerical Representation

This section reports the minimum length fixed-point representations that can be used for a set of popular DNNs. Section II-A briefly reviews DNNs basics. Section II-B lists the DNNs used in this work and Section II-C reports the representation lengths that can be used if chosen 1) per layer, or 2) per network comparing the resulting memory footprints to a commonly used baseline 16-bit fixed-point representation.

A. Background

A DNN comprises several *layers* that process the input data in a feed-forward fashion. Each layer accepts *input data* from a preceding layer, plus a set of previously learned *weights* and produces a set of *output data*, which is the input data for the next layer. Each layer performs a fixed sequence of arithmetic operations depending on the layer's type. The layer computations exhibit data parallelism which hardware can exploit to boost performance.

Modern DNNs use a variety of layer types, but this work focuses on networks whose time is spent primarily on the convolution layers. A convolutional layer applies sets of weights (filters) to overlapping windows of the input. For each window and filter the inner product of the window data and filter weights is computed to produce a single output value. The output data dimensions then depend on the number of windows and filters.

B. Networks

We consider the five neural networks analyzed in [9], which are listed in Table I. They range from the relatively simple four layer LeNet to the 22 layer GoogLeNet which was the best network in the 2014 ImageNet Competition [11]. As in [9], we group layers in GoogLeNet into 11 layer groups, where each group uses the same precision.

C. Per-Layer Representation

In [9], we found the minimum set of precisions for each layer that could be used to store the data or weights while maintaining the network accurancy to within 1%.

The **Data** columns of Table I report the minimum length fixed point representation, in bits, used for the data of each layer of each network. For the **Weights** the lengths do not vary by much across layers and hence we choose one length for the whole network. The results suggest that: 1) the minimum length representation that can be used by each stage varies considerably within and across DNNs, and 2) overall the representations are much shorter even when compared with a 16-bit fixed-point representation.

Network	**Data (Per Layer)**	**Weights (Uniform)**
LeNet[12]	2,4,3,3	7
Convnet[13]	8,7,7,5,5	9
AlexNet[14]	10,8,8,8,8,8,6,4	10
NiN[15]	10,10,9,12,12,11,11,11,10,10,10,9	10
GoogLeNet[4]	14,10,12,12,12,12,11,11,11,10,9	9

TABLE I: Minimum precision, in bits, for data and weights for a set of neural networks.

III. Proteus

The variability in precision needed by each layer and each network can be exploited to improve the performance and energy of DNN implementations. In this work, we show how it can be used to reduce memory traffic and footprint in the memory system by proposing Proteus (Pr) a layered extension that is compatible with existing DNN implementations.

First we consider a general DNN implementation where for each layer, the input data and weights are read from memory, processed by the computational unit and the results are written to memory. In the baseline system, both memory and compute operate on data in a native representation, such as 16-bit fixed point or 32-bit floating point.

Pr adds a translation layer between memory and compute to translate data between a reduced precision representation used in memory and the native representation used in the compute engine. The reduced precision data uses fewer bits per data element allowing data elements to be packed together in memory, reducing the memory footprint. We do not modify the physical dimensions of the memory or the access width. Instead, memory traffic is reduced by accessing fewer rows to read the same amount of data. Using two different representations for computations and storage allows Pr to be implemented as a layered extension over existing compute engines.

Using a fixed-point representation for computation and storage has been considered before for DNNs; however, past work used one representation that was selected to work well for *any* DNN that may be executed. Pr allows for a different per layer storage representation.

Pr assumes that the DNN is annotated with information on the representation to be used per layer, that is, the number of bits and the fixed exponent of the fixed point representation. This can be stored with the hyperparameters[1] associated with each layer.

The conversion can be done at the level of individual memory accesses or at the level of cache blocks. For example, for a graphics processor it could either be done at the level of the cache block used by the compute engines on fill (to reduce hit time) or upon access to a line (to increase effective cache capacity).

Another indirect yet important benefit is that for a fixed amount of memory it increases the size of DNNs that can be processed. Today, memory footprint is seen as the main limiter in DNN processing during classification [4]. In modern semiconductor technologies, data movement energy tends to dominate compute energy with off-chip accesses being 2-3 orders of magnitude more expensive than floating-point addition or multiplication [16].

The remainder of this section discusses how Pr can be incorporated first in a state-of-the-art accelerator (Section III-A)

[1] hyperparameters define the input dimensions, access patterns and computation of the layer

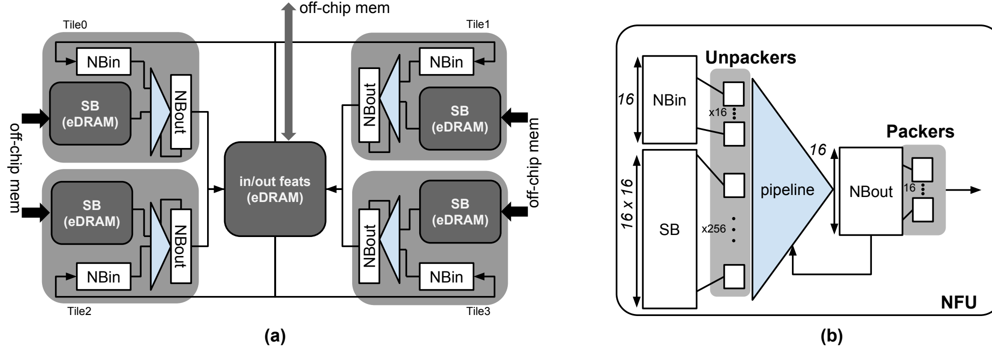
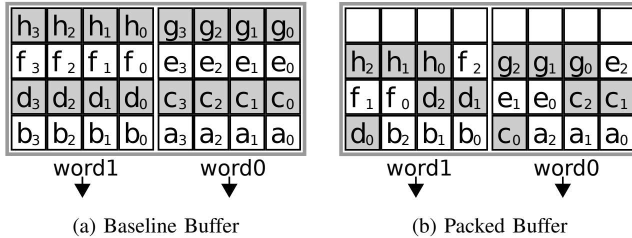


Fig. 1: a) Memory elements of an example accelerator including 4 NFUs. b) Proteus elements in a NFU.



(a) Baseline Buffer (b) Packed Buffer

Fig. 2: Buffer organization example for 4-bit words and 2 words per row comparing a baseline and packed buffer with a reduced precision of $P = 3$ bits

and then in a GPU (Section III-B).

A. Incorporating into a DNN accelerator

This section explains PR can be incorporated into a state-of-the-art DNN accelerator. Section III-A1 describes the baseline accelerator, and Section III-A2 shows how we incorporate PROTEUS.

1) Accelerator Architecture: The baseline accelerator we consider is a derivative of the *DaDianNao* accelerator. The accelerator incorporates multiple *Neural Functional Units (NFUs)*. Figure 1 shows an example design with four NFUs, highlighting the memory elements and the connections among them. Each NFU includes a 256-wide SIMD pipeline [3]. The input neurons (data) and synaptic weights (weights) are read from two buffers, *NBin* and *SB*, respectively. The output neurons and intermediate results are stored in the *NBout* buffer. The data stored in the NBout can be fed back into the processing pipeline when necessary.

Our baseline system uses 16-bit fixed point natively for both the data and weights. Every cycle 16 elements x 16 bits are read from NBin and 256 elements x 16 bits are read from SB. At its output, the NFU can write 16 x 16-bit results to NBout. NBin and NBout are SRAMs with 64 entries (2KB). SB is a 2MB eDRAM.

Our baseline design is configured to match a *node* of the *DaDianNao* system. It incorporates 16 NFUs, plus an additional shared 4MB eDRAM to hold the input and output data of the current layer (36MB of eDRAM in total). An off-chip DRAM provides the initial input and the weights.

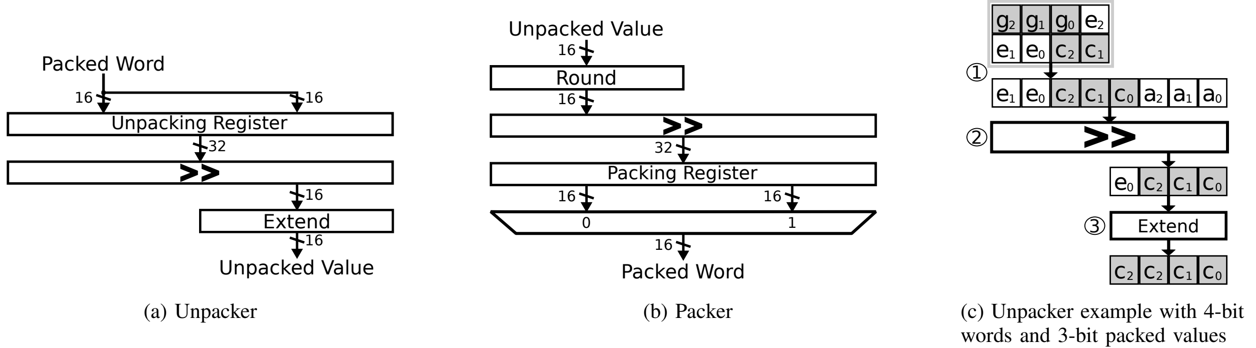
2) Proteus in the Accelerator: With PR, data and weights are stored using the storage representation in all levels of memory except in NBout as we will explain. The data and weights are converted to the internal compute representation just prior to entering the computation units. Compressing the data and weights by using a reduced precision representation will reduce the dynamic power at each memory level: the buffers, the on-chip eDRAM, and the off-chip DRAM.

3) A Different Memory Layout: By using a different layout of data and weights in storage, PR enables the use of a low-cost, low-energy unpacking unit. For ease of explanation, let us restrict attention to the weights for the time being.

In the native layout, the 256 weights that need to be processed in a single cycle appear next to each other in storage, corresponding to a row in SB, followed by the next block of 256 weights to be processed in the subsequent cycle. However, when we pack reduced precision elements into memory they become unaligned with the destination pipeline input and we need additional logic to align them. Requiring different shifts for each value and the potential for very long lateral wires could lead to a significant overhead.

To address this, PR organizes the data differently. Starting with the native representation, we group the weights into 256 ”virtual columns”, each corresponding to one input of the pipeline. Figure 2a shows a simplified example with two 4 bit words in each row, and thus 2 vitrual columns. Each cycle, a row is read from the buffer and fed into the pipeline. So in the first cycle a is read as word0 and b is read as word1, in the second cycle, c will be read as word0 and d will be read as word1.

Figure 2b shows the memory organization in the buffers for PROTEUS when we pack 3-bit values into 4-bit words, thus every 4 rows of words in the baseline buffer can be stored in 3 packed rows. Keeping the values (a-h) in the same virtual columns as the baseline will reduce the cost of alignment. This alignment is done as part of a *translation layer* unit after each buffer that compresses and decompresses the data and weights.



(a) Unpacker

(b) Packer

(c) Unpacker example with 4-bit words and 3-bit packed values

Fig. 3: Unpacker and Packer

4) Translation Layer: The translation layer consists of two modules, the *unpacker* and the *packer*. The unpacker converts the reduced precision *P*-bit weights/inputs that are packed in the input buffers to the full precision 16-bit values used by the compute pipeline. The packer converts the results from the output buffer after the computational units to the reduced precision storage representation and packs them into 16-bit words. As we introduced in Section III-A3, we pack data within one 16-bit wide virtual column of words in the buffers, so we only need to consider the design of the unpackers and packers for one such virtual column. In the full NFU there will be 256 unpackers for SB and 16 unpackers/packers for NBin/NBout, respectively. NBout is also used to store partial sums that are fed back into the pipeline. As such it is part of the internal computation of the layer and should store data in the native precision. Output data is only packed when it is written back to the central eDRAM.

Unpacker: Figure 3a shows the unpacker for our system with words having a native precision of 16 bit. The three steps involved in unpacking are illustrated with a simplified example in figure 3c. This example shows how we would unpack the data in the word0 column of figure 2b.

1) When a new word needs to be read in, it is loaded into either the upper or lower word of the unpacking register in an alternating fashion. The register is two words wide to allow values split across two words to be rejoined. Note that we do not read in a word every cycle, only when a word contains a partial value that is needed.

2) The 32 bits in the unpacking register are rotated to align the current value with the correct position in the 16-bit output. A circular shifter is used so that values split both between bit 15 and 16 and bit 31 and 0 can be recombined.

3) Once we have the *P*-bit value in the right bit position, we need to extend it to an equivalent 16-bit fixed point value. Since we are dealing with fixed point values, this involves sign extending the most significant bit and zero extending the least significant fractional bit.

The wide bit masks are used as control signals instead of narrower encoded signals to avoid adding a decoder to each unpacker unit. Since there are many parallel unpackers operating in lock step for each buffer, they all share the same control signals and decoding can be done by a single control block per buffer.

Packer: The packer, shown in figure 3b, is mostly the reverse of the unpacker, also using a 32-bit circular shifter and packing register. One key difference is that data produced by the pipeline must be rounded to the nearest *P*-bit representation to minimize compression error. Fractional bits are rounded to the least significant bit (lsb) of the reduced precision value. If the unpacked value is outside the range of the reduced precision representation then it saturates to the maximum or minimum *P*-bit value. The rounded unpacked data is then shifted to align with the next available space in the packing register. When loading in to the packing register only *P* bits are loaded using a 32-bit mask connected to the inidivual register enables. Once a full word has been packed, it is written out.

5) Data Alignment in Memory: Under ideal circumstances, PR can reduce the memory traffic of the data to $P/16$ of the baseline. We call this fraction the **traffic ratio (TR)**. However, there are alignment constraints which prevent us from achieving this ideal traffic ratio. Our goal with PROTEUS is to improve memory efficiency without impacting performance. That means we must be able to feed the pipeline with new data every cycle. As such, each stream of values must have its first value aligned, where a stream is a set values that is both contiguous in memory and processed in sequential rows.

In 3D convolutions, due to the overlapping sliding window access pattern, a stream size is equal to the depth, d, of the 3D input data. Since data is read in 16 word rows, we must align the input data every $d/16$ rows. For example, in figure 2b where rows are 2 words, if $d = 4$ then we must align data every 2 rows. This means that values **e** and **f** must be aligned to the beginning of the next row and we get no compression.

Figure 4 shows the resulting input data compression for all the networks. For LeNet and Convnet, $d \leq 32$, so only 8 bits yield compression. For all networks, $d = 3$ in the first layer, which reduces the overall TR.

The aligned traffic ratios for the selected precisions are shown in Table II. As expected, the smaller networks see less data compression than their ideal, while the larger networks get closer to the ideal. Weights on the other hand have a stream size equal to the filter size, so the aligned TR is much closer

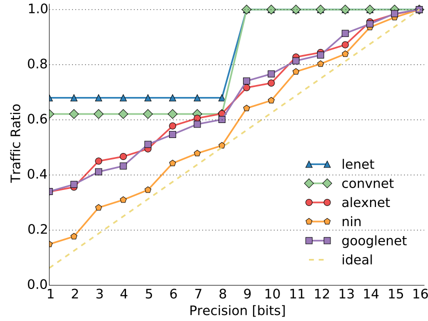


Fig. 4: Alignment overhead: Compression, measured as the traffic ratio, for all data precisions over the 16-bit baseline.

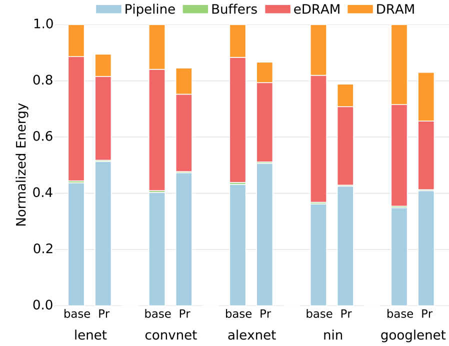


Fig. 5: Energy breakdown of each network, normalized to the baseline

	Data		Weights	
Network	Ideal	Aligned	Ideal	Aligned
LeNet	0.16	0.76	0.45	0.44
Convnet	0.48	0.61	0.56	0.56
AlexNet	0.55	0.56	0.63	0.63
NiN	0.64	0.66	0.63	0.63
GoogLeNet	0.72	0.78	0.56	0.56

TABLE II: Traffic ratio for convolution layers

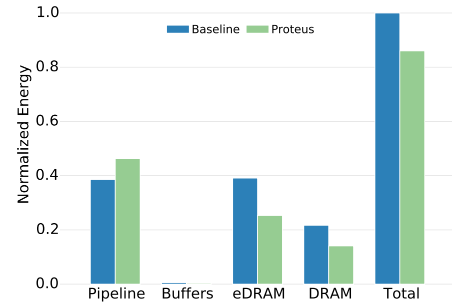


Fig. 6: Average Energy of each component, normalized to the baseline total

to the ideal.

6) Memory Layout Conversion and Execution Time: The weights are pre-specified and thus the preprocessing of the weights is a one time cost that can be done off-line. Since we get no compression on the first layer, input data can be provided as in the baseline with no preprocessing.

B. Incorporating into a GPU

We also consider incorperating PROTEUS into a GPU. This is implemented as an address remapping technique to compresses the data in the GPU's global memory. Padding is used whenever a block of words or data crosses the boundaries of a cache block. We use a packer/unpacker network comprising several multiplexers similar to what was described in Section III-A. However, since we use padding at the level of cache blocks, there is no need to shift and join portions of different cache blocks.

We apply the address remapping prior to the GPU L1 cache. However, memory coalescing [17] further limits the amount of affective compression we can achieve when aligning values within a cache line. This results in groups of P having the same effective compression as the largest P: 16-11, 10-9, 8-7.

IV. EVALUATION

This section evaluates PR. Section III-A1 studies PR in the context of the accelerator architecture, while Section IV-B shows the benefits possible when PR is employed in a modern graphics processor system.

A. Accelerator Evaluation

1) Methodology: We implemented the baseline accelerator NFU pipeline with and without the packers/unpackers in Verilog and synthesized them using the Synopsis Design Compiler vH-2013.03-SP5-2 [18] with the FreePDK 45nm library v1.4 [19]. We modeled the area and power for the SRAM buffers using CACTI v5.3 [20]. The eDRAM energy was modeled with *Destiny* [21]. The off-chip memory was modeled using DRAMSim2 [22] with 2GB of DDR3-800. Off-chip memory is not a bottleneck so we chose a lower frequency memory to be conservative.

2) Energy Savings: Figure 5 shows the total energy of each network with PR relative to the baseline (base). Overall PR yields an energy savings of up to 21% (LeNet) and 14% on average.

Figure 6 compares the energy of each component in the baseline accelerator and Proteus, relative to the total baseline energy. The added logic of the packer/unpacker in the pipeline increases power by 19%, while compressing the data and weights reduces power in the buffers by 33%, the eDRAM by 35% and the DRAM by 35%.

3) Area Overhead: The packers and unpackers increase the area of the pipeline from $1.42mm^2$ to $1.70mm^2$, an overhead of 17%. However this only results in a 1.84% overhead for the full chip.

B. GPU Evaluation

This section measures the impact on performance and energy of applying PR in deep learning neural networks implemented over GPUs.

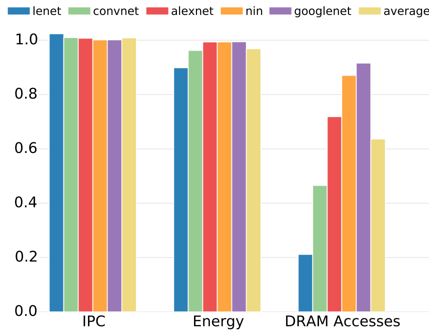


Fig. 7: IPC, Energy and DRAM accesses on a GPU

We use GPGPU-Sim v3.2.2 and GPUWattch v1.0 [23] with the baseline Fermi configuration to model and evaluate PR. We use *cuda-convnet* [13] to evaluate the modifications described in III-B on GPGPU-Sim, applying the values of P for each layer given in Table II.

For consistency, we compare the GPU implementation with a 16-bit fixed-point baseline. Since cuda-convnet uses 32-bit floating point natively, we approximate the improved system efficiency by scaling portions of the core power from GPUWattch. Specifically, we scale down the register file, integer and special functional units, pipeline, and constant dynamic energy conservatively by a factor of two. We also scale the floating point unit energy by a factor of two to approximate fix point energy.

The results for the five networks are shown in figure 7 and are normalized to a 16-bit baseline. Performance is improved by 0.2%-2% and energy is improved by 0.6-10% across the different networks. Most of the energy gains result from the reduction of DRAM accesses and the reduced shared memory access energy. While there are large reductions in global memory accesses, from 8% to 79%, the convolution layers utilize the GPU's shared memory to perform the convolution operation, which limits the benefit. NiN and GoogLeNet show the least improvement from PR due to the padding described in Section III-B.

V. Conclusion

We proposed PROTEUS, a dynamically configurable, layered extension over existing hardware that performs memory compression by leveraging the reduced precision tolerance of Deep Neural Networks with the aim of reducing overall memory traffic, memory energy and increases effective memory capacity. Using simulation, PROTEUS' benefits were demonstrated when incorporated into a DNN accelerator and a GPU.

We demostrate savings in dynamic memory energy, however there is also opportunity for static energy savings by virtue of the reduced memory footprint. While we presented PROTEUS as a hardware extension, it is likely that a variation of the approach can be implemented purely in software as well. Furthermore, it may be possible to use different representations for other, possibly finer-sized groupings of data such as portions of layers.

References

[1] G. Dahl, D. Yu, L. Deng, and A. Acero, "Context-dependent pre-trained deep neural networks for large vocabulary speech recognition," *IEEE Transactions on Audio, Speech, and Language Processing*, vol. 20, pp. 30–42, January 2012.

[2] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge," *arXiv:1409.0575 [cs]*, Sept. 2014. arXiv: 1409.0575.

[3] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam, "Dadiannao: A machine-learning supercomputer," in *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*, pp. 609–622, Dec 2014.

[4] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," *CoRR*, vol. abs/1409.4842, 2014.

[5] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pp. 365–376, ACM, 2011.

[6] I. Buck, "NVIDIA's Next-Gen Pascal GPU Architecture to Provide 10X Speedup for Deep Learning Apps." "http://blogs.nvidia.com/blog/2015/03/17/pascal/", 2015.

[7] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision," *CoRR*, vol. abs/1502.02551, 2015.

[8] M. Courbariaux, Y. Bengio, and J. David, "Low precision arithmetic for deep learning," *CoRR*, vol. abs/1412.7024, 2014.

[9] P. Judd, J. Albericio, T. Hetherington, T. Aamodt, N. E. Jerger, R. Urtasun, and A. Moshovos, "Reduced-Precision Strategies for Bounded Memory in Deep Neural Nets," *arXiv:1511.05236 [cs]*, Nov. 2015. arXiv: 1511.05236.

[10] J. Kim, K. Hwang, and W. Sung, "X1000 real-time phoneme recognition VLSI using feed-forward deep neural networks," in *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 7510–7514, May 2014.

[11] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge," *International Journal of Computer Vision (IJCV)*, 2015.

[12] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, pp. 2278–2324, Nov 1998.

[13] A. Krizhevsky, "cuda-convnet: High-performance c++/cuda implementation of convolutional neural networks." https://code.google.com/p/cuda-convnet/.

[14] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25* (F. Pereira, C. Burges, L. Bottou, and K. Weinberger, eds.), pp. 1097–1105, Curran Associates, Inc., 2012.

[15] M. Lin, Q. Chen, and S. Yan, "Network in network," *CoRR*, vol. abs/1312.4400, 2013.

[16] S. Galal, *Energy Efficient Floating-Point Unit Design*. PhD thesis, Stanford University, 2012.

[17] M. Harris, "How to access global memory efficiently in cuda c/c++ kernels." http://devblogs.nvidia.com/parallelforall/how-access-global-memory-efficiently-cuda-c-kernels/.

[18] Synopsys, "Design compiler." http://www.synopsys.com/Tools/Implementation/RTLSynthesis/DesignCompiler/Pages/default.aspx.

[19] FreePDK45. http://www.eda.ncsu.edu/wiki/FreePDK45:Contents.

[20] N. Muralimanohar and R. Balasubramonian, "Cacti 6.0: A tool to understand large caches."

[21] M. Poremba, S. Mittal, D. Li, J. Vetter, and Y. Xie, "Destiny: A tool for modeling emerging 3d nvm and edram caches," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2015*, pp. 1543–1546, March 2015.

[22] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "Dramsim2: A cycle accurate memory system simulator," *IEEE Comput. Archit. Lett.*, vol. 10, pp. 16–19, Jan. 2011.

[23] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. S. Kim, T. M. Aamodt, and V. J. Reddi, "Gpuwattch: Enabling energy optimizations in gpgpus," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, pp. 487–498, 2013.