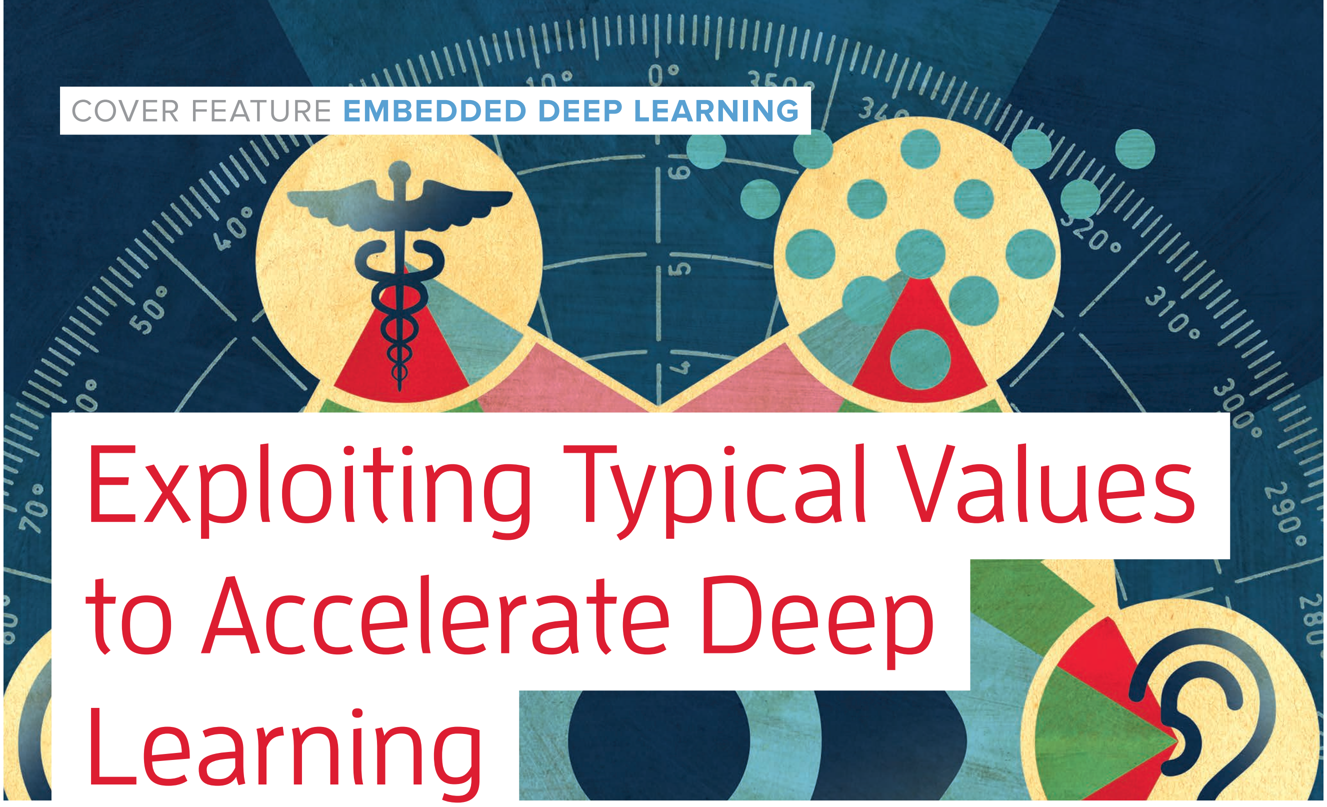


Exploiting Typical Values to Accelerate Deep Learning

Andreas Moshovos, University of Toronto
Jorge Albericio, NVIDIA
Patrick Judd, University of Toronto and NVIDIA
Alberto Delmás Lascorz, University of Toronto
Sayeh Sharify, University of Toronto
Zissis Poulos, University of Toronto
Tayler Hetherington, University of British Columbia
Tor Aamodt, University of British Columbia
Natalie Enright Jerger, University of Toronto

To deliver the hardware computation power advances needed to support deep learning innovations, identifying deep learning properties that designers could potentially exploit is invaluable. This article articulates our strategy and overviews several value properties of deep learning models that we identified and some of our hardware designs that exploit them to reduce computation, and on- and off-chip storage and communication.

Deep learning (DL) enables computing devices to "learn by example" and thus to tackle tasks that were beyond the reach of traditional computing. For example, using DL, it is reasonable today to expect that a computing device can infer what object an image or a doodle depicts. In the most commonly used form of DL, that of supervised learning, the system learns how to distinguish objects by first training over numerous known examples. By inspecting these examples, DL can "learn" how to distinguish with great accuracy whether an image that it has not seen before is that of a plane or a teapot. That is as long as our previously inspected examples contained enough images of planes and teapots.

The core building blocks of DL have been around for decades, but practical applications were limited to a few niche cases. Recently, numerous practical applications have materialized with more being demonstrated regularly. What gave rise to these "sudden" successes? The DL community has been able to collect or exploit sufficient annotated examples and also to harness the tremendous computing power of modern computing hardware to innovate further in the core machine learning building blocks and in the way they are connected. Most relevant to our discussion, shortly after 2010, the processing power of commodity computing devices, in the form of graphics processors, reached the level necessary to enable new DL applications that were previously out of reach.

While DL has seen great successes, many tasks are still out of reach and others certainly could use further refinement (e.g., autonomous driving). A clear path for further innovation in DL is to harness even more computing power to process more example data and to build more sophisticated building blocks and arrangements. As before, more processing power can remain an enabler of further innovation, but of course, cannot guarantee it.

Our expertise has been in developing performance and energy efficiency enhancing techniques for general-purpose processors. Today, such processors are at the core of all computing devices, be it server-class machines, smartphones, or embedded devices. For the past four years or so, we have been exploring hardware-level acceleration of DL applications. Our goal has been to develop hardware-level techniques that, when incorporated into next-generation hardware devices, will hopefully enable the DL community to explore even more advanced applications.

This article reviews our general approach to hardware-level acceleration of DL and highlights some of the techniques we developed. The defining characteristic of our techniques is that they are value-based. That is, they exploit properties in the data stream of DL applications by exploiting the computational structure of these applications to boost performance and energy efficiency above and beyond what is currently possible. In this article, we focus on the acceleration of inference with convolutional neural networks.

Before we attempt to highlight how value-based acceleration enhances structure-based approaches and why we chose that direction to guide our exploration, let us first review why acceleration is now receiving so much attention.

THE NEED FOR ACCELERATION

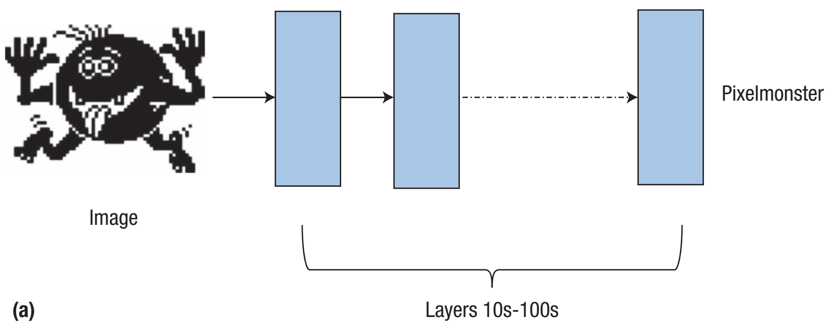
For the past three decades or so, computing hardware performance was roughly doubling every two years. A task that would take about one hour to perform on a 1985 desktop computer would complete in less than a minute or two on computer built in 1995. This exponential performance growth was fueled by Moore's law: semiconductor technology advances enabled increasingly denser and faster devices facilitating computer architecture innovations.

Unfortunately, using more and faster transistors requires more power, but operating voltage reductions have been effective at subduing the rate at which overall power increased with every generation of computing hardware. However, these performance improvement methods typically required a disproportionate increase in transistor count to deliver performance benefits and thus incurred a disproportionate power cost. As a result, in the early 2000s, processor power consumption and density surpassed practical limits. Performance scaling dramatically slowed down. Chip multiprocessors emerged promising sustained performance improvements as long as applications could be broken into threads, that is, parts that can execute mostly concurrently. Graphics processors target a certain class of such workloads that could be broken down into 1000s of threads, each executing the same code and mostly in lockstep. Computer graphics is such a data-parallel workload. As the underlying semiconductor technology scaling trends persist, the current architectural techniques are reaching their limits. Further innovation is now required to sustain performance improvements.

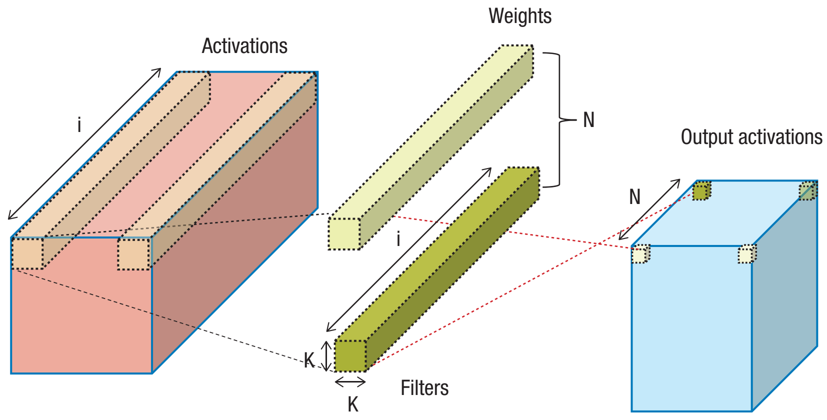
Since power is now the main constraint, further performance improvements require reducing the energy expended per operation: if each operation requires less energy, we should be able to use the abundant hardware resources to perform more operations concurrently while still staying within our power envelope. Hardware acceleration is such an approach. A hardware accelerator is a "processor" that has been specialized in part or fully for a particular task or class of tasks. Therefore, let us now take a closer look at DL to understand how specialization can improve energy per operation and thus performance.

HARDWARE ACCELERATION AND DEEP LEARNING

DL utilizes neural networks (NN). Figure 1a shows an example of a



(a)



$$\underbrace{o(y', x', n)}_{\text{Output activation}} = f(\underbrace{\sum_{x=0}^{k_{x-1}} \sum_{y=0}^{k_{y-1}} \sum_{i=0}^{N_{i-1}} \underbrace{w^n(x, y, i)}_{\text{Weight}} \times \underbrace{a(x + x' \times S, y + y' \times S, i)}_{\text{Input activation}}}_{\text{Window}})$$

(b)

FIGURE 1. (a) A feed-forward image classification neural network. (b) A convolutional layer.

feed-forward NN where several layers operate in sequence. In other NNs there is feedback among layers. Each layer accepts several input numbers and produces another set of output numbers. For image classification, the input to the first layer is an image. Presently, there are only a few different layer types with convolutional and, to a lesser extent, fully-connected layers dominating execution time in convolutional neural networks (CNNs). In our discussion, we focus on CNNs and on convolutional layers since fully connected layers can be thought of as a special case of convolutional layers.

Figure 1b shows that a convolutional layer (CVL) accepts as input a 3D array of runtime calculated values, or activations (for layer 1 these are our external input, an image) and produces an output activation 3D array. The CVL convolves the input activations with several filters, each a 3D array of predetermined values, or weights. These weight values contain the "knowledge" embedded in the NN, and they are calculated during training. They remain constant during inference, which, for an image classification task, is the process of using the network to determine what an image depicts.

A typical input or output activation array contains 1000s of values, and each layer typically applies 100s of filters each containing 100s to a few 1000s of weights. Each output activation calculation amounts to a dot product of a filter with an equally sized sub-array of the input activation array. Figure 1b shows how output activation o(x', y', n) is expressed as a function of input activations a(x, y, i) and weights w(x'', y'', i). Each dot product involves 100s to 1000s of activation and weight pairs. A constant bias is usually added at the end, and the result passes through a nonlinear activation function which produces the output activation. Several activation functions exist with the Rectified Linear Unit (ReLU) often used for image classification. ReLU converts negative activation values to zero, while allowing positive activations to pass through. To fully process an input activation array, the filters scan the input using a stride S. The input activation subarray used in each computation with a filter is called a window. In the discussion that follows we ignore the addition of the bias as it is easy to implement along with the activation function.

The Opportunity for Deep Learning Acceleration

A dot-product can be implemented as a triple-nested loop:

```
outa = 0
  for xi =  0  to K
    for yi =  0  to K
      for ii =  0  to  imax
        outa += a(x+xi,y+yi,ii) *
                w(xi,yi,ii)
```

A general-purpose processor implements these loops as several tens of machine code instructions, each typically a simple calculation or a data movement. Executing each instruction entails several actions such as fetching the instruction representation from memory, decoding it to interpret what it represents, and reading and writing several storage elements such as registers or memory. Such processors are flexible and can execute any arbitrary code fairly well. However, if all that we care about is executing dot products, this flexibility incurs significant energy and thus performance overheads. Specializing our hardware to perform dot products can drastically reduce these overheads.

Computation Structure-Based Acceleration

Specialization can exploit the computation structure of dot products. Figure 2 shows such a structure-based accelerator. It accepts 16 activations and 16 weights. It multiplies these in pairs and then reduces the 16 products using an adder tree to accumulate the result into an output register. This hardware can compute an output activation over multiple cycles. The accelerator can use several units such as this to process more activation and weight pairs per cycle. Since convolutional layers typically have many filters, each can be assigned a separate unit with all units reusing the same 16 activations. Reusing data is desirable as memory accesses are much more energy expensive than typical computations in modern semiconductor technologies.

DaDianNao is such a structure-based accelerator.[1] It takes advantage of activation reuse in convolutional layers and judiciously uses on-chip resources to balance computation and communication needs. DaDianNao contains 256 processing units, similar to that in Figure 2, organized in 16 tiles of 16 units each. Each unit can process a separate filter and in total DaDianNao computes 4K products and 256 partial dot products per cycle. Different configurations are possible and desirable depending on the application.

Value-Based Acceleration

We purposely targeted techniques that could complement structure-based acceleration: as an academic group we felt that our contribution would be more meaningful if we attempted to look further into what may be useful after structured-based approaches are perfected by industry. Drawing on our experience with general-purpose processor optimizations we decided on the following three principles: 1) Try to exploit typical execution behavior, 2) do not require NN modifications to achieve benefits, and 3) investigate in-depth specialization before trying to generalize. Here is why:

Exploiting typical behavior. Many general-processor performance techniques exploit typical program behavior. Take for example, hardware caches, a key memory access acceleration technique. In today's technology a processor can perform calculations about 100 times faster than main memory can supply the data. Unfortunately, it is not possible to build large and fast main memories. Fortunately, by exploiting common program behavior, it is possible to build memory hierarchies that behave like a large and fast memory most of the time. This is only possible because most programs exhibit memory access stream locality: they tend to access the same or nearby memory locations close in time. As a result, a cache, a small and fast memory, can expect to service many memory requests using the following strategy: keep copies of a limited number of recently accessed memory locations and those nearby. Programs do not have to exhibit locality, but most happen to do so.

Mirroring this experience with general-purpose processors, we asked

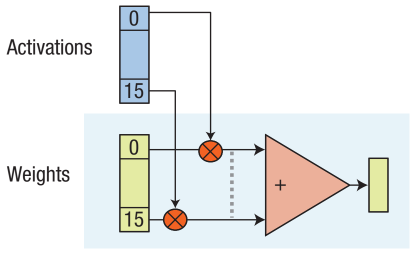


FIGURE 2. A computation-structure-based accelerator.

whether there are properties in NN execution that hardware can exploit to boost performance. We wanted to complement approaches that exploit the structure of computation and thus targeted the value stream. Between weights and activations, we decided to first target the activations. Our thinking was that while there were great opportunities in the weight stream, since the weights are known in advance, it is likely that software approaches could deliver much of the potential benefits or should at least be part of the solution. Activations are runtime calculated values and thus less amenable to static analysis. However, as our activation-based methods have matured we recently did explore options that exploit properties of both.[2–4]

Target out-of-the-box networks. For general purpose computing, techniques that required software changes had mixed success. Software development is hard enough as it is, especially for software developed over several years by large development teams. Mirroring that experience, we opted to target accelerators that would work with out-of-the-box NNs. This is not to say that co-designing NNs and hardware is not worth pursuing. To the contrary, co-design should lead to

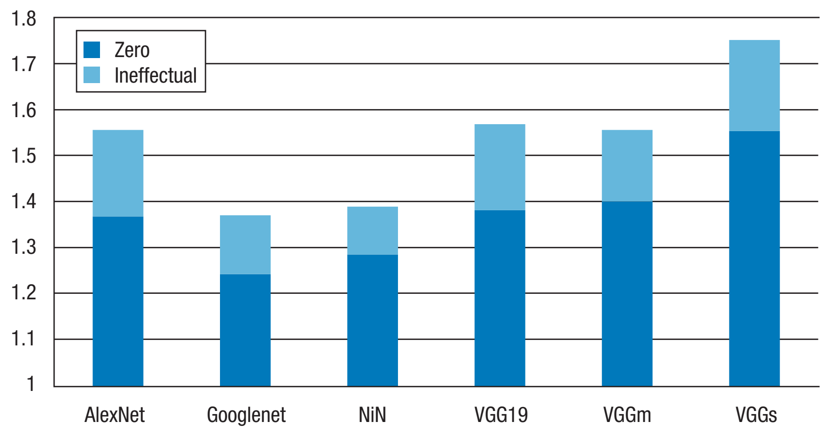


FIGURE 3. Performance improvement when skipping ineffectual activations. Dark blue: skipping activations that are exactly zero; light blue: thresholding while maintaining accuracy.

much greater benefits. However, such efforts take time to mature and yield results and may incur significant overheads to apply even if this is done prior to execution. We opted to design hardware that will deliver immediate benefits while at the same time rewarding related advances in NN design such as reducing value precision.

Risks: breadth vs. depth. Any accelerator design carries risks. What if the application evolves so much that it can no longer execute on the accelerator? For example, accelerators that were specialized for early video formats are by now obsolete as video decoding algorithms have changed dramatically. Or, what if an application uses a mix of other techniques that the accelerator fails to benefit?

An ideal accelerator would be: 1) specialized enough to deliver a desired level of performance, 2) general-enough to support a broader class of applications, and 3) future-proof. Not all these goals are attainable. While breadth is desirable, there is value in in-depth exploration of what is possible for each algorithm of interest in isolation. Such an exploration can ultimately inform a design that is general enough while at the same time benefitting specialized applications and devices with a known expected use-life. Accordingly, we decided to focus on neural networks and since the networks that were readily available were those targeting image classification, most of our work targets this class of NNs. Profiling of these NNs confirmed that convolutional and, to a lesser extent, fully connected layers dominate execution time. Thus we targeted these two layers. Finally, we opted to first target inference, in part as it is a building block for training as well and also since we expect that there will be a lot more devices that will only need to perform inference.

INTERESTING RUNTIME VALUE PROPERTIES

Studying the value stream of image classification NNs revealed several properties which could be potentially exploited for acceleration.

Ineffectual activations

In all CNNs studied, many of the multiplications are ineffectual as they involve a zero valued activation. Even more multiplications could be avoided as long as their activation input value was close enough to zero. What is “close enough” varied per network and layer. We developed an empirical method for finding such thresholds per layer. These ineffectual multiplications represent an opportunity for improving performance. However, exploiting them is a challenge for a massively data-parallel engine. To get any performance, a method was required to promote other useful computations to replace such ineffectual ones. Unfortunately, just checking if an activation is ineffectual takes practically as much time as performing the multiplication, worse, getting another activation requires another data access. Fortunately, the input to every CNN layer but the first is the output of a previous layer. Accordingly, at the output of each layer we can pack the effectual activations tightly in memory so that processing for the next layer proceeds smoothly without having to check for ineffectual computations nor perform additional memory accesses. Our Cnvlutin[5] is such a design, and Figure 3 reports its performance improvements over DaDianNao.

Why do so many zero or near zero activations exist? In the context of image classification and at the first layer, an activation can be thought of as being the probability that a certain visual feature, for example a circle representing an iris, appears at some position. Unless our image is full of such circles all over the place, most such activations would be zero or near zero. While this is an oversimplification, it suggests that ineffectual activations are an intrinsic property of NNs.

Similarly, the Efficient Inference Engine skips zero activations while also taking advantage of weight sparsity[6] using units that perform a single multiply-accumulate. SCNN also targets sparsified NNs, that is NNs where extra steps were taken to convert many weights to zero, skipping both ineffectual weights and activations.[7]

TABLE 1. Activation precision profiles.

Network	Activation precision in bits per layer / effective with dynamic precision detection
AlexNet	9-8-5-5-7 / 5.4-7.4-4.2-4.4-5.8
NiN	8-8-8-9-7-8-8-9-9-8-8-8 / 6.4-7.1-7.8-7.0-5.8-5.2-8.4-7.5-7.6-7.6 4.7-6.8
GoogLeNet	10-8-10-9-8-10-9-8-9-10-7 / 6.2-5.8-6.8-6.3-5.3-6.7-6.3-5.0-5.5-7.9-4.8
VGG-M	7-7-7-8-7 / 5.3-5.1-5.8-3.4-4.8
VGG-S	7-8-9-7-9 / 5.3-5.1-5.0-5.4-4.0
VGG-19	12-12-12-11-12-10-11-11-13-12-13-13-13-13-13-13 / 9.1-7.7-10.0-9.0-11.1-8.8-9.7-8.3-11.6-10.4-12.2-11.7-11.5-11.5-10.4-5.9

Precision variability

We observed early on that the precision NNs need varies per layer, a property that others have observed as well. In the process, we developed a profiling-based method for determining what precisions each layer could use while still maintaining accuracy.[8] As Table 1 shows, the precision needed varies from as little as 5 bits for some layers of AlexNet to up to 13 bits for some layers of VGG-19. These results imply that conventional hardware that uses a one-size-fits-all precision performs many unnecessary and energy wasting computations. But could we build an accelerator that avoids these computations boosting energy efficiency and execution performance? Specifically, we asked whether we could build an accelerator whose execution time scales proportionally with the precision needed. Compared to designs that always use a fixed precision, e.g., 16 bits, for all activations, our desired accelerator would be 16/PL faster when executing layer L, where PL is the activation precision chosen for the layer. Our goal was to squeeze performance even from single bit reductions in precision. For example, for layers using 8 and 7 bits of precision, the accelerator would be 2× and 2.3× faster respectively compared to always using 16-bits of precision. Existing processing engines exploit precision variability at very coarse granularities such as 8- or 16- or 32-bits, and the performance benefits all fall far short of what is possible. Our Stripes accelerator uses bit-serial processing while exploiting data-parallelism to deliver the desired performance scaling.[8] Stripes only boosts performance for convolutional layers. Tartan extends these benefits to fully connected layers albeit at an increased area cost.[3] Stripes and Tartan can be configured accordingly to target any device from high-end server class down to embedded devices. For smaller scale devices, Loom is a variant that exploits precision variability for both weights and activations thus boosting performance even further.[4] By supporting the full spectrum of precisions, the aforementioned accelerator designs reward any advances in the design of reduced precision NNs which may ultimately lead to binary models as proposed by Courbariaux et al.[9]

All aforementioned designs also reduce memory storage and communication requirements as they store only as many bits as necessary to represent the activations in memory. This enables storing and processing larger networks. The Proteus extension brings these benefits to existing bit-parallel engines reducing memory footprint and bandwidth by about 40 percent on average.[10] It uses a lightweight mechanism for converting data from a representation that is convenient for data storage and communication to one that is convenient for data processing.

Dynamic precision detection. While profiling allowed us to determine per layer precisions that maintain TOP-1 (exact match) accuracy at runtime these precisions prove pessimistic. Profiling finds the worst case precision needed for all possible images and across all activations for the layer. In practice, however, the accelerator will be processing: 1) one specific input at any given point of time, and 2) a limited number of input activations per cycle, e.g., 256, and not all activations of the layer. Further reduction in precision is possible when limiting attention to each set of activations that are being processed concurrently. Dynamic Stripes is a surgical, low-cost extension to both Stripes and Loom that detects

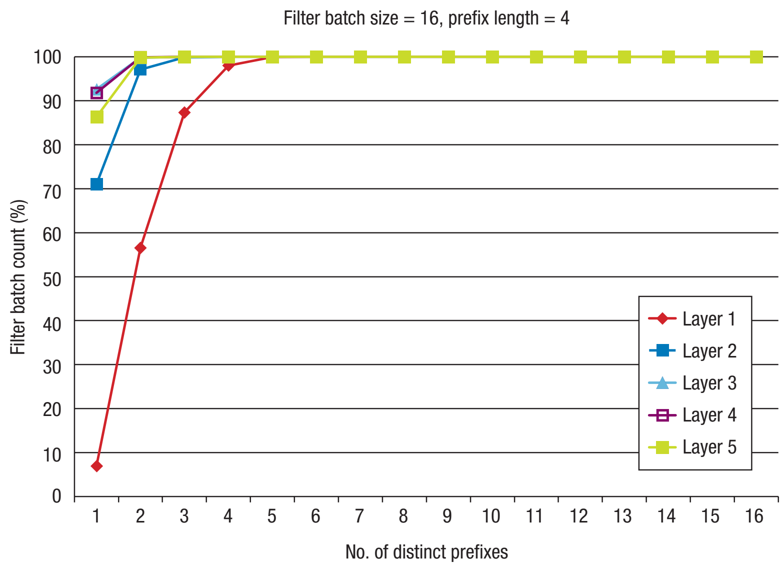


FIGURE 4. Unique 4-bit prefixes for weights appearing at same coordinates across 16 different filters in AlexNet's convolutional layers.

and exploits precision variability at run-time.[11,12] Table 1 shows that the effective activation precisions when these are detected dynamically at a granularity of 256 activations are much shorter than those detected via profiling. Dynamic precision detection coupled with precision detection for weights also drastically reduces off-chip traffic and on-chip storage and communication.[12]

Repeated Calculations

Early on we found out that many of the multiplications happen to process the exact same value pair. Of particular interest were the cases where different filters happen to have exactly the same weight at the same coordinates. At runtime, each of these would be multiplied with the same activation and thus would all be identical. Why would different filters have identical values at the same coordinates? We speculate on at least two reasons: 1) the filter container is a 3D array, whereas the feature that the filter is looking for is not necessary a shape that fits tightly in this container. This will give rise in several weights being zero or near zero, or equivalently not all features will be relevant to all potential object classes. 2) Some features are partially similar which will give rise to some of the weights being similar or the same. The weight redundancy increases to interesting levels once precision is trimmed.

Beyond whole values there is a lot more redundancy when restricting attention to portions of the weights such as their prefixes. Figure 4 demonstrates some of this redundancy in AlexNet. This set of measurements looks at groups of 16 weights each from a different filter. All weights appear at the same coordinates, and the graph shows the distribution of unique 4-bit prefix values. While there are 16 possible combinations for a prefix of 4 bits, in layers 2 to 5 at least 70 percent of the weight groups contain just a single prefix value. Virtually all groups for layers 3 through 5 contain up to 3 distinct prefixes for these layers. Redundancy is lower for layer 1 where just 7 percent of the groups contain a single prefix value. However, about 88 percent of the weight groups contain just 3 distinct prefix values. This redundancy may be useful for compressing the representation of the weights in memory and for reducing the number of computations needed.

Effectual Bit "Density"

Finally, at the individual bit level, activations exhibit a strong bias toward zero. Specifically, as Figure 5 shows, on average, only 8 percent of the activation bits are 1. The figure measures the activation bit values as they are being used in multiplications after we trim activations to the precision needed per layer. In primary school we learned how to do multiplication with a pencil and paper: take one digit from the multiplier and multiply that with the multiplicand. Repeat for the next multiplier digit. Since our numbers are binary, the multiplier bit will be either 0 or 1, and when it is 0 it adds nothing to the final result. Using this method, 92 percent of the time we would be multiplying with a 0 bit when processing CNNs. As Figure 5 shows, If somehow we could develop an accelerator that only processed the effectual bits, that is those that are 1, the potential for performance improvement is 12.5×. Figure 5 further shows that even if somehow we could eliminate all zero-valued activations, nearly 75 percent of activation bits would still be zero resulting in a performance improvement potential of 4×. The behavior persists albeit to a lesser extent even when using 8-bit quantization.[13]

By exploiting precision variability, Stripes and Dynamic Stripes do remove some of these ineffectual calculations. However, at the end there will be some zeroes that will remain. For example, when processing a pair of activations of 8-bits "0100 0000" and "0000 0010," even with dynamic precision detection Stripes will process 6 bits. However, if we were to process only the effectual bits per activation, one step is enough to process both. The Bit-Pragmatic, or simply Pragmatic, accelerator exploits this CNN property.[13]

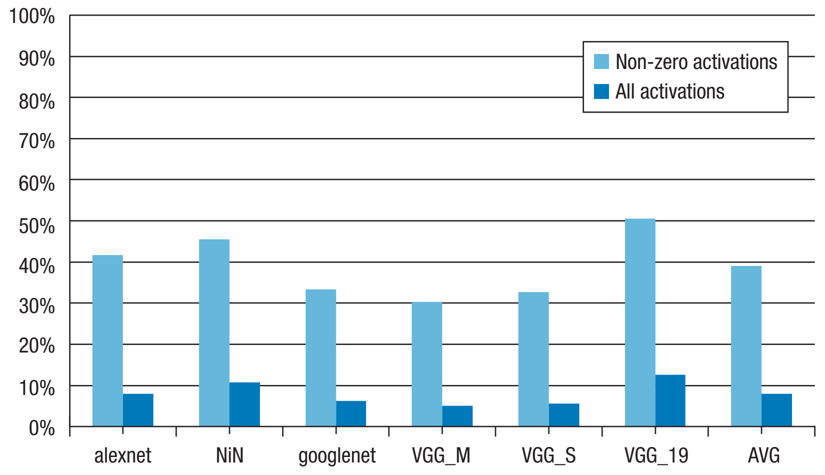


FIGURE 5. Fraction of activation bits that are 1. average over all convolutional layers weighted according to use frequency.

THE BIT-PRAGMATIC ACCELERATOR

Figure 6 shows a simplified example that illustrates the key concept underlying the Pragmatic accelerator. Part (a) shows a structure-based accelerator processing two activations A0 and A1 and two weights W0 and W1 all in using a 16-bit fixed-point representation. Two 16b×16b multipliers produce the 32b products A0×W0 and A1×W1 and an adder reduces those to a single 33b value. An output register accumulates the result. This accelerator will always process two pairs of activations and weights per cycle. To process 16 activation and weight pairs it will need 16 cycles. In our example, each of the activation values contain just a single power of two, 2^3 for A0 and 2^{13} for A1. As a result, the bit-parallel accelerator will process 15 + 15 zero activation bits all contributing nothing to the final output.

Part (b) shows a simplified Pragmatic accelerator that processes only the effectual activation bits. The activations now are no longer represented in a positional representation, but instead as lists of powers of two, since each has just one constituent power of two, and the lists are (0011) and (1011) for A0 and A1 respectively. If A0 were "0000 1100," it would be represented as (0100, 0011). Each cycle, this unit "multiplies" one power of two per activation with the corresponding weight. The multipliers have been replaced with shifters, since multiplying by a power of two amounts to simple shifting. The rest of the unit remains unchanged as every cycle of two products of 32b each are reduced and accumulated. The unit processes the two activation and weight products in a single cycle and thus is as fast as the bit-parallel unit of part (a). However, if A0 or A1 contained more than one ineffectual bits, then this unit would require a proportional number of cycles to calculate the products. So, its execution time scales proportionally with the number of effectual bits which is in part what we wanted. Unfortunately, this design is at best as fast as the bit-parallel design and only when all activations contain just one effectual bit. In the worst case, when at least one of the activations has 16 effectual bits, it will be 16× slower.

Fortunately, convolutional layers exhibit parallelism and weight reuse across windows, two properties that Pragmatic exploits to ensure that it is always at least as fast as a bit-parallel engine without requiring to read more weight or activation bits from memory. The latter would require wider memories, an expensive addition. Part (c) shows Pragmatic's approach. The unit of part (b) has been replicated 16 times. Each of the 16 units processes a different activation pair. However, all units share the same weights. This is possible by processing 16 windows in parallel, one per unit. Whereas the bit-parallel unit processed 2×16b activations, for a total of 32b of activation inputs per cycle, the Pragmatic unit processes 32 activations, one power of two per activation. This is equivalent to 32 bits of activations per cycle. While Pragmatic uses 4 bits per activation to powers of two, this conversion is done after the activations are read from storage.

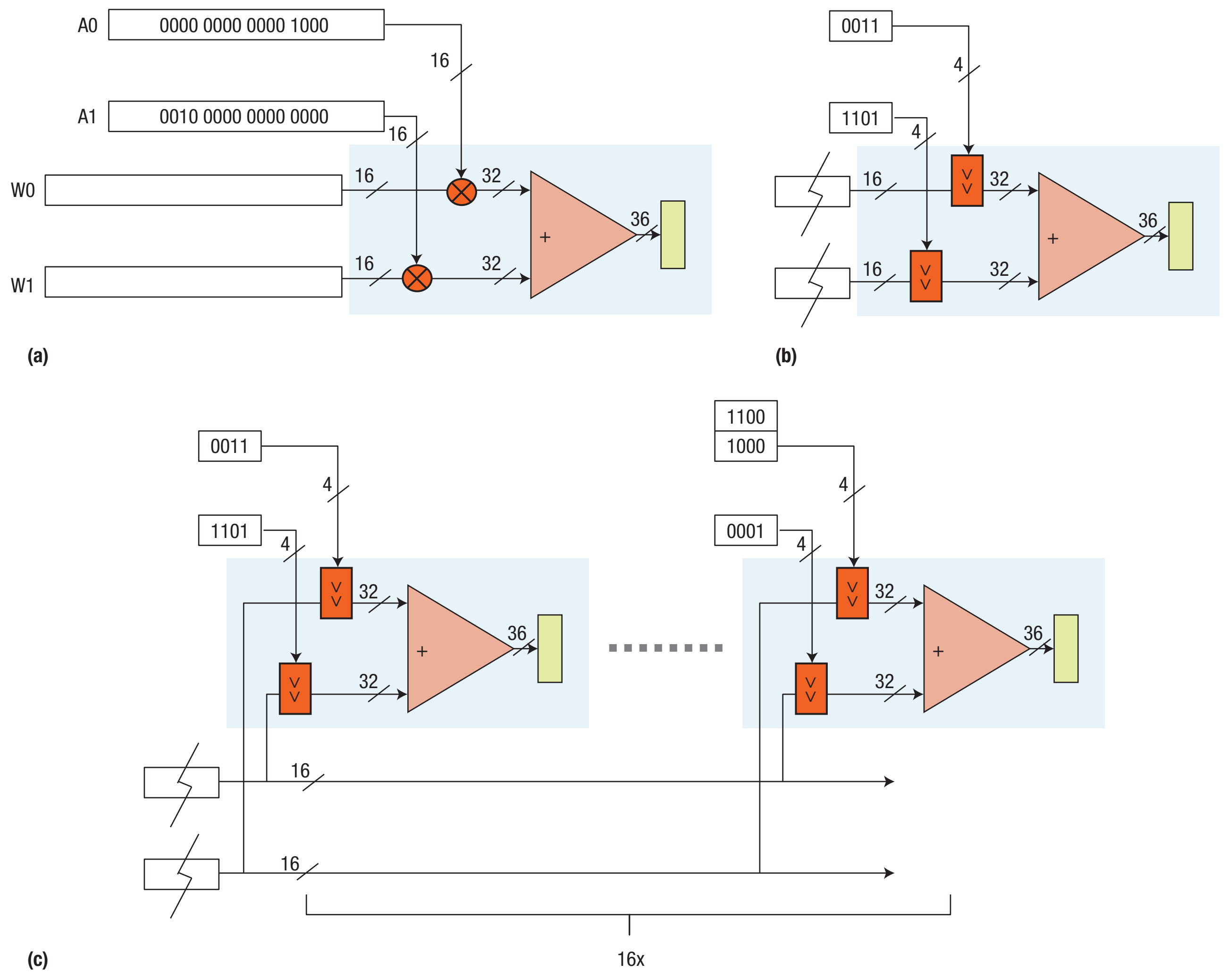


FIGURE 6. Pragmatic's approach: an example. (a) Structure-based accelerator. (b) Processing the powers of two serially. (c) Exceeding performance of the structure-based accelerator.

In the worst case, when all 16 bits of at least one activation are 1, this unit would require 16 cycles to process all 32 activations producing 32 activation and weight products. This matches the processing capability of the bit-parallel engine of part (a). The two engines proceed through the computation in different order; however, at the end, they produce the same results. When all activations have at most one effectual bit, the Pragmatic unit will take just 1 cycle to do the work that the conventional unit would do in 16 cycles, and thus be 16× faster. In general, if the maximum number of effectual bits per activation is N, then Pragmatic will be 16/N× faster.

Making It Practical

Unfortunately, the straightforward implementation of Pragmatic as described proved impractical. The units were about 4× larger than their bit-parallel equivalent, and the performance improvements were not compelling enough. We had to develop several techniques that combined allowed a practical implementation of Pragmatic. Our discussion highlights three of them. The first is two-stage shifting. In the straightforward design, for every output activation we are processing 16 weight and activation offsets pairs simultaneously. Since we shift each weight by a 4 bit power of two, in the worst case, one of the powers will be 0 and another 15. Each of those shifters needs thus to accept a 16b weight and to produce a 32b output "product". Consequently, the adder tree needs to accept 32b products as inputs. While this design offers us maximum flexibility to eliminate ineffectual activations bits it does so at a high cost. Two-stage shifting gives up some of this flexibility and thus some of the performance improvement potential to drastically reduce costs. The idea is to process the input activations into subgroups. For example, instead of allowing any power of two to be processed concurrently with any other power of two, we can process each activation in groups of four bits at a time. In this case processing two activations with values "0100 0000 0000 0000" and "0000 0000 0000 0010" will be done in two cycles even though each contain just a single effectual bit. In the first cycle we will process the group of the four least significant bits, 0000 and 0010, and in the second the group of the four most significant bits, 0100 and 0000. In practice we found that processing bits in groups of four was sufficient to achieve most of the performance possible with unrestricted processing. Pragmatic chooses the beginning of each group dynamically at run time. For example, it would process "0000 0000 0001 0000" and "0000 0000 0000 1000" in a single cycle.

The second technique was to allow partial decoupling of the activation lanes. In the straightforward design Pragmatic processes all activations in the group before proceeding to the next group. By adding buffers at the weight inputs and by statically placing activations into subgroups, it is possible to allow some subgroups to run ahead of others. In practice using just one weight buffer and thus allowing subgroups to run just one activation set ahead boosted performance considerably. These buffers are anyhow necessary to support full utilization when executing fully connected layers.

Finally, so far we assumed that activations are represented as a sum of powers of two. However, the underlying design can easily handle both adding and subtracting powers. This is a form of Booth encoding, a technique usually reserved for reducing the latency of high performance multipliers. For example, activation "0011 1100 0000 0000" can be represented as "(0010 0000 0000 0000 - 0000 0010 0000 0000)," or as "($2^{13} - 2^{9}$)." Pragmatic uses a modified form of Booth encoding to avoid increasing the number of cycles in conjunction with 2-stage shifting.

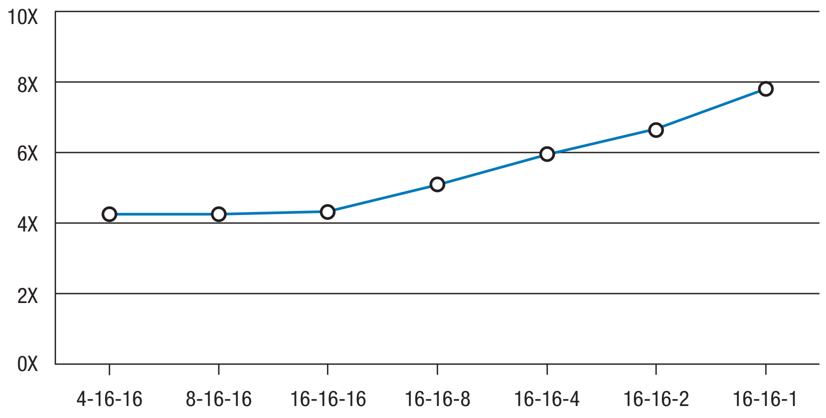


FIGURE 7. Performance improvement with various pragmatic configurations.

Execution Time Reduction

Figure 7 shows how performance (the inverse of execution time) improves compared to an equivalent configured DaDianNao-like accelerator for various configurations. Three parameters define a configuration: the terms per filter, the filters per tile, and the number of tiles. The terms per filter is the number of activation and weight products calculated per filter. The filters per tile is the number of filters processed per processing engine tile. The x-axis shows the configurations in a tiles-filters/tile-terms/filter format. A 16-8-4 configuration has 16 tiles, each processing 8 filters and each processing 4 products, in total it processes 512 terms per cycle. For a design configured to match DaDianNao's original 16-16-16 server-class configuration, Pragmatic boosts performance by 4.3× on average. When processing fewer terms per filter, Pragmatic experiences less imbalance across activations, and performance increases and reaches nearly 8× for a configuration with one term per filter, which may be more appropriate for an embedded design.

TABLE 2. Value-based accelerator characteristics relative to dadiannao.[1]

Accelerator	Configuration	Performance	Power	Area	Frecuency	Tech. node
DaDianNao	16-16-16	3.9 Tmul/sec	17.6 Watt	78mm2	980 Mhz	65nm

Accelerator	Compared to DaDian- Nao Conf.	Relative performance	Relative energy efficiency	Relative area	Value property
Cnvlutin[5]	16-16-16	1.6×	1.47×	1.05×	Ineffectual activation values
Dynamic stripes[11]	16-16-16	2.6×	1.54×	1.35×	Dynamic activation precision
Loom[4]	1-8-16	3.6×	2.9×	0.94×	Dynamic activation + weight precisions
Pragmatic[13]	16-16-16	4.3×	1.71×	1.68×	Ineffectual activation bits
Tactical[2]	4-16-16	10.2×	2.4×	1.14×	Zero weights + ineffectual activation bits
Laconic	1-8-16	16×	1.63×	2.39×	Ineffectual activation bits of weights + activations

Reported is the performance, energy efficiency and area compared to an equivalent DaDianNao configuration shown under column "DaDianNao Conf." DaDianNao configurations are labeled as "tiles - filters/tile - products/filter."

SUMMARY

Table 2 summarizes some of our designs and reports their relative performance, energy efficiency, and area normalized to an equivalent DaDianNao configuration. The table also reports on a more recent accelerator, Tactical[2] that combines the benefits of Pragmatic or Dynamic Stripes with a lightweight zero weight skipping front-end resulting in multiplicative benefits. The Laconic configuration shown uses half of the weight memory wires. With an equal number of weight memory wires, the speedup increases to 30×. The results reported for Tactical are for pruned versions of AlexNet, Googlenet, and ResNet-50. Further, the results for Loom are with dynamic precision detection.

Early successes in hardware acceleration for Deep Learning relied on exploiting its computation structure and data reuse, e.g. Y. Chen and Chen.[1,14] As our work and that of others exemplify, many recent DL hardware accelerators exploit the various forms of informational inefficiency that deep learning neural networks (DNNs) exhibit. It has been found that informational inefficiency manifests in DNNs as ineffectual neurons,[6,15] activations,[6,5,15] or weights,[16,15] as an excess of precision, e.g. Warden and Judd,[17,8] as ineffectual activation bits,[13] or in general as over-provisioning. Whether these inefficiencies are best exploited statically, dynamically, or both is an open question. Furthermore, which forms of inefficiency will persist as DNNs evolve remains to be seen. These past successes demonstrate that at this stage of our exploration on how to best deliver the hardware performance advances needed to support DL innovation, identifying DNN properties that hardware and/or software could potentially exploit is invaluable. Moreover, the progression of advances, starting from simply looking at precision for reducing storage and arriving to effectual bit density to improve performance, demonstrate that it is not easy to foresee upfront what innovations lie ahead. Accordingly, we ought to encourage further exploration even for directions that may seem unlikely to deliver benefits or that today seem too farfetched.

Along these lines, our accelerators capitalize on some of the value properties of CNNs while working with out-of-the-box networks thus making deployment possible today. More importantly, they open up new opportunities and create new incentives for CNN designers providing a safe path towards innovation while offering rewards for even small advances. Specifically, if deployed, they have the potential to accelerate innovation in: 1) extremely low precision NN design with an eye towards ternary or ternary networks (Stripes, Loom, Pragmatic, and Tactical), and 2) weight pruning (Tactical). They enable experimentation with the whole spectrum of precision choices while also delivering excellent performance for full-precision networks. They have the potential to "incentivise" the machine learning community to further invest in these directions delivering immediate, proportional rewards. Eventually, if extremely low precision and heavily pruned networks take over, more efficient hardware platforms can safely take over. New opportunities could also arise such as further reducing the number of bits that are 1, adopting other quantization schemes such as using only a single power of two on a case-by-case basis, or even rearranging filters to reduce effectual bit imbalance. All without requiring that any newly developed scheme work for all networks.

We encourage the interest readers to visit the first author's web page for our most recent findings. ◼

REFERENCES

1. 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, Dec 2014, pp. 609–622.
2. A. D. Lascorz, P. Judd, D. M. Stuart, Z. Poulos, M. Mahmoud, S. Sharify, M. Nikolic, and A. Moshovos, "Bit-Tactical: Exploiting Ineffectual Computations in Convolutional Neural Networks: Which, Why, and How," *CoRR*, vol. abs/1803.03688, 2018. [Online]. Available: https://arxiv.org/abs/1803.03688
3. A. Delmás, S. Sharify, P. Judd, and A. Moshovos, "Tartan: Accelerating fully-connected and convolutional layers in deep learning networks by exploiting numerical precision variability," *CoRR*, vol. abs/1707.09068, 2017. [Online]. Available: https://arxiv.org/abs/1707.09068
4. S. Sharify, A. D. Lascorz, P. Judd, and A. Moshovos, "Loom: Exploiting weight and activation precisions to accelerate convolutional neural networks," *CoRR*, vol. abs/1706.07853, 2017. [Online]. Available: https://arxiv.org/abs/1706.07853
5. J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. Enright Jerger, and A. Moshovos, "Cnvlutin: Ineffectual-neuron-free deep neural network computing," in 2016 IEEE/*ACM International Conference on Computer Architecture* (ISCA), 2016.
6. S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "EIE: Efficient Inference Engine on Compressed Deep Neural Network," arXiv:1602.01528 [cs], Feb. 2016, arXiv: 1602.01528. [Online]. Available: https://arxiv.org/abs/1602.01528
7. A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, "Scnn: An accelerator for compressed-sparse convolutional neural networks," in Proceedings of the 44th *Annual International Symposium on Computer Architecture*, ser. ISCA '17. New York, NY, USA: ACM, 2017, pp. 27–40. [Online]. Available: https://doi.acm.org/10.1145/3079856.3080254
8. P. Judd, J. Albericio, T. Hetherington, T. Aamodt, and A. Moshovos, "Stripes: Bit-serial Deep Neural Network Computing," in *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-49, 2016.
9. M. Courbariaux, Y. Bengio, and J.-P. David, "BinaryConnect: Training Deep Neural Networks with binary weights during propagations," *CoRR*, vol. abs/1511.00363, Nov. 2015. [Online]. Available: https://arxiv.org/ abs/1511.00363
10. P. Judd, J. Albericio, T. Hetherington, T. M. Aamodt, N. Enright Jerger, and A. Moshovos, "Proteus: Exploiting numerical precision variability in deep neural networks," in *Proceedings of the 2016 International Conference on Supercomputing*, ser. ICS '16. New York, NY, USA: ACM, 2016, pp. 23:1–23:12. [Online]. Available: http://doi.acm.org/10.1145/2925426.2926294
11. A. Delmás, P. Judd, S. Sharify, and A. Moshovos, "Dynamic stripes: Exploiting the dynamic precision requirements of activation values in neural networks," *CoRR*, vol. abs/1706.00504, 2017. [Online]. Available: http://arxiv.org/abs/1706.00504
12. A. Delmás, S. Sharify, P. Judd, M. Nikolic, A. Moshovos, "DPRed: Making Typical Activation Values Matter In Deep Learning Computing", *CoRR*, vol. abs/1804.06732, 2018. [Online]. Available: https://arxiv.org/abs/1804.06732
13. J. Albericio, A. Delma´s, P. Judd, S. Sharify, G. O'Leary, R. Genov, and A. Moshovos, "Bit-pragmatic deep neural network computing," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-50 '17. New York, NY, USA: ACM, 2017, pp. 382–394. [Online]. Available: http://doi.acm.org/10.1145/3123939.3123982
14. Chen, Yu-Hsin and Krishna, Tushar and Emer, Joel and Sze, Vivienne, "Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep

ABOUT THE AUTHORS

ANDREAS MOSHOVOS is a professor in the Electrical and Computer Engineering Department at the University of Toronto. His research interests are in architecting highly efficient and high-performance computing hardware. Moshovos has a PhD in computer science from the University of Wisconsin-Madison. He is a Senior Member of IEEE and a Fellow of ACM. Contact him at moshovos@eecg.toronto.edu.

JORGE ALBERICIO is a senior deep learning architect at NVIDIA. He has a PhD in systems engineering and computing from the University of Zaragoza. He was a postdoctoral fellow at the University of Toronto from 2013 to 2016, where he worked on branch prediction, approximate computing, and hardware accelerators for machine learning. He is a member of IEEE. Contact him at jorge.albericio@gmail.com.

PATRICK JUDD is a fourth-year PhD candidate at the University of Toronto and has joined NVIDIA as a senior deep learning architect. His research interests include computer architecture, machine learning, and approximate computing. His research focuses on the design of hardware accelerators for deep neural networks that exploit approximation for improved performance and energy efficiency. Judd is a student member of IEEE. Contact him at patrick.judd@mail.utoronto.ca.

ALBERTO DELMÁS LASCORZ is a third-year PhD candidate at the University of Toronto, where he focuses on hardware design for machine-learning accelerators. His research interests include computer architecture, deep learning, and embedded and reconfigurable systems. Delmás Lascorz previously studied computer engineering at the University of Zaragoza. He is a student member of IEEE. Contact him at a.delmaslascorz@mail.utoronto.ca.

SAYEH SHARIFY is a third year PhD candidate at the University of Toronto. Her research interests include computer architecture, machine learning, embedded systems, and reconfigurable computing. She designs hardware accelerators for machine-learning algorithms. Sharify previously studied computer engineering at Sharif University of Technology. She is a student member of IEEE and ACM. Contact her at sayeh@ece.utoronto.ca.

ZISSIS POULOS is a PhD candidate in the Electrical and Computer Engineering Department at the University of Toronto. His research interests are in the design of high-performance hardware for machine learning applications, as well as in developing approximation methods for network diffusion and social graph reasoning. He holds a MASc in electrical engineering from the University of Toronto and is a student member of IEEE. Contact him at zpoulos@eecg.toronto.edu.

TAYLER HETHERINGTON is a final-year PhD candidate in computer engineering at the University of British Columbia and is currently working at Oracle Labs. His research interests include computer architecture, specifically general-purpose GPUs, machine-learning accelerators, and system software. He is a student member of the IEEE. Contact him at taylerh@ece.ubc.ca.

TOR AAMODT is a professor in the Department of Electrical and Computer Engineering at the University of British Columbia. His research interests include architecture of general-purpose GPUs and machine-learning accelerators. Aamodt has a PhD in electrical and computer engineering from the University of Toronto. He is a member of IEEE and ACM. Contact him at aamodt@ece.ubc.ca.

NATALIE ENRIGHT JERGER is the Percy Edward Hart Professor of Electrical and Computer Engineering at the University of Toronto. Her research interests include computer architecture, approximate computing, interconnection networks, and hardware acceleration of machine learning. Enright Jerger has a PhD in electrical engineering from the University of Wisconsin-Madison. She is a senior member of IEEE and ACM. Contact her at enright@ece.utoronto.ca.

Convolutional Neural Networks," in IEEE *International Solid-State Circuits Conference*, ISSCC 2016, Digest of Technical Papers, 2016, pp. 262–263.

15. A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, "Scnn: An accelerator for compressed-sparse convolutional neural networks," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA '17. New York, NY, USA: *ACM*, 2017, pp. 27–40. [Online]. Available: http://doi.acm.org/10.1145/3079856.3080254
16. S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen, "Cambricon-x: An accelerator for sparse neural networks," in 49th Annual IEEE/ACM *International Symposium on Microarchitecture*, MICRO 2016, Taipei, Taiwan, October 15-19, 2016, 2016, pp. 1–12. [Online]. Available: https://doi.org/10.1109/MICRO.2016.7783723
17. P. Warden, "Low-precision matrix multiplication," https://petewarden.com, 2016.