Exploiting Typical Values to Accelerate Deep Learning

Andreas Moshovos, *Member, IEEE,* Jorge Albericio, *Member, IEEE,* Patrick Judd, *Member, IEEE,* Alberto Delmas Lascorz, *Member, IEEE,* Sayeh Sharify, *Member, IEEE,* Zissis Poulos *Member, IEEE,* Tayler Hetherington, *Member, IEEE,* Tor Aamodt, *Member, IEEE,* Natalie Enright Jerger, *Senior Member, IEEE,*

***Abstract*—Many of the performance enhancing techniques that empower modern general-purpose processors rely on typical program behavior which is far from random. We build on this approach to develop specialized hardware that accelerates the execution of Deep Learning Networks. Specifically, we identifying properties in the value stream of Deep Learning Networks with the goal of exploiting them at the hardware level to reduce the execution time of Deep Learning Networks. Higher performance has a multitude of potential applications: it can enable the processing of larger and potentially more accurate networks, it can allow smaller scale devices to tackle problems that are today beyond reach, or it can enable the real-time processing of sensor inputs. This article affirms the need for hardware acceleration for Deep Learning, articulates our strategy, overviews several interesting value properties of Deep Learning Networks and the progression of hardware accelerators we have been developing. Finally, it emphasizes our *Pragmatic* accelerator which exploits the lopsided bit value distribution of the runtime calculated values of image classification Deep Learning networks.**

***Index Terms*—Deep Learning Hardware, Inference, Convolutional Neural Networks, Value-Based Acceleration**

A. Moshovos, P. Judd, A. Delmas Lascorz, S. Sharify, Z. Poulos, and N. Enright Jerger are with the Edward S. Rogers Sr. Department of Electrical and Computer Engineering of the University of Toronto, Toronto, ON, Canada.

T. Hetherington and T. Aamodt are with the Electrical and Computer Engineering Department of the University of British Columbia, Vancouver, BC, Canada.

Jorge Albericio is with NVIDIA Corporation. This work was completed while J. Albericio was a Postdoctoral Fellow at the Edward S. Rogers Sr. Department of Electrical and Computer Engineering of the University of Toronto.

Manuscript received Oct 12, 2017

I. Why Deep Learning Hardware?

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, 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 Deep Learning community has been able to collect or exploit vast amounts of data 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 data and to build more sophisticated building blocks and arrangements. More processing power is thus an enabler for 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 to boost performance and energy efficiency above and beyond what is possible by exploiting the computational structure of these applications. In this article, we focus on the acceleration of inference with convolutional neural networks.

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

II. The Need for Acceleration

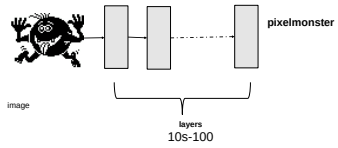
For the past three decades or so, computing hardware performance was roughly doubling every two years. A task that would take about one hour to perform on a 1985 desktop computer would complete in less than a minute on computer built in 1995. This exponential performance growth was fueled by Moore's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

Unfortunately, using more and faster transistors requires more power. In the early 2000s, processor power consumption and density surpassed practical limits and 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. At the extreme, graphics processors target a certain class of such workloads that could be decomposed into 1000s of threads each executing the same code and mostly in lock-step. Computer graphics is such a *data-parallel* workload. As the underlying semiconductor technology scaling trend persists and as current architectural techniques are reaching their limits, further innovation is now required to boost performance.

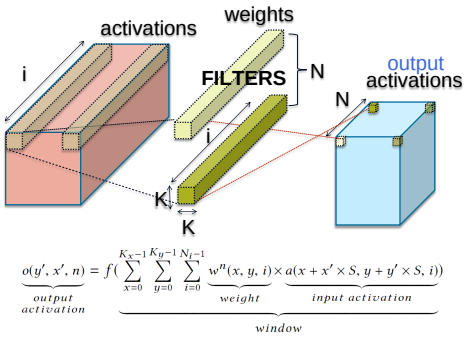
Since power is now the main constraint, further performance improvements for either general-purpose or graphics processors require reducing the energy expended per operation: if each operation requires less energy, we will be able to use the abundant hardware resources to perform more operations concurrently 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 Deep Learning to understand how specialization can improve energy per operation and thus performance.

III. Hardware Acceleration and Deep Learning

Deep Learning utilizes neural networks (NN). Figure 1a shows an example feed-forward NN where several layers operate in sequence. 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. In other NNs there is feedback among layers. 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 can be thought of as a special case of convolutional layers.



(a)



(b)

Fig. 1. a) A Feed-Forward Image Classification Neural Network. b) A Convolutional Layer.

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, e.g., an image) and produces an output activation 3D array. The CVL convolves the input activations with several filters, each a 3D array of pre-determined 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 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 activation and weight pairs. A constant *bias* is usually added at the final result prior to passing it through a non-linear activation function producing 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. To fully process an input activation array, the filters scan the input using a stride S. The input activation subarray used each time is called a *window*. In the discussion that follows we ignore the addition of the bias as it easy to implement along with the activation function.

A. The Opportunity for Deep Learning Acceleration

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

```
outa = 0
for xi = 0 to S
  for yi = 0 to S
    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 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 reduce these overheads.

B. 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 accumulating 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

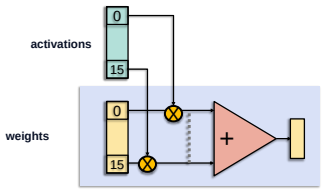


Fig. 2. A Computation-Structure-Based Accelerator

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's 256 units are 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.

C. Value-Based Acceleration

We purposely targeted techniques that could complement structure-based acceleration. 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:

1) 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 memory can supply the data. Unfortunately, it is not possible to build large and fast 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 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 are 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. Activations are runtime calculated values and thus less amenable to static analysis. We recently did explore options that exploit properties of both [2].

2) Co-designing the network and hardware: 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 much greater benefits. However, such efforts take time to mature and yield results. 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.

3) 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 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. 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 will ultimately inform a general-enough design and would immediately benefit specialized applications. Accordingly, we decided to focus on neural networks and since the networks that were readily available were those targeting image classification in most of our work we targeted 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.

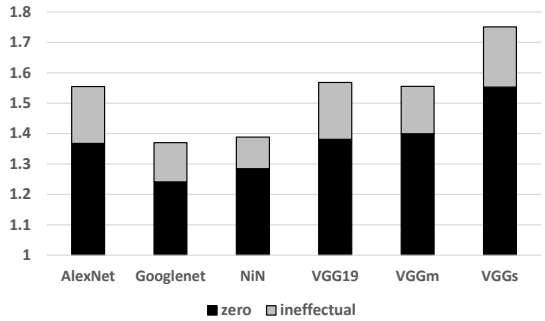


Fig. 3. Performance improvement when skipping ineffectual activations. Bottom part: skipping activations that are exactly zero, top bar: thresholding while maintaining accuracy.

IV. Interesting Runtime Value Properties

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

A. 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 per 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 way was needed to replace such ineffectual computations with useful 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 accesses. An instance of such a design is Cnvlutin [3] and Figure 3 reports the performance improvements possible over DaDianNao.

Why 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 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.

The Efficient Inference Engine also skips zero activations while also taking advantage of weight sparsity [4] using units that perform a single multiply-accumulate. SCNN targets sparsified NNs skipping both ineffectual weights and activations [5].

B. 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 [6]. As Table I 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 $\frac{16}{P_a^L}$ faster when executing layer L, where P_a^L 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 $2\times$ and $2.3\times$ 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.

Our Stripes accelerator uses bit-serial processing while exploiting data-parallelism to deliver the desired performance scaling [6]. Stripes only boosts performance for convolutional layers. Tartan extends these benefits to fully-connected layers albeit at an increased area cost [7]. 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 [8]. By supporting the full spectrum of precisions, the aforementioned accelerator designs reward any advances in the design of reduced precision NNs, e.g., [9].

Network	Per Layer Activation Precision in Bits
AlexNet	9-8-5-5-7
NiN	8-8-8-9-7-8-8-9-9-8-8-8
GoogLeNet	10-8-10-9-8-10-9-8-9-10-7
VGG_M	7-7-7-8-7
VGG_S	7-8-9-7-9
VGG_19	12-12-12-11-12-10-11-11-13-12-13-13-13-13-13-13

TABLE I
PER LAYER ACTIVATION PRECISION PROFILES.

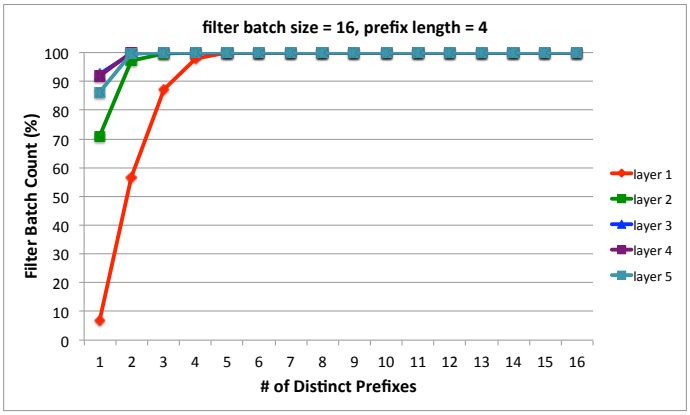


Fig. 4. Unique 4-bit prefixes for weights appearing at same coordinates across 16 different filters.

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. Proteus reduces memory footprint and bandwidth by about 40% on average [10]. It uses a light-weight mechanism for converting data from a representation that is convenient for data storage and communication to one that is convenient for data processing.

1) Dynamic Precision Detection: While profiling allowed us to determine per layer precisions that maintain accuracy at runtime these precisions would be 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. 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 and exploits precision variability at run-time [11].

C. 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. 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 the 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% of the weights 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% of the groups contain a single prefix value. However, about 88% 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 computations needed.

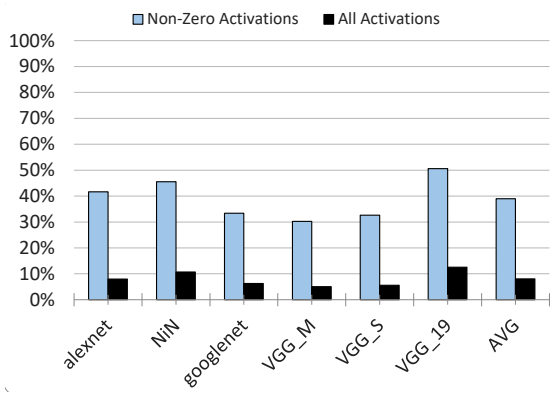


Fig. 5. Fraction of Activation Bits that are 1. Average over all convolutional layers weighted according to use frequency.

D. Bit "Density"

Finally, at the individual bit level activations exhibit a strong bias towards zero. Specifically, as Figure 5 shows, on average, only 8% 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 multiplicant. Repeat for the next multiplier digit. Since our numbers are binary, the bit will be either 0 or 1, and when it is 0 it adds nothing to the final product. Using this method, 92% of the time we would be multiplying with a 0 bit when processing CNNs. 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 $12.5\times$. Figure 5 shows that even if somehow we could eliminate all zero-valued activations, nearly 75% of activation bits are still zero. The performance improvement potential is $4\times$. The behavior persists albeit to a lesser extent even when using 8-bit quantization [12].

By exploiting precision variability Stripes and Dynamic Stripes do remove some of this 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 [12].

V. The Bit-Pragmatic Accelerator

Figure 6 shows a simplified example that illustrates the key concept underlying the Pragmatic accelerator. Part (a) shows how a structure-based accelerator would process two activations A0 and A1 and two weights W0 and W1 all in a 16-bit fixed-point representation. Two 16b$\times$16b multipliers produce the 32b products A0$\times$W0 and A1$\times$W1 and an adder reduces those to a single 33b value. The result is accumulated into an output register. 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, the two activations contain each just a single power of two, 2^3 for A0 and 2^{13} for A1. 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, A0's list contains `(0011)` and A1's list `(1011)`. If A0 was `0000 1100`, it would be represented as `(0100, 0011)`. Each cycle, this unit "multiplies" one power 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 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). If A0 or A1 contained more than one ineffectual bits, then this unit would require a proportional number of cycles to calculate the products 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 $16\times$ 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$\times$16b activations, for a total of 32b of activation inputs, 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 represent the

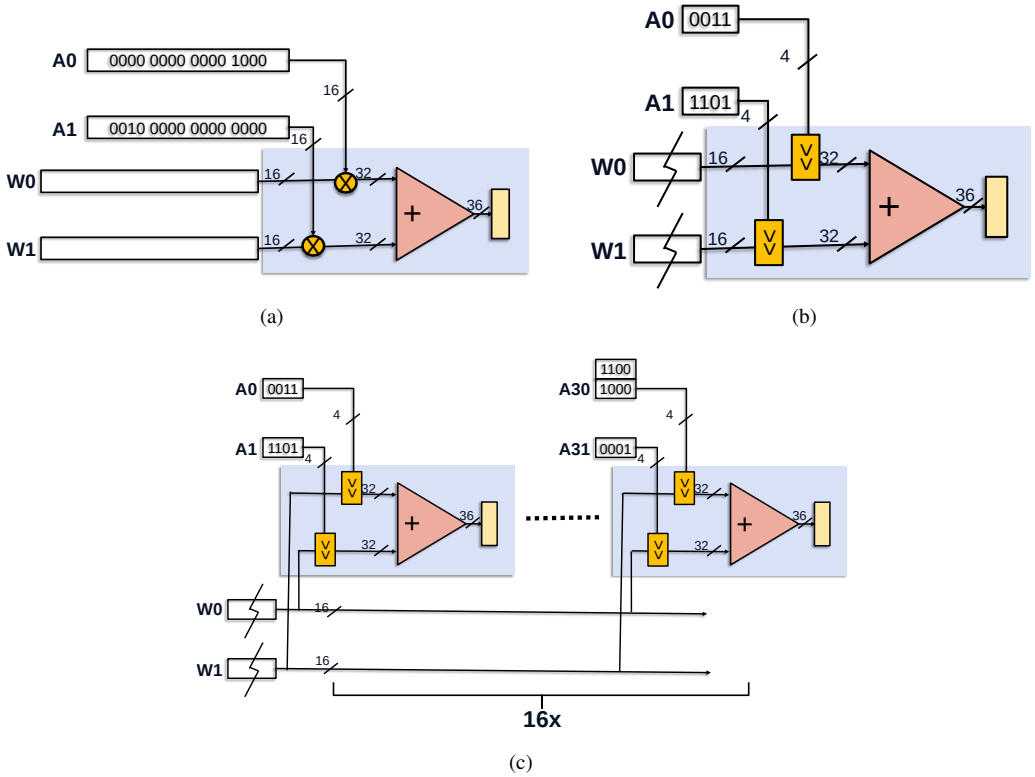


Fig. 6. Pragmatics approach: An example. (a) Structure-based accelerator, (b) Processing the powers of two serially, and (c) exceeding performance of the structure-based accelerator.

power of two being processed, this conversion is done after the activations are read from storage. 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 $\frac{16}{N}$ faster.

A. *Making it Practical*

Unfortunately, the straightforward implementation of Pragmatic as described proved impractical. The units were about 4× larger than their bit-parallel equivalents and while performance improvements were higher than the overall area overhead the trade off did not seem as compelling.

There were several techniques that combined allowed a practical implementation of Pragmatic. Our discussion highlights three of them. The first is two-stage shifting. In the straight forward 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

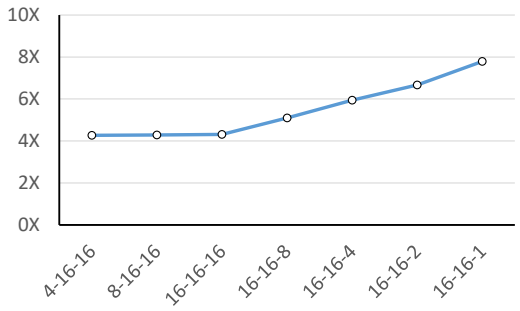


Fig. 7. Performance Improvement with Various Pragmatic Configurations.

TABLE II
VALUE-BASED ACCELERATOR CHARACTERISTICS RELATIVE TO DADIANNAO [1]. REPORTED IS THE PERFORMANCE, ENERGY EFFICIENCY AND AREA COMPARED TO AN EQUIVALENT DADIANNAO CONFIGURATION SHOWN UNDER COLUMN "DADIANNAOCONF.". DADIANNAOCONFIGURATIONS ARE LABELLED AS "TILES - FILTERS/TILE - PRODUCTS/FILTER".

Accelerator	Configuration	Performance	Power	Area	Frequency	Tech. Node
DaDianNao (DaDianNao) [1]	16-16-16	3.9 Tmul/sec	$17.6\ Watt$	$78mm^2$	980 Mhz	65nm

Accelerator	Compared to DaDian-NaoConf.	Relative Perfor-mance	Relative Energy Efficiency	Relative Area	Layers	Value Property
Cnvlutin [3]	16-16-16	$1.6\times$	$1.47\times$	$1.05\times$	CVL	Ineffectual Activation Values
Dynamic Stripes [11]	16-16-16	$2.6\times$	$1.54\times$	$1.35\times$	CVL	Dynamic Activation Precision
Pragmatic [12]	16-16-16	$4.3\times$	$1.71\times$	$1.68\times$	CVL	Ineffectual Activation Bits
Tactical [2]	4-16-16	$10.2\times$	$2.4\times$	$1.14\times$	CVL	Zero Weights and Ineffectual Activation Bits

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.

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). Pragmatic uses a modified form of Booth encoding to avoid increasing the number of cycles in conjunction with 2-stage shifting.

B. 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 DaDianNaos original 16-16-16 server-class configuration, Pragmatic boosts performance by $4.3\times$ on average. When processing less terms per filter Pragmatic experiences less imbalance across activations performance increases and reaches nearly $8\times$ for a configuration with one term per filter which may be more appropriate for an embedded design.

VI. SUMMARY

Table II summarizes some of our designs and reports their relative performance, energy efficiency and area normalized to an equivalent DaDianNao configuration. The table also reports our most recent accelerator, *Tactical* [2] that combines the benefits of Pragmatic or Stripes with a lightweight zero weight skipping front-end resulting in multiplicative benefits. The results reported for Tactical are for pruned versions of AlexNet, Googlenet and ResNet-50.

VII. Conclusion

Early successes in hardware acceleration for Deep Learning relied on exploiting its computation structure, e.g., [1], [13]. 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 [4], [14], activations [4], [3], [14], or weights [15], [14], as an excess of precision, e.g., [16], [6], as ineffectual activation bits [12], 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.

Our accelerators capitalize on some of these properties while working with out-of-the-box networks thus making deployment possible today. More importantly, they open up new opportunities and incentives for CNN designers providing a safe path towards innovation while offering rewards for even small advances. Specifically, if deployed, will accelerate innovation in: 1) extremely low precision NN design with an eye towards binary and 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, once and if extremely low precision and heavily pruned networks take over, more efficient hardware platforms can be deployed. New opportunities also arise such as reducing the number of bits that are 1 further, 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.

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: Enabling innovation towards chaff-free deep learning computing," *CoRR*, vol. abs/1803, 2018. [Online]. Available: https://arxiv.org/abs/1803.03688

[3] 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.

[4] 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: http://arxiv.org/abs/1602.01528

[5] 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

[6] 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.

[7] A. Delmas, 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: http://arxiv.org/abs/1707.09068

[8] 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: http://arxiv.org/abs/1706.07853

[9] M. Courbariaux, Y. Bengio, and J.-P. David, "BinaryConnect: Training Deep Neural Networks with binary weights during propagations," *ArXiv e-prints*, Nov. 2015.

[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. Delmas, 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] J. Albericio, A. Delmás, 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

[13] Chen, Yu-Hsin and Krishna, Tushar and Emer, Joel and Sze, Vivienne, "Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks," in *IEEE International Solid-State Circuits Conference, ISSCC 2016, Digest of Technical Papers*, 2016, pp. 262–263.

[14] 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

[15] 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

[16] P. Warden, "Low-precision matrix multiplication," https://petewarden.com, 2016.