Loom: Exploiting Weight and Activation Precisions to Accelerate Convolutional Neural Networks

Sayeh Sharify, Alberto Delmas Lascorz, Kevin Siu, Patrick Judd, Andreas Moshovos
University of Toronto
{sayeh,delmasl1,siukevi4,juddpatr,moshovos}@ece.utoronto.ca

ABSTRACT

Loom (*LM*), a hardware inference accelerator for Convolutional Neural Networks (CNNs) is presented. In *LM* every bit of data precision that can be saved translates to proportional performance gains. For both weights and activations *LM* exploits profile-derived per layer precisions. However, at runtime *LM* further trims activation precisions at a much smaller than a layer granularity. On average, across several image classification CNNs and for a configuration that can perform the equivalent of 128 $16b \times 16b$ multiply-accumulate operations per cycle *LM* outperforms a state-of-the-art bit-parallel accelerator [3] by 3.19× without any loss in accuracy while being 2.59× more energy efficient. *LM* can trade-off accuracy for additional improvements in execution performance and energy efficiency and compares favorably to an accelerator that targeted only activation precisions.

1 INTRODUCTION

Deep neural networks (DNNs) have become the state-of-the-art technique in many recognition tasks. Convolutional Neural Networks (CNNs) in particular dominate applications where the input is an image or video. Devices executing such CNNs will be required to perform mostly if not only inference. An example is computational photography where machine learning has shown great promise in replacing classical algorithms [10].

We present *Loom* (*LM*), a hardware accelerator for inference with CNNs targeting embedded systems. *LM* exploits the precision requirement variability of CNNs to reduce the memory footprint, increase bandwidth utilization, and to deliver performance which scales inversely proportional with precision for both convolutional (CVLs) and fully-connected (FCLs) layers. Ideally, compared to using a fixed precision of 16 bits, *LM* achieves a speedup of $\frac{256}{P_a \times P_w}$ and $\frac{16}{P_w}$ for CVLs and FCLs where P_w and P_a are the precisions of weights and activations, respectively. *LM* also reduces the number of weight and activation bits read by $\frac{16-P_w}{16}$ and $\frac{16-P_a}{16}$. To deliver these benefits *LM* processes both activations and weights bit-serially while compensating for the loss in computation bandwidth by exploiting parallelism. Judicious reuse of activations and weights enables *LM* to improve performance and energy efficiency over conventional bit-parallel designs without requiring a wider memory interface. For both weights and activations *LM* utilizes profile-derived per layer precisions. For activations, *LM* further trims their precision at a much finer granularity at runtime utilizing the approach of Lascorz *et al.* [8]. By exploiting precision *LM* delivers benefits for *all* activations and weights regardless of whether they are ineffectual or not.

We evaluate *LM* on an SoC and compare against a bit-parallel fixed-precision accelerator (*DPNN*) over a set of image classification CNNs. For a configuration that is sized to match the peak computation bandwidth of a bit-parallel accelerator that can perform at peak 128 $16b \times 16b$ multiply-accumulate operations per cycle, on average *LM* yields a speedup of 3.25×, 1.74×, and 3.19× over *DPNN* for the convolutional, fully-connected, and all layers, respectively. The energy efficiency of *LM* over *DPNN* is 2.63×, 1.41× and 2.59× for the aforementioned layers, respectively. *LM* enables trading off accuracy for additional improvements in performance and energy efficiency. For example, accepting a 1% relative loss in accuracy, *LM* yields 3.57× higher performance and 2.87× more energy efficiency than *DPNN*. We also perform a sensitivity study varying the equivalent peak compute bandwidth and the number of bits that *LM* processes per cycle. *LM* scales well up up to a configuration equivalent to 256 $16b \times 16b$ multiply-accumulate operations per cycle and that a 2-bit per cycle design achieves the best energy efficiency albeit not the best performance.

2 LOOM: A SIMPLIFIED EXAMPLE

This section explains how *LM* would process CVLs and FCLs on an example using 2-bit activations and weights.

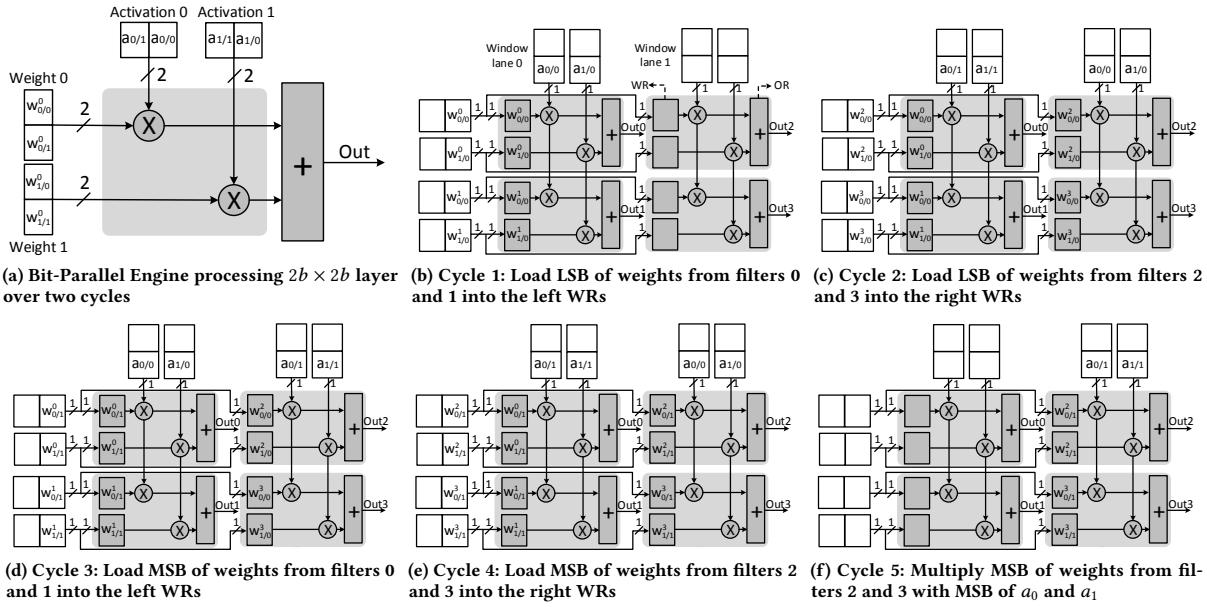
Conventional Bit-Parallel Processing: Figure 1a shows a bit-parallel processing engine which multiplies two input activations with two weights generating a single 2-bit output activation per cycle. The engine can process two new 2-bit weights and/or activations per cycle a throughput of two $2b \times 2b$ products per cycle.

***Loom*'s Approach:** Figure 1b shows an equivalent *LM* engine which matches the bit-parallel engine's throughput by producing 8 $1b \times 1b$ products every cycle. The engine comprises an 2×2 array of bit-serial subunits (4 in total). Each subunit accepts 2 bits of input activations and 2 bits of weights per cycle and performs 2 $1b \times 1b$ products. The subunits along the same column share the activation inputs while the subunits along the same row share their weight inputs. In total, this engine accepts 4 activation and 4 weight bits equaling the input bandwidth of the bit-parallel engine. Each subunit has two 1-bit Weight Registers (WRs), one 2-bit Output Register (OR) for accumulating its products.

Figure 1b through Figure 1f show how *LM* would process an FCL. As Figure 1b shows, in **cycle 1**, the left column subunits receive the least significant bits (LSBs) $a_{0/0}$ and $a_{1/0}$ of activations a_0 and a_1, and $w^0_{0/0}$, $w^0_{1/0}$, $w^1_{0/0}$, and $w^1_{1/0}$, the LSBs of four weights from filters 0 and 1. Each of these two subunits calculates two $1b \times 1b$ products (the product and accumulation would take place

DAC '18, June 24–29, 2018, San Francisco, CA, USA

ACM ISBN 978-1-4503-5700-5/18/06…$15.00
https://doi.org/10.1145/3195970.3196072



(a) Bit-Parallel Engine processing $2b \times 2b$ layer over two cycles

(b) Cycle 1: Load LSB of weights from filters 0 and 1 into the left WRs

(c) Cycle 2: Load LSB of weights from filters 2 and 3 into the right WRs

(d) Cycle 3: Load MSB of weights from filters 0 and 1 into the left WRs

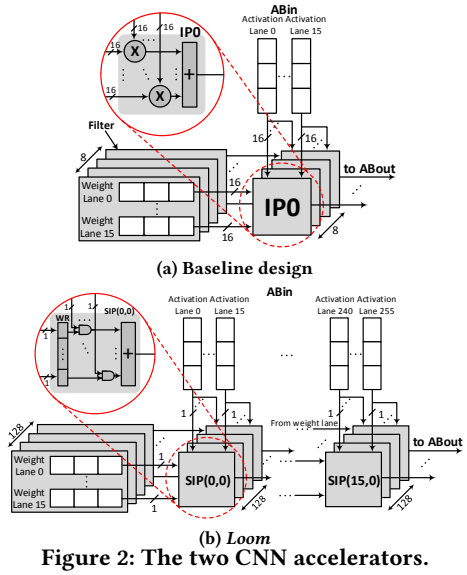
(e) Cycle 4: Load MSB of weights from filters 2 and 3 into the right WRs

(f) Cycle 5: Multiply MSB of weights from filters 2 and 3 with MSB of a_0 and a_1

Figure 1: Processing an example Fully-Connected Layer using *LM*'s Approach.

in the subsequent cycle adding one more pipeline stage, a detail the example omits for clarity) and stores their sum into its OR. In Figure 1c and **cycle 2**, the left column subunits now multiply the same weight bits with the most significant bits (MSBs) $a_{0/1}$ and $a_{1/1}$ of activations a_0 and a_1 respectively accumulate these into their ORs. In parallel, the two right column subunits load $a_{0/0}$ and $a_{1/0}$, the LSBs of the input activations a_0 and a_1, and multiply them by the LSBs of weights $w^2_{0/0}$, $w^2_{1/0}$, $w^3_{0/0}$, and $w^3_{1/0}$ from filters 2 and 3. In **cycle 3**, the left column subunits now load and multiply the LSBs $a_{0/0}$ and $a_{1/0}$ with the MSBs $w^0_{0/1}$, $w^0_{1/1}$, $w^1_{0/1}$, and $w^1_{1/1}$ of the four weights from filters 0 and 1. In parallel, the right subunits reuse their WR-held weights $w^2_{0/0}$, $w^2_{1/0}$, $w^3_{0/0}$, and $w^3_{1/0}$ and multiply them by the most significant bits $a_{0/1}$ and $a_{1/1}$ of activations a_0 and a_1 (Figure 1d). In **cycle 4** and Figure 1e, the left column subunits multiply their WR-held weights and $a_{0/1}$ and $a_{1/1}$ the MSBs of activations a_0 and a_1 and finish the calculation of output activations o_0 and o_1. Concurrently, the right column subunits load $w^2_{0/1}$, $w^2_{1/1}$, $w^3_{0/1}$, and $w^3_{1/1}$, the MSBs of the weights from filters 2 and 3 and multiply them with $a_{0/0}$ and $a_{1/0}$. In **cycle 5** and Figure 1f, the right subunits complete the multiplication of their WR-held weights and $a_{0/1}$ and $a_{1/1}$ the MSBs of the two activations. By the end of this cycle, output activations o_2 and o_3 are ready as well.

In total it took 4+1 cycles to process 32 $1b \times 1b$ products (4, 8, 8, 8, 4 products in cycles 1 through 5, respectively). Notice that at the end of the 5th cycle, the left column subunits are idle, thus the WRs could have loaded another set of weights commencing the computation of a new set of outputs. In the steady state, with $2b$ input activations and weights, this engine will be producing 8 $1b \times 1b$ terms every cycle thus matching the 2 $2b \times 2b$ throughput of the parallel engine. If the weights could be represented using only one bit, *LM* would be producing two output activations per cycle, twice the bandwidth of the bit-parallel engine.



(a) Baseline design

(b) *Loom*

Figure 2: The two CNN accelerators.

In general, if the bit-parallel hardware was using P_{base} bits to represent the weights while only P_w bits were actually required, for the FCLs the *LM* engine would outperform the bit-parallel engine by $\frac{P_{base}}{P_w}$. The *LM* would use an array of $P_{base} \times k$ units, where k the number of $P_{base} \times P_{base}$ products *DPNN* processes per cycle. Each subunit would produce k $1b \times 1b$ products. Since there is no weight reuse in FCLs, 16 cycles are required to load a different set of weights to each of the 16 columns. Thus having activations that use less than 16 bits would not improve performance (but could improve energy efficiency).

Convolutional Layers: *LM* processes CVLs similarly to FCLs but exploits weight reuse across different windows to exploit a reduction in precision for both weights and activations. Specifically, in

CVLs the subunits across the same row share the same weight bits which they load in parallel into their WRs in a single cycle. These weight bits are multiplied by the corresponding activation bits over P_a cycles. Another set of weight bits needs to be loaded every P_a cycles, where P_a is the input activation precision. Here *LM* exploits weight reuse across multiple windows by having each subunit column process a different set of activations. Assuming that the bit-parallel engine uses P bits to represent both input activations and weights, *LM* will outperform the bit-parallel engine by $\frac{P^2}{P_w \times P_a}$ where P_w and P_a are the weight and activation precisions *LM* uses respectively.

3 LOOM ARCHITECTURE

This section describes the baseline fixed precision bit-parallel accelerator and the *Loom* architecture.

3.1 Data Supply and Baseline System

Our baseline design (*DPNN*) shown on Figure 2a is an appropriately configured data-parallel engine inspired by the DaDianNao accelerator [3] the *de facto* standard used for comparison in most accelerator studies. *DPNN* uses 16-bit fixed-point activations and weights. *DPNN* comprises k inner product units (IP) each processing a different filter. Every cycle *DPNN* accepts as input N activations and N corresponding weights per filter out of k filters. In the configuration shown $N = 16$ and $k = 8$. The N activations are broadcast to all IP units. Each IP unit multiplies each of the N activations with one out of its N weights, reduces the resulting N 32b products with an adder tree, and accumulates the result into an output register. In total, every cycle, *DPNN* calculates $N \times k$ products producing k partial output activations.

An Activation Memory (AM) and a Weight Memory (WM) supply respectively the activations and the weights. An input activation buffer (*ABin*) buffers the input activations while an output activation buffer (*ABout*) temporarily buffers the output activations. For clarity, in our description we assume a single tile that processes up to 128 weights (8 filters) and 16 activations per cycle.

3.2 Loom

For *LM* to match our *DPNN* configuration it needs to process 128 filters concurrently and 16 weight bits per filter per cycle, for a total of $128 \times 16 = 2048$ weight bits per cycle. Alternatively, *LM* could process 32 filters over 64 windows, however, we leave this investigation for future work. *LM* also accepts 256 1-bit input activations each of which it multiplies with 128 1-bit weights thus matching the computation bandwidth of base in the worst case where both activations and weights need 16 bits. Figure 2b shows the *Loom* design. It comprises 2K Serial Inner-Product Units (SIPs) organized in a 128×16 grid. Every cycle, each SIP multiplies 16 $1b$ input activations with 16 $1b$ weights and reduces these products into a partial output activation. The SIPs along the same row share a common $16b$ weight bus, and the SIPs along the same column share a common $16b$ activation bus. Accordingly, as in *DPNN*, the SIP array is fed by a $2Kb$ weight bus and a $256b$ activation input bus. Similar to *DPNN*, *LM* has an ABout and an ABin. *LM* processes both activations and weights bit-serially.

Reducing Memory Footprint and Bandwidth: Since both weights and activations are processed bit-serially, *LM* can store weights and activations in a bit-interleaved fashion and using only as many bits as necessary thus boosting the effective bandwidth and storage capacity of the weight memory and the AM. For example, given 2K $13b$ weights to be processed in parallel, *LM* would pack first their bit 0 onto continuous rows, then their bit 1, and so on up to bit 12. *DPNN* would stored them using 16 bits instead. A transposer can rotate the output activations prior to writing them to AM from ABout. Since each output activation entails inner-products with tens to hundreds of inputs, the transposer demand will be low.

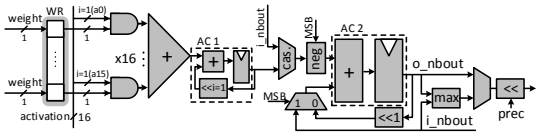


Figure 3: *LM*'s SIP.

Convolutional Layers: Processing starts by reading in parallel 2K weight bits from memory, loading 16 bits to all WRs per SIP row. The loaded weights will be multiplied by 16 corresponding activation bits per SIP column bit-serially over P_a^L cycles where P_a^L is the activation precision for this layer L. Then, the second bit of weights will be loaded into WRs and multiplied with another set of 16 activation bits per SIP row, and so on. In total, the bit-serial multiplication will take $P_a^L \times P_w^L$ cycles. where P_w^L the weight precision for this layer L. Whereas *DPNN* would process 16 sets of 16 activations and 128 filters over 256 cycles, *LM* processes them concurrently but bit-serially over $P_a^L \times P_w^L$ cycles. If P_a^L and/or P_w^L are less than 16, *LM* will outperform *DPNN* by $256/(P_a^L \times P_w^L)$. Otherwise, *LM* will match *DPNN*'s performance.

Fully-Connected Layers: Processing starts by loading the LSBs of a set of weights into the WR registers of the first SIP column and multiplying the loaded weights by the LSBs of the corresponding activations. In the second cycle, while the first column of SIPs is still busy with multiplying the LSBs of its WRs by the second bit of the activations, the LSBs of a new set of weights can be loaded into the WRs of the second SIP column. Each weight bit is reused for 16 cycles multiplying with bits 0 through bit 15 of the input activations. Thus, there is enough time for *LM* to keep any single column of SIPs busy while loading new sets of weights to the other 15 columns. For example, as shown in Figure 2b *LM* can load a single bit of 2K weights to SIP(0,0)..SIP(0,127) in cycle 0, then load a single-bit of the next 2K weights to SIP(1,0)..SIP(1,127) in cycle 1, and so on. After the first 15 cycles, all SIPs are fully utilized. It will take $P_w^L \times 16$ cycles for *LM* to process 16 sets of 16 activations and 128 filters while *DPNN* processes them in 256 cycles. Thus, when P_w^L is less than 16, *LM* will outperform *DPNN* by $16/P_w^L$ and it will match *DPNN*'s performance otherwise.

SIP: Bit-Serial Inner-Product Units: Figure 3 shows *LM*'s Bit-Serial Inner-Product Unit (SIP). Every clock cycle, each SIP multiplies 16 single-bit activations by 16 single-bit weights to produce a partial output activation. Internally, each SIP has 16 1-bit Weight Registers (WRs), 16 2-input AND gates to multiply the weights in the WRs with the incoming input activation bits, and a 16-input $1b$ adder tree that sums these partial products. AC_1 accumulates and shifts the output of the adder tree over P_a^L cycles. Every P_a^L cycles, AC_2 shifts the output of AC_1 and accumulates it into the OR. After $P_a^L \times P_w^L$ cycles the Output Register (OR) contains the

Table 1: Activation and weight (W) precision profiles in bits for the convolutional and fully-connected layers.

Network	Convolutional Layers			
	100% Accuracy		99% Accuracy	
	Act. / Per Layer	W	Act. / Per Layer	W
NiN	8-8-8-9-7-8-8-9-9-8-8-8	11	8-8-7-9-7-8-8-9-9-8-7-8	10
AlexNet	9-8-5-5-7	11	9-7-4-5-7	11
GoogLeNet	10-8-10-9-8-10-9-8-9-10-7	11	10-8-9-8-8-9-10-8-9-10-8	10
VGGS	7-8-9-7-9	12	7-8-9-7-9	11
VGGM	7-7-7-8-7	12	6-8-7-7-7	12
VGG19	12-12-12-11-12-10-11-11-13-12-13-13-13-13-13-13	12	9-9-9-8-12-10-10-12-13-11-12-13-13-13-13-13	12
	Fully-Connected Layers			
	Weights /Per Layer		Weights/Per Layer	
NiN	N/A		N/A	
AlexNet	10-9-9		9-8-8	
GoogLeNet	7		7	
VGGS	10-9-9		9-9-8	
VGGM	10-8-8		9-8-8	
VGG19	10-9-9		10-9-8	

inner-product of an activation and weight set. In each SIP, a multiplexer after AC_1 implements cascading. To support signed 2's complement activations, a negation block is used to subtract the sum of the input activations corresponding to the most significant bit of weights (MSB) from the partial sum when the MSB is 1. Each SIP also includes a comparator (max) to support max pooling layers.

Dynamic Precision Reduction: So far we assumed that software provided profile-derived per layer activation and weight precisions [5]. Lascorz *et al.,* observed that the hardware can further shorten these precisions by inspecting the actual values at runtime [8]. *LM* determines adjusts precision per group of 256 activations that it processes concurrently. Per bit position OR trees produce a 16-bit vector indicating the positions where any of the activations has a 1. A leading one detector identifies the most significant position and thus the precision in bits that is sufficient.

Processing Layers with Few Outputs: For *LM* to keep all the SIPs busy an output activation must be assigned to each SIP. This is possible as long as the layer has at least 2K outputs. However, in the networks studied some FCLs have only 1K output activations, To avoid underutilization, *LM*'s implements *SIP cascading*, in which SIPs along each row can form a daisy-chain, where the output of one can feed into an input of the next via a multiplexer. This way, the computation of an output activation can be sliced along the bit dimension over the SIPs in the same row. In this case, each SIP processes only a portion of the input activations resulting into several partial output activations along the SIPs on the same row. Over the next *Sn* cycles, where *Sn* is the number of bit slices used, the *Sn* partial outputs can be reduced into the final output activation.

Other Layers: Similar to *DaDN*, *LM* processes the additional layers needed by the studied networks. To do so, *LM* incorporates units for MAX pooling as in *DaDN*. Moreover, to apply nonlinear activations, an activation functional unit is present at the output of the ABout.

Total computational bandwidth: In the worst case, with 16b activations and weights, a single $16b \times 16b$ product that would have taken *DPNN* one cycle to produce, now takes LM 256 cycles. Since *DPNN* calculates 128 products per cycle, LM needs to calculate the equivalent of 256×128 $16b \times 16b$ products every 256 cycles. LM has $128 \times 16 = 2048$ SIPs each producing 16 $1b \times 1b$ products per cycle. Thus, over 256 cycles, LM produces $2048 \times 16 \times 256$ $1b \times 1b$ products matching *DPNN*'s compute bandwidth.

Tuning the Performance, Area and Energy Trade-off: We can trade off some of the performance benefits to reduce the number of SIPs and the respective area overhead by processing multiple activation bits per cycle. The evaluation section considers 2-bit (LM_{2b}) and 4-bit (LM_{4b}) *LM* configurations which need 8 and 4 SIP columns and accommodate precisions that are multiple of 2 and 4, respectively. For example, for LM_{4b} reducing the P_a^L from 8 to 5 bits produces no performance benefit, whereas for the LM_{1b} it would improve performance by 1.6×.

4 EVALUATION

This section evaluates *Loom* performance, energy and area and explores the trade-off between accuracy and performance comparing to *DPNN* and *Stripes* [6].

Performance, Energy, and Area Methodology: Execution time is modeled via a custom cycle-accurate simulator and energy and area measurements are collected over layouts of all designs. The designs were synthesized for worst case, typical case, and best case corners with the Synopsys Design Compiler using a TSMC 65nm library. Layouts were produced with Cadence Innovus using the typical corner case synthesis results which were more pessimistic for *LM* than the worst case scenario. Power results are based on the actual data-driven activity factors. The clock frequency of all designs is set to 1GHz. The ABin and ABout SRAM buffers were modeled with CACTI [11] and AM and WM were modeled as eDRAM with Destiny [13]. We first evaluate *LM* assuming that all the activations fit on chip and the weights can be read from off-chip memory without any bandwidth constraint to explore the design space without being affected by the choice of a particular off-chip memory. We conclude by investigating performance with a single-channel of low-power DDR4-4267.

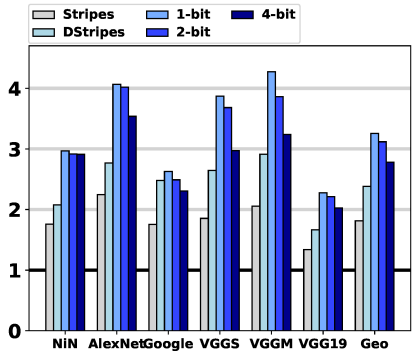
Weight and Activation Precisions: Table 1 reports the profile-derived per layer precisions of input activations and network precisions of weights for the CVLs and FCLs using the method of Judd *et al.* [5]. Since *LM*'s performance for the CVLs depends on both P_a^L and P_w^L, we adjust them independently. We use per layer activation precisions and a common across all CVLs weight precision. We found little inter-layer variability for weight precisions but additional per layer exploration is warranted. Since *LM*'s performance for FCLs performance depends only on P_w^L we only adjust weight precision for FCLs. The precisions that guarantee no *top-1* accuracy loss for CVLs input activations vary from 5 to 13 bits and for weights vary from 10 to 12. When a 99% relative *top-1* accuracy is still acceptable, the activation and weight precision can be as low as 4 and 10 bits, respectively. The per layer weight precisions for the FCLs vary from 7 to 10 bits.

Performance and Energy Efficiency: Figures 4a and 4b show respectively the performance and energy efficiency of *Loom*, *Stripes*, and *DStripes* configurations relative to *DPNN* with the precision 100% profiles of Table 1 and for all layers combined. *Stripes* is based on Stripes which exploits only profile-derived per layer activation precisions and only for CVLs [6]. *DStripes* incorporates dynamic prediction reduction [8].

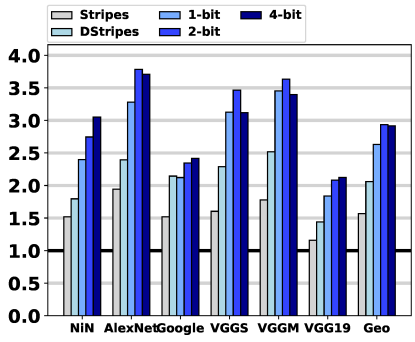
On average, LM_{1b} outperforms *DPNN* by more than 3× while being more than 2.5× energy efficient. When *LM* processes multiple

Table 2: Relative execution time speedup and energy efficiency with *Stripes* and *LM* for fully-connected and convolutional layers vs. *DPNN*.

Network	Stripes		*Loom* 1-bit		*Loom* 2-bit		*Loom* 4-bit	
	Perf	Eff	Perf	Eff	Perf	Eff	Perf	Eff
FULLY-CONNECTED LAYERS								
100% TOP-1 Accuracy								
NiN	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
AlexNet	1.00	0.88	1.65	1.34	1.66	1.56	1.66	1.74
Google	0.99	0.87	2.25	1.82	2.27	2.14	2.28	2.39
VGGS	1.00	0.88	1.63	1.32	1.63	1.54	1.63	1.71
VGGM	1.00	0.88	1.63	1.32	1.64	1.54	1.64	1.72
VGG19	1.00	0.88	1.62	1.31	1.63	1.53	1.63	1.71
Geomean	**1.00**	**0.88**	**1.74**	**1.41**	**1.75**	**1.65**	**1.75**	**1.84**
99% TOP-1 Accuracy								
Geomean	**1.00**	**0.88**	**1.85**	**1.49**	**1.85**	**1.75**	**1.86**	**1.95**
CONVOLUTIONAL LAYERS								
100% TOP-1 Accuracy								
NiN	1.76	1.54	2.97	2.40	2.92	2.75	2.91	3.05
AlexNet	2.34	2.04	4.25	3.43	4.20	3.96	3.66	3.84
Google	1.76	1.50	2.63	2.12	2.49	2.34	2.12	2.22
VGGS	1.89	1.65	3.98	3.21	3.78	3.56	3.02	3.17
VGGM	2.12	1.86	4.12	3.33	3.69	3.47	3.34	3.50
VGG19	1.34	1.17	2.17	1.76	2.09	1.97	2.03	2.13
Geomean	**1.84**	**1.61**	**3.25**	**2.63**	**3.10**	**2.92**	**2.78**	**2.92**
99% TOP-1 Accuracy								
Geomean	**1.99**	**1.74**	**3.63**	**2.93**	**3.45**	**3.25**	**3.11**	**3.26**



(a) *LM*'s performance relative to *DPNN*.



(b) *LM*'s energy efficiency relative to *DPNN*.

Figure 4: *LM*'s performance and energy efficiency relative to *DPNN* for all layers with 100% accuracy.

bits per cycle the performance benefits are lower but energy efficiency improves up to 2.9×. LM_{1b} consistently outperforms *Stripes* and *DStripes* in performance and *Stripes* in energy efficiency. LM_{1b} is more energy efficient than *DStripes* except for GoogleNet where its energy efficiency is within 2% of *DStripes*.

Table 2 reports per network performance and energy efficiency for *LM* configurations relative to *DPNN* for the FCLs and CVLs separately, and for the 100% and 99% accuracy profiles. In general, LM_{1b} outperforms LM_{2b} and LM_{4b} in most cases with the latter two being more energy efficient. On occasion the latter two outpeform LM_{1b} under the 100% accuracy profiles in FCLs. Since for *LM* the performance improvement in FCLs is only due to the use of lower weight precisions, processing multiple activation bits per cycle does not effect performance in the steady state. However, processing more activation bits per cycle reduces the initiation interval per layer an effect that becomes noticeable for small FCLs.

The table reports detailed results for *Stripes*. For FCLs, *Stripes* performance and energy efficiency suffer as it does not exploit weight precisions. With the 99% accuracy profiles, both performance and energy efficiency improve considerably for FCLs and CVLs. Performance with *DStripes* would be identical to *Stripes* for the FCLs. We do not present detailed results for *DStripes* due to space limitations noting that *LM* consistently outperforms *DStripes* while being more energy efficient except for the CVLs for GoogLeNet where the difference in energy efficiency is small.

Area Overhead: Post layout measurements were used to measure the area of *DPNN* and *Loom*. The LM_{1b} configuration requires 1.34× more area over *DPNN* while achieving on average a 3.19× speedup. The LM_{2b} and LM_{4b} reduce the area overhead to 1.25× and 1.16× while still improving the execution time by 3.05× and 2.74×, respectively. Thus *LM* exhibits better performance vs. area scaling than *DPNN*.

Scaling: Thus far we assumed that all activations fit on chip and focused on a single *LM* configuration. We next consider configurations with practical on- and off-ship memory hierarchies. Specifically, we size the activation memory so that most layers can fit on-chip avoiding off-chip accesses that today require at least two orders of magnitude more energy a critical consideration in embedded systems. Accordingly, *DPNN* requires 2MB of activation memory (VGG19 requires 10MB which is impractical for embedded systems and thus has to spill activations off-chip). Since *LM* processes both activations and weights bit-serially, it naturally stores and communicates values on- and off-chip using the per layer precisions. As a result, *LM* requires only 1MB on-chip memory for the activations. However, since *LM* processes more filters concurrently compared to *DPNN*, it can benefit from a larger weight memory.

Figure 5 shows how average performance over all networks scales for different configurations where the number of SIPs is chosen to match the peak compute bandwidth (x-axis) of a bit-parallel accelerator. For example, the "128" configurations can perform the equivalent of 128 $16b \times 16b$ multiply-accumulate operations per cycle. For each configuration Figure 5 reports performance relative to *DPNN* and absolute performance as frames per second (fps). The figure reports results for the convolutional layers only and also for all layers. This is done because fully-connected layers are off-chip bound (and thus are affected by our choice of off-chip memory)

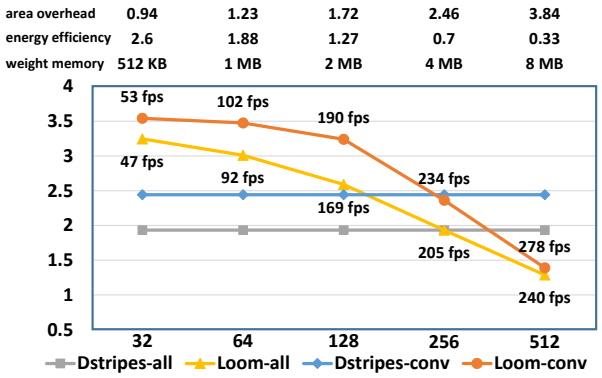


Figure 5: Scaling vs. equivalent *DPNN* peak compute bandwidth. Conv: convolutional layers only. All: all layers. All results with a LPDDR4-4267 off-chip memory.

whereas the convolutional layers are compute bound. Here we restrict attention to LM_{1b}.

LM outperforms *DPNN* for all design points shown and can achieve real-time processing rates even for the "32" configuration. The relative performance advantage of *LM* drops for the larger configurations since *LM* requires more parallelism and suffers more from increased underutilization as the number of weight lanes grows. *DStripes*'s relative performance over *DPNN* remains constant for the range shown. *LM* outperforms *DStripes* up to the "128" configurations. At "256" *LM* and *DStripes* perform nearly identically and at "512" the latter performs better.

The figure also reports the weight memory capacity, the relative (vs. *DPNN*) area overhead, and the energy efficiency for the various *LM* configurations. For the "64" and "32" configurations *LM* requires $128KB$ and $544KB$ less memory in total than *DPNN*. However, for the "128" and the "256" configurations *LM* requires more memory than *DPNN*. Regardless, the performance benefits exceed the relative area overhead and thus *LM* provides a better performance/area trade-off than *DPNN*. For the "256" configuration energy efficiency suffers with *LM*. However, this measurement ignores the energy of off-chip traffic which is on average 0.61× less with *LM*. Moreover, as CNNs evolve to process higher resolution images the size of activation memory increases significantly compared to the filter sizes which makes the effect of data compression more important [9]. Thus we expect that for higher resolution images *LM* will ever more appealing.

5 RELATED WORK

Due to space limitations, we limit attention to a few works that are the most related. We have already compared to *Stripes* [6] extended with dynamic prediction reduction [8].

Pragmatic's performance for the CVLs depends only on the number of activation bits that are 1, but does not improve performance for FCLs [1]. Further performance improvement may be possible by combining *Pragmatic*'s approach with *LM*'s but the costs per SIP may make this prohibitively expensive. *Proteus* exploits per layer precisions reducing memory footprint and bandwidth but requires crossbars per input weight [7]. *Loom* does not need crossbars. Hardwired NN implementations naturally exploit per layer precisions [14]. *Loom* does not require that the whole network fit on chip nor does it hardwire precisions. Furthermore, *Loom* further trims activations precisions at runtime.

Several accelerators target ineffectual weights and/or activations for dense and/or sparse networks [2, 4, 12, 15]. Most target either FCLs or CVLs alone. *LM* targets both layer types and benefits *all* inputs ineffectual or not.

6 CONCLUSION

This work presented *Loom*, a hardware inference accelerator for DNNs whose execution time for the convolutional and the fully-connected layers scales inversely proportionally with the precision p used to represent the input data. *LM* can trade-off accuracy vs. performance and energy efficiency on the fly. Future work may consider extending *LM* to further exploit weight sparsity.

REFERENCES

[1] Jorge Albericio, Alberto Delmás, Patrick Judd, Sayeh Sharify, Gerard O'Leary, Roman Genov, and Andreas Moshovos. 2017. Bit-pragmatic Deep Neural Network Computing. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-50 '17)*. 382–394.

[2] Jorge Albericio, Patrick Judd, Tayler Hetherington, Tor Aamodt, Natalie Enright Jerger, and Andreas Moshovos. 2016. Cnvlutin: Ineffectual-Neuron-Free Deep Neural Network Computing. In *2016 IEEE/ACM International Conference on Computer Architecture (ISCA)*.

[3] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and O. Temam. 2014. DaDianNao: A Machine-Learning Supercomputer. In *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*. 609–622.

[4] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. 2016. EIE: Efficient Inference Engine on Compressed Deep Neural Network. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA '16)*. IEEE Press, Piscataway, NJ, USA, 243–254. https://doi.org/10.1109/ISCA.2016.30

[5] Patrick Judd, Jorge Albericio, Tayler Hetherington, Tor Aamodt, Natalie Enright Jerger, Raquel Urtasun, and Andreas Moshovos. 2015. Reduced-Precision Strategies for Bounded Memory in Deep Neural Nets . *arXiv:1511.05236v4 [cs.LG]* (2015).

[6] Patrick Judd, Jorge Albericio, Tayler Hetherington, Tor Aamodt, and Andreas Moshovos. 2016. Stripes: Bit-serial Deep Neural Network Computing . In *Proc. of the 49th Annual IEEE/ACM Intl' Symposium on Microarchitecture*.

[7] Patrick Judd, Jorge Albericio, Tayler Hetherington, Tor M Aamodt, Natalie Enright Jerger, and Andreas Moshovos. Proteus: Exploiting numerical precision variability in deep neural networks. In *Proceedings of the 2016 International Conference on Supercomputing*. 23.

[8] Alberto Delmas Lascorz, Sayeh Sharify, Patrick Judd, and Andreas Moshovos. 2017. Dynamic Stripes: Exploiting the Dynamic Precision Requirements of Activation Values in Neural Networks. *CoRR* abs/1706.00504 (2017). arXiv:1706.00504 http://arxiv.org/abs/1706.00504

[9] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C. Berg. 2016. SSD: Single Shot MultiBox Detector. *arXiv:1512.02325 [cs.CV]* (2016).

[10] Rastislav Lukac. 2016. *Computational photography: methods and applications*. CRC Press.

[11] Naveen Muralimanohar and Rajeev Balasubramonian. 2015. CACTI 6.0: A Tool to Understand Large Caches. (2015).

[12] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W. Keckler, and William J. Dally. 2017. SCNN: An Accelerator for Compressed-sparse Convolutional Neural Networks. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*. 27–40.

[13] M. Poremba, S. Mittal, Dong Li, J.S. Vetter, and Yuan Xie. 2015. DESTINY: A tool for modeling emerging 3D NVM and eDRAM caches. In *Design, Automation Test in Europe Conference Exhibition*.

[14] T. Szabo, L. Antoni, G. Horvath, and B. Feher. 2000. A full-parallel digital implementation for pre-trained NNs. In *IJCNN 2000, Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks, 2000*, Vol. 2. 49–54 vol.2.

[15] Shijin Zhang, Zidong Du, Lei Zhang, Huiying Lan, Shaoli Liu, Ling Li, Qi Guo, Tianshi Chen, and Yunji Chen. 2016. 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*. 1–12.