BOVEDA: BUILDING AN ON-CHIP DEEP LEARNING MEMORY HIERARCHY BRICK BY BRICK

Isak Edo [1] **Sayeh Sharify** [1] **Daniel Ly-Ma** [1] **Ameer Abdelhadi** [1] **Ciaran Bannon** [1] **Milos Nikolic** [1] **Mostafa Mahmoud** [1] **Alberto Delmas Lascorz** [1] **Gennady Pekhimenko** [2 3] **Andreas Moshovos** [1 3]

ABSTRACT

Data access between on- and off-chip memories account for a large fraction of overall energy consumption during inference with deep learning networks. On-chip memory compression can greatly reduce this energy cost as long as it balances the *simplicity* and *low cost* of the *compression/decompression* implementation and its *effectiveness* in data size reduction. We present *Boveda*, a simple and effective on-chip *lossless* memory compression technique for fixed-point precision networks. It reduces data widths by exploiting the value distribution deep learning applications naturally exhibit. *Boveda* can increase the effective on-chip capacity, reduce off-chip traffic, and/or achieve a desired performance/energy target while using smaller on-chip memories. *Boveda* can be placed after any memory block in the on-chip memory hierarchy and can work with any data-parallel processing units such as the vector-like or the tensorcore units of modern graphics processors (Durant et al., 2017; Jia et al., 2018), systolic arrays such as that used in the Tensor Processing Unit (Jouppi et al., 2017), and units that process sparse tensors such as those used in the SCNN accelerator (Parashar et al., 2017). To demonstrate the potential of *Boveda*, we implement it over (i) SCNN, a state-of-the-art accelerator for sparse networks, (ii) a Tensorcore-like architecture, and (iii) TPU. *Boveda* reduces memory footprint by 34% for SCNN and sparse models on top of zero compression. For dense models, *Boveda* improves compression by 47%. We also present a prototype FPGA implementation.

1 INTRODUCTION

Data compression in the memory hierarchy is appealing for deep learning (DL) workloads and accelerators where memory accesses account for a large fraction of overall energy consumption (Dally, 2011; Horowitz, 2014b). First, data compression at any level of the hierarchy boosts effective capacity by encoding each value using fewer bits. Second, it improves effective latency and energy efficiency by reducing access to slower and more energy-demanding higher hierarchy levels and off-chip memory. Third, it reduces the number of read or written bits per value, boosting effective bandwidth and energy efficiency. Finally, it complements dataflow and blocking for data reuse, which are the front-line techniques for boosting energy efficiency in the memory hierarchy for deep learning.

The Need for On-Chip Memory Compression: Since off-chip accesses have been more than an order of magnitude costlier than on-chip ones for quite some time now (e.g., accessing 32b from DRAM costs 640pj vs. 5pj from an 8KB SRAM (Horowitz, 2014a)), the first compression methods for neural networks (Han et al., 2015; Lascorz et al., 2019; Rhu et al., 2018), targeted off-chip accesses primarily. Their success has increased the relative energy cost of on-chip accesses. Accordingly, we complement past work on off-chip compression, data reuse, and blocking by exploring compression in the on-chip memory hierarchy (Chen et al., 2016; Gao et al., 2019; Rhu et al., 2018).

On- vs Off-Chip Accesses: Compared to compression methods for off-chip accesses, on-chip memory compression presents different trade offs: 1) On-chip access is faster and requires much less energy (the difference being at least an order of magnitude for both metrics) (Horowitz, 2014a). This severely limits the energy and complexity/latency costs afforded by on-chip compression methods. 2) Since DL hardware tends to favour wide data-parallel units, the on-chip memory hierarchy has to support much wider (bits) than off-chip accesses to keep these units busy. 3) On-chip data throughput is drastically higher than off-chip accesses due to on-chip data reuse (Chen et al., 2016). 4) While off-chip memory interfaces are standardised, on-chip memory organisation depends on the specific design. This affords us more flexibility.

The Need and Opportunity for DL-Specific On-Chip Compression: Data compression in the memory hierarchy has received extensive attention in the context of

[1]Department of Electrical & Computer Engineering, University of Toronto [2]Department of Computer Science, University of Toronto [3]Vector Institute. Correspondence to: Isak Edo <isak.edo@mail.utoronto.ca>.

Proceedings of the 4th MLSys Conference, San Jose, CA, USA, 2021.

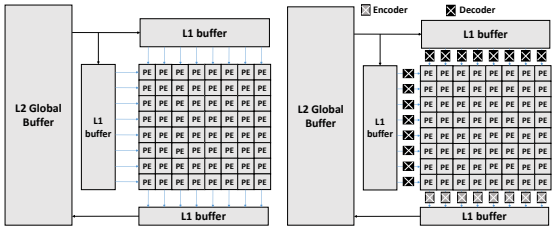
general-purpose systems, including compressed on-chip cache (Alameldeen & Wood, 2004b; Hallnor & Reinhardt, 2005; Qureshi et al., 2005) and main memory designs (Abali et al., 2001; Ekman & Stenstrom, 2005; Hong et al., 2018; Hong et al., 2019; Pekhimenko et al., 2013; Young et al., 2019), and hardware-based compression algorithms (Alameldeen & Wood, 2004a; Arelakis & Stenstrom, 2014; Arelakis et al., 2015; Pekhimenko et al., 2012). Section 5.1 compares with several such methods. The key challenge for on-chip compression algorithms is the need to balance the *simplicity* of the design/implementation and *low compression/decompression latency* with *effectiveness* in data size reduction. We make three observations that allow us to find a sweet spot in such a tradeoff in the context of DL hardware designs:

1) The need to efficiently support random access pattern, using *indexing* primitive, that severely restricts general-purposed compression designs (e.g., the use of indirection to locate the required data (Alameldeen & Wood, 2004a)) can be avoided because deep learning workloads predominantly access data in large sequential blocks to boost data reuse.

2) In contrast to general-purpose processors, which perform a few narrow L1 data cache accesses per cycle, DL workloads favour data-parallel processing units with memory systems that can serve multiple *wide and parallel* accesses (e.g., 10s to 100s of bits per access). A typical general-purpose compression engine only needs to decode a single value per access, and hence only needs a fast way to determine where that *single value* is stored, and an interconnect that can quickly and efficiently move this narrow data. For deep learning, the need to decompress several data elements concurrently is crucial, but unfortunately, this severely limits encoding flexibility—i.e., special care is needed to perform an *efficient data layout* that would simplify the interconnect to provide the required parallelism.

3) Neural network values, dominated by feature and filter maps, do *not* necessarily exhibit the properties of typical program variables, such as pointer prefixes or commonly occurring values (Alameldeen & Wood, 2004a; Arelakis et al., 2015; Pekhimenko et al., 2012). Instead, these values are usually short fixed-point values (e.g., 8b) with value distributions that are heavily biased towards zero or near zero. Despite this value content, most systems use one datawidth to store all values regardless of content (e.g., use 8b to store the value 0x1). There is significant opportunity to reduce storage needs by somehow adapting datawidth to be only as long as necessary to fit the value content.

This Work: Utilising these observations, we present *Boveda*, a *lossless* on-chip memory compression method that works with *unmodified* neural networks. *Boveda* is directly compatible with *any* accelerator that uses data-parallel processing units such as those in graphics processors including Tensorcores with and without support for sparsity (Durant et al., 2017; Jia et al., 2018; NVIDIA, 2020), systolic-arrays such as those of the TPU family (Jouppi et al., 2017), grid-like processing units without or with sparsity support such as SCNN (Parashar et al., 2017). As Figure 1 shows for an example systolic-array accelerator, *Boveda* sits in-between the on-chip memory hierarchy and the processing units where it decodes and encodes values as they are being read or written by the compute units. *Boveda* works with any desired dataflow transparently packing data in memory so that they can be fed into the corresponding units with little cost, avoiding expensive, long-distance lateral movement. It sustains high *concurrency* and wide *datawidth*. The memory hierarchy still sees a regular stream of as wide as before accesses, albeit one containing fewer accesses. The units see a regular stream of incoming data containing exactly the same values as if *Boveda* was not there.



(a) Baseline Accelerator (b) Boveda Additions

Figure 1. Incorporating *Boveda* into an accelerator.

Taking Advantage of the Naturally Occurring Value Distribution: To achieve high compression ratios, *Boveda* exploits the typical value *content* of neural networks that use fixed-point values (observation 3 above). *Boveda* adjusts datawidth to value content. For example, *Boveda* may use just 3b to store `0x2` and 5b to store `0x1b`. Unfortunately, allowing each value to select its datawidth independently would result in unacceptable metadata overhead (e.g., a width field per value) and, even worse, would result in an unacceptably expensive interconnect to unpack the values.

Balancing Metadata Cost and Compression Ratio: The first design choice in *Boveda* is straightforward and reduces metadata cost while getting most of the benefits from adapting datawidth to value content. *Boveda* groups values and selects a common datawidth that is sufficiently wide to accommodate the value with the highest magnitude in the group. For example, for a group of eight 8b values where the highest magnitude value is `0x12`, *Boveda* would use a container of 8×5b, whereas, for another group where the maximum magnitude value is `0x0a`, it will use 8×4b. In either case, a metadata field of 3b specifies the datawidth.

The Need for a Hardware Efficient Data Layout: However, this grouping alone is not only insufficient, worse it results in an overall increase in energy. The reason is that the interconnect needed to read the compressed values, expand

them, and route them to the execution units turns out to be very expensive. We studied such a conventional design and found that it would increase overall energy. Indicatively we measured energy for a processing unit with 16 8b multipliers and two banks of 8KB SRAM, one per multiplier input side. The energy needed per access for the interconnect that expands the values was 78% of the energy needed per access. A compression rate of 44% would be needed just to break-even, and that ignores any metadata overhead. Constraining datawidth to powers of two helps but only slightly moving the break-even point at 42% compression ratio.

Boveda's Data Layout: *Boveda*'s second design avoids such expensive interconnects enabling energy savings. *Boveda* organises data in such a way so that they are packed in columns that are aligned with their corresponding processing unit inputs. This allows *Boveda* to use an ensemble of lightweight per column encoder/decoder units.

Boveda boosts the effective on-chip capacity without requiring any modifications to the neural network model. This can yield energy and/or performance benefits depending on whether the model is off-chip or compute bound. At design time, *Boveda* reduces the amount of on-chip memory needed to meet the desired performance target. To a neural network developer, *Boveda* presents a system that needs to go off-chip less often and that rewards quantisation without requiring it for all models. We study these effects by considering several design points and quantisation methods. We demonstrate that *Boveda* is a generic technique that can be used with many accelerators, including an accelerator that can exploit sparsity and process sparse tensors. Specifically, we demonstrate *Boveda* over Tensorcore-like units, over a TPU-like systolic array, and SCNN, an accelerator for pruned models. Finally, we incorporate *Boveda* over an FPGA accelerator and demonstrate a working prototype.

We highlight the following experimental findings: a) *Boveda* compresses 60% and 40% more data than two prior state-of-the-art compression algorithms (Alameldeen & Wood, 2004a; Panda & Seznec, 2016). b) *Boveda* reduces the total model footprint to 53% and the volume of bits accessed on-chip to 55% on average for 8b models across different application domains. c) For a 4b model, it reduces average width to 2.6b, including metadata. d) It reduces total energy consumption by 23%, 26% and 17% for a Tensorcore-like accelerator, SCNN, and TPU, respectively.

2 Challenges and Opportunities

Compression in the memory hierarchy has received considerable attention, especially in the context of general-purpose systems. Section 5.1 considers and compares several such methods which we adapted for deep learning workloads. Here we comment further on the different set of challenges and opportunities that exist for on-chip compression methods for deep learning workloads.

Indexing: General-purpose systems must support random, fine-grain accesses. This requires the ability to locate the compressed values in memory both quickly and at a fine-grain granularity. This favours compression methods that use small containers. To support random access some compression methods reduce the amount of data transferred but not the size of the containers they use in storage. For example, they encode data within a cache line so that it needs to read or write fewer bits. However, the full cache line is still reserved. Thus they only improve bandwidth but not effective capacity.

In contrast, deep learning workloads exhibit long sequential accesses as they are dominated by matrix/vector or matrix/matrix operations. Blocking for data reuse also results in long sequential streams on- and off-chip. It merely breaks large tensors into smaller yet still sizeable sub-tensors, which are too accessed sequentially. Accordingly, there is no need to support random accesses to fine-grain blocks of compressed data. For the occasional data structure that may need random access, we can simply disable compression. It is convolutional layers that are the most demanding in terms of indexing across all layer types encountered. Section 3.1 discusses support for these layers in more detail.

Concurrency and Payload: Memory hierarchies for general-purpose systems need to support a few narrow processing cores, whereas deep learning favours data- parallel execution massively. This needs a *highly concurrent* and *wide payload* memory hierarchy. When data is not compressed, this is easily achieved by using several wide on-chip memory banks. Individual data values can be laid out in those memories to align directly with the corresponding functional unit inputs obviating *lateral* data movement. However, once data is compressed, this alignment will be broken. As wire delay and energy are major considerations in current technology nodes, care must be taken to avoid as much as possible data movement over long distances.

Content: Compression methods for general-purpose systems capitalize on value behaviours found in "typical" programs, such as full or partial value redundancy. For example, memory pointers tend to share prefixes (e.g., pointers to the stack or heap-allocated structures). Programs often use aggregate data structures that tend to exhibit partially repeated value patterns (e.g., flag fields). Compression methods need to handle various data types, including integers and floating-point numbers or characters from various character sets (e.g., UTF-16). Further, programs manage datatypes of various power-of-two datawidths, such as 8b, 16b, 32b or more. Finally, programmers often use the "default" integer or floating-point datatypes (32b or 64b today). Compression methods capitalise on these characteristics.

The bulk of values in deep learning workloads are for fmaps and imaps, large arrays of short fixed-point values such as 16b or 8b with even 4b possible in some cases (Choi et al., 2018). However, there are models for which 16b is still necessary, e.g., for certain segmentation models where even small drops in accuracy translate in obvious artefacts. Regardless, as Section 5.1 shows, attempting to compress data using methods developed for "typical" programs yields unsatisfactory results. Fortunately, however, imaps and fmaps tend to exhibit a value distribution that is heavily biased towards zero or a value near zero.

Conventional memory hierarchies do not capitalize on this property as they store *all* imap or fmap elements using the datawidth needed for the *largest* magnitude *possible*. This is excessive for most values and across all networks studied. This behaviour is exhibited by all models studied. Appendix B highlights two such cases: ResNet18 (image classification) (He et al., 2015), and SSD_MobileNet (object-detection), both quantised to 8b (Reddi et al., 2019). *Boveda* capitalizes on these distributions to store elements using a number of bits (datawidth) that is just long enough to fit their current value.

Cost Amortization: To amplify bandwidth and capacity benefits, we favour an *on-chip* compression scheme where data remains encoded as much as possible. Preferably, we would like to decompress data just before the functional units, which favours simple to implement schemes, especially for decoding. Many compression methods for general-purpose systems operate between the last-level cache and other caches of the on-chip hierarchy where latency is not as critical and can tolerate additional complexity. The cost of (de)compression has to be carefully controlled as, ultimately, the goal is to improve energy efficiency.

First, we need to consider the cost of metadata, that is, of the additional information stored by the compression scheme. Rather than storing each value with a different datawidth, *Boveda* groups them and uses a sufficiently long datawidth for all values within the group. This delivers most of the benefits while keeping metadata costs low.

Second, we need to consider the cost of data decompression, compression and routing. The encoding and decoding of data have to be lightweight enough to reduce energy from having fewer bits from memory vs. the energy cost of decoding the original value ends up being a net win. It is for this reason that *Boveda* avoids any other data manipulation besides datawidth adaptation.

The cost of routing values is another important consideration. Of particular concern is *lateral* data movement, given that deep learning hardware tends to favour wide memories. Where in a memory row data is stored relatively to where it needs to go to (functional unit wires) can significantly impact energy costs. If the data layout is not carefully planned, encoded bits will have to traverse potentially the full datawidth of access resulting in a net increase in energy.

Summary: We wish to develop a lossless on-chip compression scheme which: 1) can support the relatively long sequential accesses needed by neural networks, 2) can support multiple wide accesses to maintain high utilisation of processing units, 3) is simple enough so that decoding can happen just before the processing units, thus keeping data compressed for as long as possible, 4) avoids lateral movement of bits over long wires, 5) takes advantage of value behaviour that is typical of neural networks, and 6) uses an encoding and decoding method that is low cost so that overall it results in a net energy win.

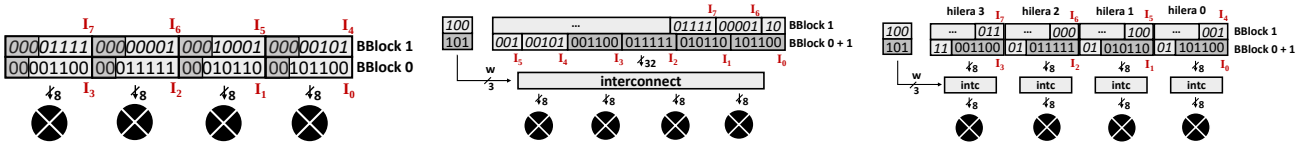
3 BOVEDA COMPRESSION

Whether the imap or fmap is a 1D, 2D, or 3D matrix, it will be mapped onto a flat, 1D address space when laid out in memory. When this matrix is processed by the data parallel units of deep learning hardware, the memory will see a sequence of wide accesses. For example, a processing tile with 32 8b multiply-accumulate units will need 2x32x8b=2x256b inputs per cycle, 256b for imaps and 256b for fmaps. Without compression, two memory buffers, each 256b wide could be used to supply these values. Here we describe how *Boveda* can be used to reduce traffic and footprint to such buffers. The buffers will still see 256b wide accesses. However, there will be fewer of those since in the common case, *Boveda* will be using fewer bits per value. To illustrate that *Boveda* is compatible with *any* processing tile that makes wide accesses to on-chip buffers, we describe three such configurations: 1) SCNN (Parashar et al., 2017), a state-of-the-art accelerator for the convolutional layers of *pruned* models, 2) Tensorcore- and, 3) TPU-like designs.

3.1 Boveda over SCNN

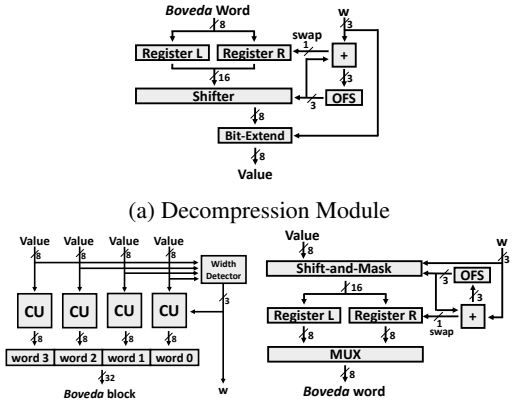
Since SCNN executes fully-connected layers at reduced efficiency, we limit our discussion to convolutional layers in this section. The inputs are fmaps (weights which are statically known), imaps (activations which are runtime calculated), and the ouputs are the omaps (activations). Appendix E explains convolutional layers in more detail.

SCNN Primer: SCNN stores values in an N.SAMPLES-CHANNEL-HEIGHT-WIDTH (NCHW) order and the omap is calculated by a spatial input stationary convolution. This allows SCNN to process imaps and fmaps one channel at time and to exploit sparsity. Figure 9 in Appendix E shows the organisation of an SCNN tile. For our discussion, it suffices to know that: 1) The tile has three buffers respectively holding imaps (and omaps), fmaps, and partial sum omap accumulators. 2) Each cycle, at maximum throughput, the



(a) Fixed Datawidth Buffer (b) Naïve Layout w/ Variable Datawidths (c) *Boveda*'s Layout

Figure 2. Avoiding Long-Distance Lateral Data Movement via the *Boveda* memory layout.



(a) Decompression Module

(b) Compressor Module (c) Compacting Unit

Figure 3. Boveda Decompression and Compression Modules.

tile reads 4 imap values and 4 fmap values from their buffers, 3) calculates the products for all 16 possible (imap,fmap) pairs, and 4) via a crossbar accumulates these products into their corresponding partial omaps. To take advantage of sparsity, the imap and the fmap omit zero values storing non-zero values as $((value),(skip))$ pairs where $(skip)$ is the number of zero values omitted after each. By using these $(skip)$ fields, SCNN deduces each value's original position and maps the products to their respective accumulators. We omit the skip fields in our discussion for clarity and assume 8b values common today.

Original Layout and Processing: Let us consider how the original SCNN would process two consecutive blocks $(I_0,...,I_3)$ and $(I_4,...,I_7)$ of 4 imap values each. Henceforth, called *BBlock* (*Boveda* block) 0 and 1. Initially let us assume that these are unsigned numbers. As Figure 2a shows, SCNN's imap buffer uses 8b per value and supports 4-value-wide reads (32b). With this organisation the values as read from the imap buffer align directly with the multiplier inputs. However, all values in *BBlock* 0 have a prefix of at least 2 zero bits, and those in *BBlock* 1 have of 3 bits. *Boveda*'s goal is to avoid storing these prefix bits.

Conventional Variable Datawidth Layout: Figure 2b shows a straightforward yet undesirable way to store the compressed values. For each *BBlock* of four values a *width* field specifies bitwidth per value, 5 for *BBlock* 0 and 6 for *BBlock* 1 (encoded as 4 and 5); a single width field per *BBlock* amortises its overhead over multiple values. Decompression comes at a hefty price because the values are no longer aligned with the multiplier inputs and may even spread over two rows. Let us first consider the misalignment of values and multiplier inputs. We need to extract *width* bits (varies per *BBlock*) and route those to a multiplier input after expanding to 8b. This routing requires a 32b-to-8b crossbar-like interconnect. Four such crossbars are needed one per multiplier column, a significant cost in area and energy. For a larger 8×8 multiplier grid 64b-to-8b crossbars would have been needed.

***Boveda*'s approach** is of much lower complexity and cost. *Boveda* treats the values as belonging to one of four *hileras* which correspond to multiplier columns; the first value in each *BBlock* belongs to *hilera* 0, the second value to *hilera* 1, and so on. The approach of Figure 2b breaks this mapping and allows compressed values to flow freely across *hileras*. *Boveda* instead restricts values to stay within their original *hilera* as Figure 2c shows. I_0 and I_4 are packed together into the *hilera* mapped onto the buffer's first 8 bits whereas I_3 and I_7 are packed into the *hilera* mapped onto the last 8 bits. To draw an analogy, *Boveda* uses values as *bovedillas* (bricks) to fill its *hileras*. The "crossbars" needed are now 8b-to-8b; their size depends only on the maximum datawidth and is independent of the number of values read per cycle; an 8×8 multiplier grid would require eight 8b-to-8b crossbars instead of eight 64b-to-8b ones.

Decompression Module: Figure 3a shows a decompression block. It decompresses a single value per cycle properly handing those values that spread over two rows. Four such blocks operate in parallel to decompress four values per cycle. Reads from the buffers remain 32b wide. From each read, each module grabs the corresponding 8b for its *hilera*. Within the module, two 8b registers L and R hold compressed data. Every time a new set of 8b is read in, it is written into L while simultaneously the current contents of L are "copied" into R. Rather than physically copying L into R a bit pointer (not shown) is used to "swap" the two. In steady state L and R will contain two consecutive rows from one *hilera* from the imap buffer and thus all bits necessary to decompress an 8b value regardless of width. A 16b-to-8b shifter extracts the current value from the value formed by concatenating L and R. The shifter only needs to support shifts up to 7 positions left and as specified by a 3b "offset" register, *OFS*. A 3b register W holds the data

width of the current *BBlock*. *OFS* and *W* and the associated control logic is shared among *all* four decompression modules. Initially, *OFS*=0 and *W*=7 both corresponding to the maximum datawidth. A "Bit-Extend" block passes the *W* LSbs from the shifter's output and sign-extends them to 8b. The module operates as a two-stage pipeline where the first stage loads values into *L* and *R* while the second stage extracts the next decompressed 8b value from the contents of *L* and *R*. The module requires 3 cycles in total (an extra cycle is needed for the initiation interval) for the first multiplier column to decompress I_0 and I_4. In the steady-state the module will output a value per cycle. Parts (b) and (c) of the figure show cycles 2 and 3. Appendix D details the decoding process for I_0 to I_4 in detail.

Compressing Values: Once all imap and fmap values for all channels of a layer are processed, the accumulators contain the output map. SCNN reads out these values, passes them through the activation function, removes zero, and copies the remaining into the omap buffer (then swaps a pointer so that the omap buffer becomes the imap buffer for the next layer). *Boveda* taps on the output of the zero compression module. The number of values per *BBlock* can be chosen freely at design time. Figure 3c shows a *Boveda* compressor module that processes four input values per cycle and where the *BBlock* size is four. Figure 3b shows that the compressor consists of three major components: (1) a width detector, (2) four compactor units (CUs), and (3) a 32b output register. The compression module reads in four 8b values per cycle and encodes them into a *BBlock*, storing them into the output register. When all 32b of the register are filled in, it sends it to the omap buffer. It is fully capable of producing a full row per cycle. In practice, however, it will output buffer rows at a slower pace, since *Boveda* compression will allow it to pack more values per row. For this reason, fewer bits will have to be copied into the imap buffer, saving energy.

The width detector identifies the bit width necessary for the highest magnitude value. Let us assume for a moment that the values are positive (true when using ReLU). The detector first produces 8 signals, one per bit plane, each being the OR of all corresponding bits across the four values. A leading-one detector identifies the MSb among those 8 signals that is 1. This is the width the *BBlock* needs and is written into the width buffer. For layers that may have signed, all that is needed is to invert them before the detector. The width, in this case, needs one more bit for the sign.

Figure 3c shows the structure of a compacting module, which mirrors the decompression module. There is one compacting module per *hilera*. Registers *L* and *R* hold the current and the next row for the *hilera*. Every cycle, the module processes a value. It extracts its *width* (detector) least significant bits and via the "shift-and-mask" block, and stores them into *R* appropriately. If the value requires more bits than those currently left unused in *R* the remaining bits are written into *L*. When *R* fills up, it is copied to the output row register (component (3)), and the two registers are swapped using a single bit pointer (not shown). A 3b *OFS* register specifies at which bit position filling *R* should continue. The shift-and-mask block contains an 8b-to-16b shifter and needs to support shifts up to 7 positions to the right (we never need to shift more than 7 bits since that would mean that *R* had no free bits left).

Tiling fmaps and imaps: SCNN sizes its on-chip buffers so the imap and the omap per layer fit on-chip and reads fmaps from off-chip in channel order. When there are multiple tiles, each imap channel is mapped onto the tiles in equally sized portions and the fmaps are broadcast. The imap portion sizing depends only on the layer dimensions. However, since SCNN uses zero compression the number of imap values per portion will vary. *Boveda* is directly compatible with this arrangement: processing still starts at the beginning of the imap buffer and, when values are written at the output of the layer, they are placed starting at the first position of the local omap buffer (which is becomes the imap for the next layer). SCNN stores fmaps channel first packing the values for all fmaps together. The tiles cycle through all fmap values in channel order. SCNN can determine when it reaches the end of each channel since the fmaps and the count of values they contain are known statically. *Boveda* is directly compatible with this processing order.

Width and Zero Skip Fields: SCNN uses a per value skip field to remove zeroes. Since the skip fields are used only in the control logic of the tile (e.g., to determine the original positions of values), it is better to store them into a separate structure next to the control logic rather than close to the datapath. We widen this buffer also to store the per *BBlock* width fields. If we assume, skip fields of 3b, and 8b values then the width field requires an overhead of 3b per *BBlock*, or less than 7% with *BBlocks* of 4 values.

3.2 Boveda over Tensorcores+

We extend a Tensorcore-like architecture (Durant et al., 2017; Jia et al., 2018) with *Boveda*. This architecture does not target pruned models but can execute any type of layer. The accelerator has a global buffer to reduce off-chip accesses and a grid of tensorcores (TCs). The TCs can process a [4x4]×[4x4] matrix multiply per cycle producing a [4x4] omap. Every cycle, the tensorcores read 4x4 imaps, 4x4 fmaps and 4x4 partial omaps values in parallel. Each TC has its own local imap, fmap and omap buffers. The TC's local memories read values from the global buffer at which point they are decompressed. Omap values are compressed before writing them to the global buffer. The width fields use a separate bank and address space of the global buffer.

There are two key differences vs. the SCNN implementation: (a) On-chip *Boveda* does not need to implement zero compression. (b) For energy efficiency, *Boveda* has to work with the diverse set of dataflows and the resulting data blocking at various levels of the i/f/o/maps that Tensorcores support. This requires being able to locate the starting point for each reuse block as needed by the dataflow. Supporting other dataflows requires additional support since, with *Boveda* the mapping of values to memory addresses becomes content dependent. *Boveda* uses pointers to support the blocking scheme of the chosen dataflow. Fortunately, only a few pointers are needed, and only a few of them have to be explicitly stored when the data is compressed on- or off-chip. Pointers can be generated in a timely fashion while processing and can be discarded once used. This is possible because: (a) dataflows use blocking to maximise reuse, and (b) as processing proceeds according to the dataflow, the accelerator naturally encounters the starting positions for the reuse block(s) that will be processed next. Appendix F explains this process in more detail.

Alignment: To perform wide reads, we restrict the starting positions for some *BBlocks* to align with rows in the on-chip memories. This depends on the dataflow. For our experiments, this alignment is enforced 1) at the first value of every fmap, and 2) every S channel values along a single (x, y) column of the imap (where S is the stride). Padding is occasionally needed but never ends up increasing footprint compared to no compression.

Other Layers: *Boveda* works well with any other layer such as depthwise separable convolutions, fully-connected, transformer, and pooling. Since each *BBlock* can be decoded in parallel, *Boveda* will need to store $P \times blocksize$ pointers and align them to initiate P operations in parallel.

3.3 Boveda over the TPU

The first generation Tensor Processing Unit (TPU) (not enough is publicly known about later generations) is a large accelerator designed for datacenters. It uses a systolic array, of 256x256 MAC units and aims to hold all imap values on-chip to maximize systolic array utilization. To keep the systolic array utilised, the TPU's on-chip memory buffers can sustain two wide reads per cycle. We insert *Boveda* decompressor between the buffers and the systolic array, and compressors at the output. Appendix G reviews the TPU and details the *Boveda* extensions.

4 Related Work

Related work falls into the following categories: compression methods for general-purpose systems and domain specific methods for machine learning. We limit attention to methods targeting specifically deep learning. Gist presents three imap compression methods during the backward pass when training on GPUs (Jain et al., 2018): two lossless for ReLU-pool and ReLU-convolution layer pairs, and one lossy. *Boveda* targets inference and is lossless. It relies on the expected distribution of *all* values, and while it benefits from sparsity, it does not require it. The Efficient Inference Engine (EIE) uses Deep Compression (Han et al., 2015) to drastically reduce fmap sizes for fully-connected layers (Han et al., 2016). Deep Compression alters the fmap to use a limited set of values (originally 16), uses Huffman encoding and lookup tables to decode values at runtime. *Boveda* operates with out-of-the-box networks. For models where Deep Compression can be applied, it will naturally outperform *Boveda*. However, Deep Compression is a specialised method. Compressing DMA uses a bit vector per block to remove zero values *off-chip* (Rhu et al., 2018). *Boveda* targets on-chip compression and *all* values. ShapeShifter is an *off-chip* compression method which like *Boveda* adapts the data container to value content and also uses a zero bit vector (Lascorz et al., 2019). ShapeShifter's containers are stored sequentially in memory space with no regards to alignment. Decompression per block is done a value sequentially at a time per block. For these reasons, it is not appropriate for on-chip compression. Diffy extends ShapeShifter by storing values as deltas (Mahmoud et al., 2018). Diffy targets computational imaging neural networks where the imap values exhibit high spatial correlation. It is more expensive than *Boveda* as encoding and decoding require calculating deltas. Proteus stores values on- and off-chip using profile-derived per layer data widths and cannot exploit the lopsided distribution of the values within the layer (Judd et al., 2016); the maximum magnitude per layer dictates the width for all its values. *Boveda* adapts the data width at a finer granularity.

5 Evaluation

Running experiments through RTL simulation is too time-consuming and constrains design space exploration. Hence, a custom cycle-accurate simulator models execution time and energy, while RTL is used for validation. The simulator uses DRAMSim2 (Rosenfeld et al., 2011) to model off-chip memory accesses. All accelerators and hardware modules are implemented in Verilog, synthesised with the Synopsys Design Compiler and laid out with Cadence Innovus for a TSMC 65nm cell library due to licensee constraints. Power is estimated via Innovus using the circuit activity reported by Mentor Graphics ModelSim. CACTI (Muralimanohar et al., 2009) models the area and power of the on-chip memories. All accelerators operate at 1GHz matching CACTI's speed estimate for the on-chip memories. Table 3 in Appendix C lists the network models studied, which cover a wide spectrum of applications and include four models from MLPerf (Reddi et al., 2019). Most models are quantised to

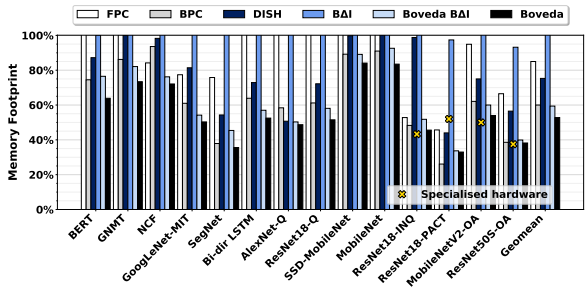


Figure 4. Boveda vs General-Purpose Compression Methods.

8b. Several models use more aggressive quantisation and were originally developed in conjunction with specialised architectures. We demonstrate that *Boveda* delivers the highest memory benefits possible without requiring method-specific hardware.

5.1 General-Purpose Compression Methods

Figure 4 reports the memory footprint in bits for the whole network and with different compression algorithms relative to the baseline. First, we compare *Boveda* to Frequent Pattern Compression (Alameldeen & Wood, 2004a) (FPC) and Bit-Plane Compression (Kim et al., 2016) (BPC) which are cache compression schemes for general-purpose systems that exploit value content. Appendix H provides more information on these and the rest of the methods considered here. FPC, on average it reduces its footprint by 15% due to removing zero values. Ignoring alignment and padding, BPC achieves 40%, mostly from removing zero 32b rows.

Also, we compare to Dictionary Sharing (Panda & Seznec, 2016) (DISH) and Base-Delta-Immediate (Pekhimenko et al., 2012) (BΔI), which target value content. On average, DISH reduces its footprint to 75%. BΔI exploits the low-dynamic range of values in programs via delta encoding. At best, it reduces its footprint by 7% ResNet50S-OA where it takes advantage of zero values. We evaluate a variant of *Boveda*, *Boveda*-BΔI, which incorporates elements from BΔI: It applies the per value compression method of BΔI but at a smaller granularity: all bits are zero and delta sizes of (8b, 4b, and 2b). We pack values in *hileras* so that decompression can be processed in parallel and without requiring a large crossbar at the output. The base is set to be 1 byte, while we reduce the working set of values to *BBlocks* of 8. *Boveda*-BΔI achieves 40% compression on average ignoring the overheads of width and pointer metadata. This is close to what *Boveda* achieves. However, decompressing values with *Boveda*-BΔI is considerably more complex and requires more energy than *Boveda*. For example, decompressing a block needs 8 additions in parallel, plus broadcasting the base across all of them. Compression is also more involved: it performs all compression possibilities in parallel before choosing the best. *Boveda* both achieves a better compression rate and is simpler to implement.

On average, *Boveda* reduces footprint to 53% while requiring lower hardware complexity. SSD-MobileNet and MobileNet benefit the least at 16%. Figure 4 highlights models with specialised quantisation showing the ideal memory footprint, where the memory hierarchy was designed specifically for them. Not only *Boveda* reduces their footprints to within 4% of that, for ResNet18-PACT, but it also reduces it much more than what would have been possible on 4b hardware. This is because *Boveda* takes advantage of actual value content. reports the memory footprint for the whole network. *Boveda* footprint includes a) the encoded values, b) the per *BBlock* width metadata, c) padding due to memory alignment, d) pointers sized for the dense accelerator. Total overhead accounts for less than 10% of the total compressed footprint on average.

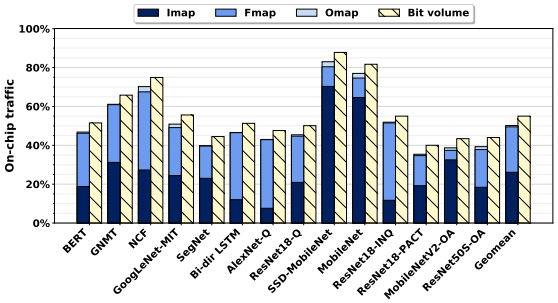
5.2 Tensorcores+

This section studies *Boveda* in the context of the Tensorcore-like accelerator with 64 Tensorcores organised in 8×8 rows. Each processing engine performs 64 MACS in parallel producing a single value. Each TC has 128-entry imap, fmap, and omap buffers. *Boveda* uses a *BBlock* size of 16. A 16-bank global buffer supplies the processing engines.

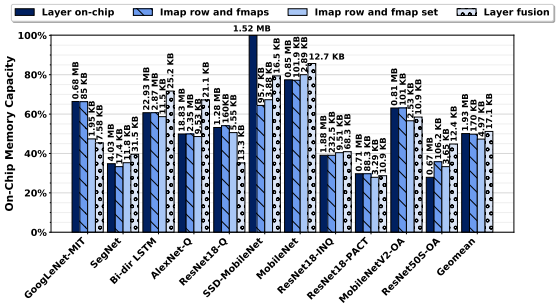
On-Chip Traffic: Figure 5a reports the reduction in on-chip memory traffic. since with *Boveda* an access may also read metadata we report two measurements: a) accesses and b) bits transferred, both are normalised to the baseline. *Boveda* performs 48% fewer transfers on average and transfers 55% fewer bits. The metadata traffic is small.

Reduction in On-Chip Capacity Needs: A significant design choice when architecting accelerators is the amount of on-chip storage. We study four on-chip sizing policies. Being able to fit: a) the imap, omap and the fmaps for the largest layer (Chen et al., 2014), b) the fmaps and a full row of windows from the imap (Siu et al., 2018), c) a full row of windows from the imap and an fmap per processing engine (Chen et al., 2014; Siu et al., 2018). Under “a” only the input and the final output go off-chip. Policy ”b“ guarantees per layer that each value is accessed *once* from off-chip. Policy ”c“ guarantees a single access per layer only for the imap and the omap. We also consider d) layer fusion which processes subsets of several layers without going off-chip for the intermediate i/omap values (Alwani et al., 2016).

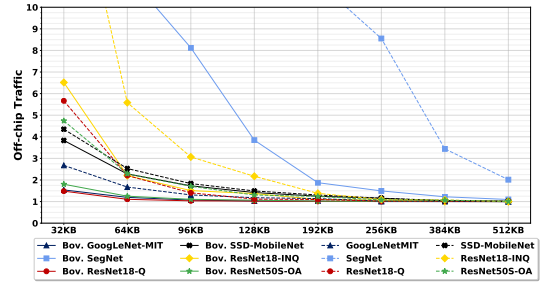
Figure 5b reports the on-chip memory capacity needed under each sizing policy, each normalised to a baseline under the *same* policy (it is different per policy). Overall, the reduction in storage needs follows the compression rates closely. In one case, SSD-MobileNet with policy ‘a’ (full layer on-chip), no reduction is possible. There is a single layer for which *Boveda* does not reduce overall on-chip data



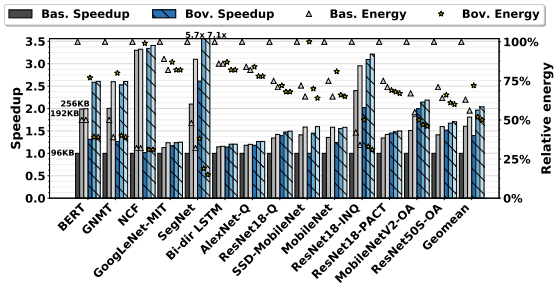
(a) Relative on-chip traffic and bit volume



(b) On-chip capacity needs with four sizing policies.



(c) Off chip traffic with and without *Boveda*



(d) Speedup and relative energy.

Figure 5. Boveda with Tensorcore+.

volume. *Boveda* improves energy and performance regardless since it reduces overall model traffic and footprint.

Reduction in Off-Chip Traffic: Figure 5c reports off-chip traffic per model with *Boveda* (solid lines) and without (dotted lines). For clarity, only a subset of the networks is shown. Traffic is normalised to that possible when every value is accessed once per layer. As the on-chip memory size increases, traffic approaches this minimum. *Boveda* allows us to use smaller on-chip memories. Moreover, for

Table 1. Hardware Costs.

		Area (um^2)		Power (mW)	
	BBlock	**8b**	**16b**	**8b**	**16b**
Width detector	4	109.8	247.68	0.042	0.498
	8	199.08	447.48	0.054	0.652
Compressor	-	288.36	710.64	0.238	0.470
Decompressor	-	345.6	810	0.205	0.419

Table 2. FPGA Prototype: Resource utilization.

Resource	**Count**	**Available**	**%Utilization**
LUT	42204	537,600	7.9
LUTRAM	88	76,800	0.1
FF	46744	1,075,200	4.3
BRAM	595	1,728	34.4
DSP	256	768	33.3

Per Boveda Resource Utilisation

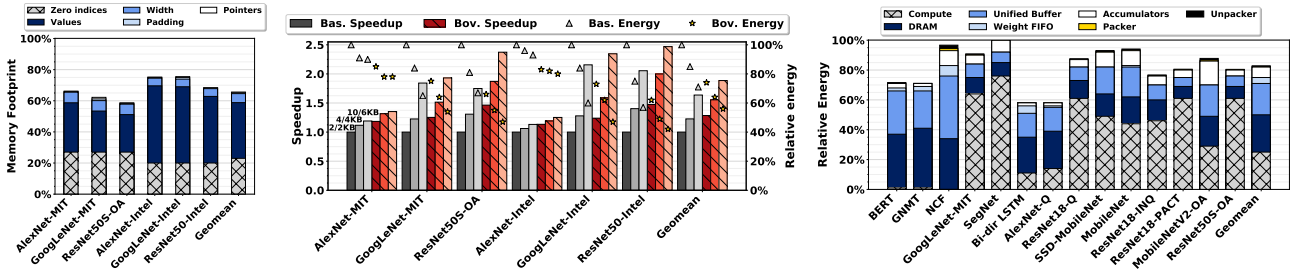
Component	**LUT**	**FF**	**BRAM**
Compressor	245 (0.05%)	120 (0.01%)	0 (0.00%)
Decompressor	378 (0.07%)	273 (0.03%)	0 (0.00%)
Total	623 (0.12%)	393 (0.04%)	0 (0.00%)

a given memory capacity *Boveda* reduces off-chip traffic. For example, in the case of SegNet, even 512KB of on-chip storage is not enough to achieve minimal traffic without *Boveda*. With 32KB of on-chip storage, *Boveda* reduces off-chip traffic by 3.8x for ResNet18 (traffic with *Boveda* is 1.48x and 5.66x without vs. reading values once) and by 2.6x for ResNet50S OA.

Execution Time: We measure performance for three configurations with on-chip global buffers of 96KB, 192KB, and 256KB. All use DDR4-3200 dual-channel off-chip memory. Figure 5d reports speedups normalised to the baseline with the 96KB global buffer. *Boveda* improves performance by 1.4x, 1.23x and 1.13x on average respectively. Improvements are the highest for SegNet whose convolutional layers are rather large and where *Boveda* compresses data considerably. Benefits with *Boveda* are also pronounced for MobileNetV2-OA, MobileNet, and ResNet18-INQ, where *Boveda* manages to avoid spilling off-chip for several layers. Since our on-chip hierarchy is designed to sustain peak execution bandwidth for the baseline, performance benefits with *Boveda* come from additional on-chip reuse leading to reduced off-chip traffic.

Energy: Figure 5d shows relative energy for the same memory configurations. *Boveda* saves 23%, 14% and 8% of the energy on average for the 96KB, 192KB and 256KB configurations. These benefits are due to less off- and on-chip traffic. As the on-chip capacity increases, off-chip accesses and their overall energy cost decrease.

Layout Measurements: Table 1 reports area and power for the (de)compression modules. The width detector is shared per *BBlock*. Total area overhead is 3.8%, 2.4%, and 1.2% for the 96KB, 192KB, and 256KB on-chip configurations. However, if we use this area for extra memory for the



(a) SCNN: Model footprint (b) SCNN: Speedup and Energy with and without *Boveda* (c) TPU: Energy breakdown with *Boveda*

Figure 6. a), b) SCNN: Compression, execution time speedup, and relative energy. c) TPU: relative energy.

baseline, *Boveda* is still 1.40x, 1.21x, and 1.11x faster on average and is slightly more energy efficient since on-chip accesses for the baseline become slightly more expensive.

5.3 SCNN and TPU

SCNN: SCNN used zero compression on- and off-chip. For the 16b networks used in the original study, SCNN used 4b zero skip indexes. We use 3b indexes instead for the 8b networks with a negligible effect on the number of eliminated zeros. *Boveda* does not compress the zero skip indexes. Figure 6a reports the reduction in total model footprint with *Boveda* over SCNN's zero compression which is 34% on average. The original SCNN sized its on-chip memory to fit all imaps on-chip for AlexNet, and GoogLeNet. This configuration results in larger networks such as ResNet50 spilling data off-chip. Furthermore, accumulator count limits the number of omap values and the number of concurrent filters. By amplifying on-chip storage capacity, *Boveda* reduces spill. We study this effect over three different per PE imap/accumulator configurations: 10KB/6KB as in the original paper; 4KB/4KB; and 2KB/2KB. The off-chip memory uses two channels of DDR4-3200. The area overhead for these configurations is 3.1%, 2.3% and, 1.8%, respectively, for SCNN 16b. Overheads are smaller for SCNN 8b.

Figure 6b reports speedup over the 2KB/2KB configuration with and without *Boveda*. On average, *Boveda* improves performance by 29%. Speedups are pronounced for the more recent ResNet50 models, which have comparatively larger imaps. With the original 10KB/6KB *Boveda* improves performance by 15%. Figure 6b shows that *Boveda* reduces energy by 26%, 24%, and 20% on average respectively for the three configurations. *Boveda* always reduces energy. Compute-bound models, e.g., GoogLeNet or ResNet50, benefit more since on-chip traffic accounts for a higher fraction of overall energy.

TPU: Figure 6c shows the memory energy breakdown with and without *Boveda* for *BBlocks* of 16 and the TPU (Jouppi, 2016). Appendix I studies different *BBlock* sizes. *Boveda*'s area overhead is less than 0.1%. On average *Boveda* reduces TPU energy by 17%.

5.4 Sensitivity Study: Extreme Quantisation

Appendix J studies *Boveda* under extremely narrow datawidths. Such extremely short datawidths are presently possible only on a handful of cases and typically at the expense of accuracy (Cai et al., 2017; Colangelo et al., 2018; Courbariaux & Bengio, 2016).

5.5 Boveda on an FPGA

We integrate *Boveda* into an FPGA design for accelerating fully connected layers. The design resembles that of Figure 1 with an 16x16 8b MAC array. Each column works on calculating an activation along a different channel of the omap of the fully connected layer. All columns share an on-chip imap buffer, the same activations are broadcasted to all columns. Each column has its own on-chip fmap buffer. The outputs are written to an on-chip omap buffer once all the columns have completed. We modify this design by adding *Boveda* compression and decompression units (16x8b BBlock) to the imap, fmap, and omap buffers. We implement the design on an Alpha Data 8V3 FPGA card (Data, 2017), which uses a Xilinx Virtex UltraScale VU095-2 FPGA (Xilinx, 2016). The design operates at 259 MHz after placement and routing, with the critical path not being in any *Boveda* units. Table 2 shows the total resource utilization of the design with the *Boveda* additions, and breaks down the resources required by *Boveda* to augment a single on-chip buffer.

6 Conclusion

Boveda is simple to implement and effective on-chip compression method for neural networks that is plug-in compatible with many accelerators. We have demonstrated that it reduces on-chip traffic while boosting the effective on-chip capacity. As a result, it reduces the amount of on-chip storage needed to avoid excessive off-chip accesses.

Acknowledgements This work was supported by the NSERC COHESA Research Network, two NSERC Discovery Grants, NSERC Strategic and CRD Grants.

REFERENCES

Abali, B., Franke, H., Poff, D. E., Saccone, R. A., Schulz, C. O., Herger, L. M., and Smith, T. B. Memory expansion technology (mxt): Software support and performance. *IBM Journal of Research and Development*, 45(2):287–301, March 2001. ISSN 0018-8646. doi: 10.1147/rd.452.0287.

Alameldeen, A. and Wood, D. Frequent pattern compression: A significance-based compression scheme for l2 caches. Technical report, University of Wisconsin-Madison Department of Computer Sciences, 2004a.

Alameldeen, A. R. and Wood, D. A. Adaptive cache compression for high-performance processors. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, ISCA '04, pp. 212–, Washington, DC, USA, 2004b. IEEE Computer Society. ISBN 0-7695-2143-6. URL http://dl.acm.org/citation.cfm?id=998680.1006719.

Alwani, M., Chen, H., Ferdman, M., and Milder, P. Fused-layer cnn accelerators. In *49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.

Arelakis, A. and Stenstrom, P. Sc2: A statistical compression cache scheme. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pp. 145–156, June 2014. doi: 10.1109/ISCA.2014.6853231.

Arelakis, A., Dahlgren, F., and Stenstrom, P. Hycomp: A hybrid cache compression method for selection of data-type-specific compression methods. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, pp. 38–49, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-4034-2. doi: 10.1145/2830772.2830823. URL http://doi.acm.org/10.1145/2830772.2830823.

Cai, Z., He, X., Sun, J., and Vasconcelos, N. Deep learning with low precision by half-wave gaussian quantization, 2017.

Chen, Y., Luo, T., Liu, S., Zhang, S., He, L., Wang, J., Li, L., Chen, T., Xu, Z., Sun, N., and Temam, O. Dadiannao: A machine-learning supercomputer. In *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*, pp. 609–622, Dec 2014. doi: 10.1109/MICRO.2014.58.

Chen, Y., Emer, J., and Sze, V. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pp. 367–379, 2016.

Choi, J., Wang, Z., Venkataramani, S., Chuang, P. I., Srinivasan, V., and Gopalakrishnan, K. PACT: parameterized clipping activation for quantized neural networks. *CoRR*, abs/1805.06085, 2018. URL http://arxiv.org/abs/1805.06085.

Colangelo, P., Nasiri, N., Nurvitadhi, E., Mishra, A., Margala, M., and Nealis, K. Exploration of low numeric precision deep learning inference using intel® fpgas. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 73–80, 2018.

Courbariaux, M. and Bengio, Y. Binarynet: Training deep neural networks with weights and activations constrained to +1 or -1. *CoRR*, abs/1602.02830, 2016. URL http://arxiv.org/abs/1602.02830.

Dally, B. Power, programmability, and granularity: The challenges of exascale computing. In *2011 IEEE International Parallel Distributed Processing Symposium*, pp. 878–878, May 2011. doi: 10.1109/IPDPS.2011.420.

Data, A. ADM-PCIE-8V3. "https://www.alpha-data.com/pdfs/adm-pcie-8v3.pdf", 2017.

Durant, L., Giroux, O., Harris, M., and Stam, N. Nvidia developer blog, May 2017. URL https://devblogs.nvidia.com/inside-volta/.

Ekman, M. and Stenstrom, P. A robust main-memory compression scheme. In *Proceedings of the 32Nd Annual International Symposium on Computer Architecture*, ISCA '05, pp. 74–85, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2270-X. doi: 10.1109/ISCA.2005.6. URL https://doi.org/10.1109/ISCA.2005.6.

Gao, M., Yang, X., Pu, J., Horowitz, M., and Kozyrakis, C. Tangram: Optimized coarse-grained dataflow for scalable nn accelerators. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pp. 807–820, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450362405. doi: 10.1145/3297858.3304014. URL https://doi.org/10.1145/3297858.3304014.

Hallnor, E. G. and Reinhardt, S. K. A unified compressed memory hierarchy. In *11th International Symposium on High-Performance Computer Architecture*, pp. 201–212, Feb 2005. doi: 10.1109/HPCA.2005.4.

Han, S., Mao, H., and Dally, W. J. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. *arXiv:1510.00149 [cs]*, October 2015. URL http://arxiv.org/abs/1510.00149. arXiv: 1510.00149.

Han, S., Liu, X., Mao, H., Pu, J., Pedram, A., Horowitz, M. A., and Dally, W. J. EIE: efficient inference engine on compressed deep neural network. In *43rd ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2016, Seoul, South Korea, June 18-22, 2016*, pp. 243–254, 2016.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015. URL http://arxiv.org/abs/1512.03385.

Hong, S., Nair, P. J., Abali, B., Buyuktosunoglu, A., Kim, K., and Healy, M. Attaché: Towards ideal memory compression by mitigating metadata bandwidth overheads. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 326–338, 2018.

Hong, S., Abali, B., Buyuktosunoglu, A., Healy, M. B., and Nair, P. J. Touché: Towards ideal and efficient cache compression by mitigating tag area overheads. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, pp. 453–465, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450369381. doi: 10.1145/3352460.3358281. URL https://doi.org/10.1145/3352460.3358281.

Horowitz, M. Computing's energy problem ((and what we can do about it). In *2014 IEEE International Solid-State Circuits Conference - Digest of Technical Papers, ISSCC 2014*. IEEE, 2014a.

Horowitz, M. 1.1 computing's energy problem (and what we can do about it). volume 57, pp. 10–14, 02 2014b. ISBN 978-1-4799-0920-9. doi: 10.1109/ISSCC.2014.6757323.

Jain, A., Phanishayee, A., Mars, J., Tang, L., and Pekhimenko, G. Gist: Efficient data encoding for deep neural network training. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ISCA '18, pp. 776–789, Piscataway, NJ, USA, 2018. IEEE Press. ISBN 978-1-5386-5984-7. doi: 10.1109/ISCA.2018.00070. URL https://doi.org/10.1109/ISCA.2018.00070.

Jia, Z., Maggioni, M., Staiger, B., and Scarpazza, D. P. Dissecting the NVIDIA volta GPU architecture via microbenchmarking. *CoRR*, abs/1804.06826, 2018. URL http://arxiv.org/abs/1804.06826.

Jouppi, N. Google supercharges machine learning tasks with TPU custom chip. https://cloudplatform.googleblog.com/2016/05/Google-supercharges-machine-learning-tasks-with-custom-chip.html, 2016. [Online; accessed 3-Nov-2016].

Jouppi, N. P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., Bates, S., Bhatia, S., Boden, N., Borchers, A., Boyle, R., Cantin, P.-l., Chao, C., Clark, C., Coriell, J., Daley, M., Dau, M., Dean, J., Gelb, B., Ghaemmaghami, T. V., Gottipati, R., Gulland, W., Hagmann, R., Ho, C. R., Hogberg, D., Hu, J., Hundt, R., Hurt, D., Ibarz, J., Jaffey, A., Jaworski, A., Kaplan, A., Khaitan, H., Killebrew, D., Koch, A., Kumar, N., Lacy, S., Laudon, J., Law, J., Le, D., Leary, C., Liu, Z., Lucke, K., Lundin, A., MacKean, G., Maggiore, A., Mahony, M., Miller, K., Nagarajan, R., Narayanaswami, R., Ni, R., Nix, K., Norrie, T., Omernick, M., Penukonda, N., Phelps, A., Ross, J., Ross, M., Salek, A., Samadiani, E., Severn, C., Sizikov, G., Snelham, M., Souter, J., Steinberg, D., Swing, A., Tan, M., Thorson, G., Tian, B., Toma, H., Tuttle, E., Vasudevan, V., Walter, R., Wang, W., Wilcox, E., and Yoon, D. H. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pp. 1–12, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4892-8. doi: 10.1145/3079856.3080246. URL http://doi.acm.org/10.1145/3079856.3080246.

Judd, P., Albericio, J., Hetherington, T., Aamodt, T. M., Jerger, N. E., and Moshovos, A. Proteus: Exploiting numerical precision variability in deep neural networks. In *Proceedings of the 2016 International Conference on Supercomputing*, ICS '16, pp. 23:1–23:12, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4361-9. doi: 10.1145/2925426.2926294. URL http://doi.acm.org/10.1145/2925426.2926294.

Kim, J., Sullivan, M., Choukse, E., and Erez, M. Bit-plane compression: Transforming data for better compression in many-core architectures. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pp. 329–340. IEEE, 2016.

Lascorz, A. D., Sharify, S., Edo, I., Stuart, D. M., Awad, O. M., Judd, P., Mahmoud, M., Nikolic, M., Siu, K., Poulos, Z., and Moshovos, A. Shapeshifter: Enabling fine-grain data width adaptation in deep learning. In *Proceedings of the 52Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, pp. 28–41, New York, NY, USA, 2019. ACM. ISBN 978-1-4503-6938-1. doi: 10.1145/3352460.3358295. URL http://doi.acm.org/10.1145/3352460.3358295.

Mahmoud, M., Siu, K., and Moshovos, A. Diffy: A dÉjÀ vu-free differential deep neural network accelerator. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-51, pp. 134–147, Piscataway, NJ, USA, 2018. IEEE Press. ISBN 978-1-5386-6240-3. doi: 10.1109/MICRO.2018.00020. URL https://doi.org/10.1109/MICRO.2018.00020.

Muralimanohar, N., Balasubramonian, R., and Jouppi, N. Cacti 6.0: A tool to model large caches. *HP Laboratories*, 01 2009.

NVIDIA. NVIDIA AMPERE GA102 GPU architecture. *White Paper*, 2020. URL https://www.nvidia.com/en-us/geforce/news/rtx-30-series-ampere-architecture-whitepaper-download/.

Panda, B. and Seznec, A. Dictionary sharing: An efficient cache compression scheme for compressed caches. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–12, 2016.

Parashar, A., Rhu, M., Mukkara, A., Puglielli, A., Venkatesan, R., Khailany, B., Emer, J., Keckler, S. W., and Dally, W. J. Scnn: An accelerator for compressed-sparse convolutional neural networks. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pp. 27–40, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4892-8. doi: 10.1145/3079856.3080254. URL http://doi.acm.org/10.1145/3079856.3080254.

Park, E., Kim, D., and Yoo, S. Energy-efficient neural network accelerator based on outlier-aware low-precision computation. In *ISCA*, pp. 688–698. IEEE Computer Society, 2018.

Park, J., Li, S., Wen, W., Tang, P. T. P., Li, H., Chen, Y., and Dubey, P. Faster CNNs with Direct Sparse Convolutions and Guided Pruning. In *5th International Conference on Learning Representations (ICLR)*, 2017.

Pekhimenko, G., Seshadri, V., Mutlu, O., Gibbons, P. B., Kozuch, M. A., and Mowry, T. C. Base-delta-immediate compression: Practical data compression for on-chip caches. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, pp. 377–388, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1182-3. doi: 10.1145/2370816.2370870.

Pekhimenko, G., Seshadri, V., Kim, Y., Xin, H., Mutlu, O., Gibbons, P. B., Kozuch, M. A., and Mowry, T. C. Linearly compressed pages: A low-complexity, low-latency main memory compression framework. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, pp. 172–184, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2638-4. doi: 10.1145/2540708.2540724. URL http://doi.acm.org/10.1145/2540708.2540724.

Qureshi, M. K., Thompson, D., and Patt, Y. N. The v-way cache: demand-based associativity via global replacement. In *32nd International Symposium on Computer Architecture (ISCA'05)*, pp. 544–555, June 2005. doi: 10.1109/ISCA.2005.52.

Reddi, V. J., Cheng, C., Kanter, D., Mattson, P., Schmuelling, G., Wu, C.-J., Anderson, B., Breughe, M., Charlebois, M., Chou, W., Chukka, R., Coleman, C., Davis, S., Deng, P., Diamos, G., Duke, J., Fick, D., Gardner, J. S., Hubara, I., Idgunji, S., Jablin, T. B., Jiao, J., John, T. S., Kanwar, P., Lee, D., Liao, J., Lokhmotov, A., Massa, F., Meng, P., Micikevicius, P., Osborne, C., Pekhimenko, G., Rajan, A. T. R., Sequeira, D., Sirasao, A., Sun, F., Tang, H., Thomson, M., Wei, F., Wu, E., Xu, L., Yamada, K., Yu, B., Yuan, G., Zhong, A., Zhang, P., and Zhou, Y. Mlperf inference benchmark, 2019.

Rhu, M., O'Connor, M., Chatterjee, N., Pool, J., Kwon, Y., and Keckler, S. W. Compressing DMA engine: Leveraging activation sparsity for training deep neural networks. In *IEEE International Symposium on High Performance Computer Architecture, HPCA 2018, Vienna, Austria, February 24-28, 2018*, pp. 78–91, 2018. doi: 10.1109/HPCA.2018.00017. URL https://doi.org/10.1109/HPCA.2018.00017.

Rosenfeld, P., Cooper-Balis, E., and Jacob, B. Dramsim2: A cycle accurate memory system simulator. *IEEE Computer Architecture Letters*, 10(1):16–19, Jan 2011. ISSN 2473-2575. doi: 10.1109/L-CA.2011.4.

Siu, K., Stuart, D. M., Mahmoud, M., and Moshovos, A. Memory requirements for convolutional neural network hardware accelerators. In *IEEE International Symposium on Workload Characterization*, 2018.

Xilinx. UltraScale FPGA Product Selection Guide. "https://www.xilinx.com/support/documentation/selection-guides/ultrascale-plus-fpga-product-selection-guide.pdf", 2016.

Yang, Tien-Ju and Chen, Yu-Hsin and Sze, Vivienne. Designing Energy-Efficient Convolutional Neural Networks using Energy-Aware Pruning. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017.

Young, V., Kariyappa, S., and Qureshi, M. K. Enabling transparent memory-compression for commodity memory systems. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 570–581, 2019.

Zhou, A., Yao, A., Guo, Y., Xu, L., and Chen, Y. Incremental network quantization: Towards lossless cnns with low-precision weights. *CoRR*, abs/1702.03044, 2017. URL http://arxiv.org/abs/1702.03044.

Zmora, N., Jacob, G., Zlotnik, L., Elharar, B., and Novik, G. Neural network distiller, June 2018. URL https://doi.org/10.5281/zenodo.1297430.

A Appendix Overview

The following sections complement the material presented in the main paper. The following additional material is included:

- Appendix B presents experimental evidence that shows that the value distribution in neural networks is such that most values tend to cluster near zero or some other value near zero. This observation motivated the simple and effective compression scheme adopted by *Boveda*.
- Appendix C lists the neural networks we used in our study.
- Appendix D complements Section 3.1 by presenting an example of how *Boveda* decompresses values.
- Appendix E reviews the tile architecture of SCNN and complements the description of Section 3.
- Appendix F reviews convolutional layers' operation and details how *Boveda* can seamlessly process activation windows even though the starting positions in memory are value-dependent.
- Appendix G reviews the TPU architecture and explains how *Boveda* can enhance it.
- Appendix H reviews previously proposed compression methods, including those Section 5.1 compares against.
- Appendix I studies memory footprint and energy reduction for different *BBlock* sizes on top of TPU.
- Finally, Appendix J studies *Boveda* under *hypothetical* extreme quantisation schemes. Extreme quantisation (e.g., using 6b or narrows operands) is not generally possible today.

B Typical Value Distributions in Neural Networks

This appendix reports measurements of the value content of neural networks. The data support our observation that choosing a data container whose bitwidth is sufficient to store the values with the highest magnitude is excessive for the bulk of the rest of the values. This is because imaps and fmaps tend to exhibit a value distribution that is heavily biased towards zero or a value near zero.

Conventional memory hierarchies do not capitalize on this property as they store all imap or fmap elements using a datawidth which is sufficiently long to accommodate *any* value possible. This is excessive for most values and across all networks studied. This behaviour is exhibited by all models studied. This section highlights two such cases: ResNet18 (image classification) (He et al., 2015), and SSD_MobileNet (object-detection), both quantised to 8b (Reddi et al., 2019).

Figure 7 shows the regular and cumulative distributions of imap and fmap values for representative convolutional and fully-connected layers. Figures 7a and 7c show imap values over a batch of 64 randomly selected inputs, whereas Figures 7b and 7d shows fmaps which are input independent.

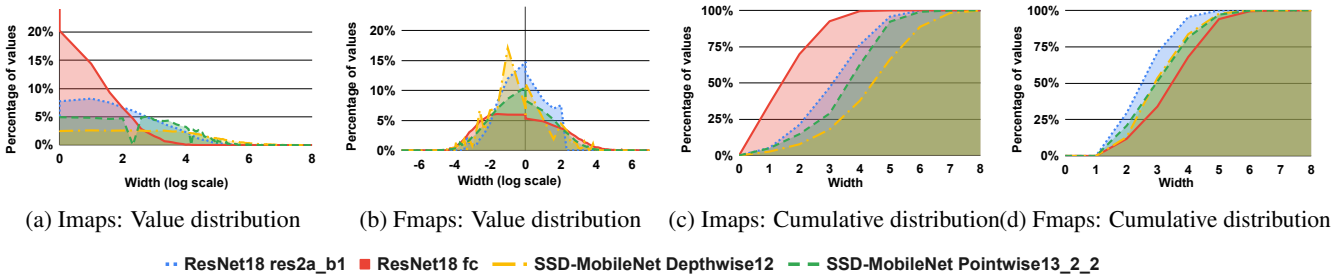
Figures 7a and 7c show that in ResNet18's `res2a_branch1`, most imap values fit within 5b, which under ideal conditions translates into a 37.5% reduction in footprint over the original 8b. Just 4b, a 50% reduction over 8b, are sufficient for virtually all imap values in its fully-connected layer `fc`. SSD_Mobilenet exhibits similar behaviour. In its 2D convolution layer `depthwise12` 90% of the values need 6b or fewer, which is also enough to represent virtually all imap values in its object detection SSD module layer `pointwise13_2_2`. Figures 7b and 7d report similar trends for the fmaps. ResNet18's `res2_branch1` only 5b are sufficient for most of the fmap values whereas 6b are sufficient for virtually all values in its `fc` layer. However, in `fc` 95% of the fmap values need at most 5b. SSD-MobileNet's fmaps are similar. Virtually all values fit in 6b, 90% fit in 5b and more than 80% in 4b.

C Neural Networks Studied

Table 3 reports the models we use in this study. Several models use more aggressive quantisation and were originally developed in conjunction with specialised architectures. We demonstrate that *Boveda* delivers the highest memory benefits possible without requiring method-specific hardware. These models include: Intel's INQ (Zhou et al., 2017), whose fmap values are limited to sixteen signed powers of two or zero. Representing weights as magnitudes requires 16b whereas 5b are enough with specialised hardware. PACT(Choi et al., 2018) requires a modified ReLU with a configurable saturation threshold and uses 4b imaps and fmaps for all but the first and last layers that use 8b. Outlier-Aware quantisation aggressively reduces the number of bits for most values (e.g., 4b), except for a few large values (outliers of 8b) that are handled separately (Park et al., 2018). Since SCNN excels for pruned models, we include such models from Intel's Skim Caffe (Park et al., 2017) repository and MIT's Eyeriss group (Yang, Tien-Ju and Chen, Yu-Hsin and Sze, Vivienne, 2017). We highlight MobileNet and SSD-MobileNet from MLPerf, and GNMT and NCF post-trained quantised by Intel from MLPerf implementation (Reddi et al., 2019; Zmora et al., 2018). We study symmetric vs. asymmetric quantisation difference in compression for GNMT, which was around 8% better for symmetric. However, we use asymmetric due to archiving a higher BLEU score.

D Boveda Decompression Example

This appendix complements Section 3.1 and details the decompression for I_0 and I_4, assuming the *Boveda* layout example of Figure 2c. For convenience Figure 8a shows the

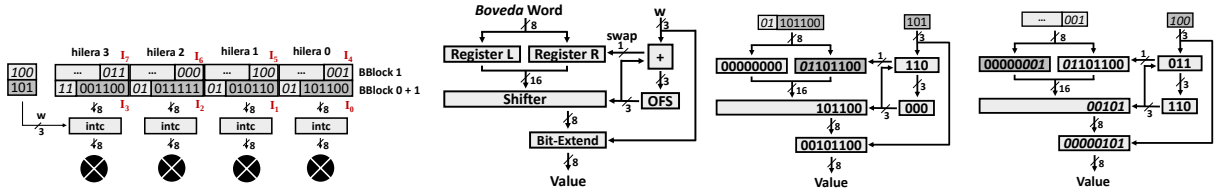


(a) Imaps: Value distribution (b) Fmaps: Value distribution (c) Imaps: Cumulative distribution (d) Fmaps: Cumulative distribution

Figure 7. Value and cumulative distribution for some layers of two 8b models: ResNet18 and SSD-MobileNet

Table 3. Neural network models studied.

	Dataset	Application	Data Type	Pruned	Specialised Q	imap (MB)	fmap (MB)
BERT	MRP	NLP	int8			10.12	81.56
GNMT	WMT 2016	NMT	int8			0.37	121.55
NCF	ml-20m	Recommendation	int8			0.75KB	0.10
SegNet	CamVid	Segmentation	int8			29.39	1.35
Bi-dir LSTM	Flickr8k	Captioning	int8			0.59	77.26
AlexNet-Q	ImageNet	Classification	int8			0.41	58.32
ResNet18-Q	ImageNet	Classification	int8			2.32	11.14
SSD-MobileNe	COCO	Object Detection	int8			10.14	6.48
MobileNet	ImageNet	Classification	int8			5.27	4.02
ResNet18-INQ	ImageNet	Classification	int4(log)/16		✓	4.64	**6.96**-22.28
ResNet18-PACT	ImageNet	Classification	int4/int8		✓	**1.24**-2.32	**5.82**-11.14
MobileNetV2-OA	ImageNet	Classification	int4/int8		✓	**4.57**-9.14	**1.65**-3.31
ResNet50S-OA	ImageNet	Classification	int3/int8	✓	✓	**3.71**-9.91	**9.12**-24.32
AlexNet-MIT	ImageNet	Classification	int8	✓		0.41	58.14
GoogLeNet-MIT	ImageNet	Classification	int8	✓		4.65	6.67
AlexNet-Intel	ImageNet	Classification	int16	✓		0.95	116.27
GoogLeNet-Intel	ImageNet	Classification	int16	✓		9.19	25.50
ResNet50-Intel	ImageNet	Classification	int16	✓		19.81	48.64



(a) *Boveda*'s Layout (b) Decompression Module (c) Decompressing I_0 (d) Decompressing I_4

Figure 8. Boveda Decompression Example

input buffer contents as laid out by *Boveda* (copy of Figure 2c), Figure 8b shows the decompression module, and Figures 8c and 8d show how I_0 and I_4 are decompressed.

In cycle 1, the imap buffer supplies the first set of 8b `0110 1100` written into register L. Concurrently, W is loaded from the width memory with the datawidth `101` for *BBlock* 0. OFS is updated to $OFS{=}(OFS{+}W{+}1)$ MOD 8=0. Since $OFS{+}W{+}1$ exceeded 8 (carry out from the adder), R contains no useful bits and thus the positions of L are R are swapped at the end of cycle 1 and a read from the imap buffer is triggered for the next cycle.

In cycle 2 (Figure 8c) the module reads in the next 8b copying them into L at the end of the cycle. Now L and R contain two consecutive rows of compressed values from the same *hilera*. We are now in steady-state. During cycle 2 and since OFS is 0, the 16b output of (L,R) is shifted by 0 thus aligning the LSb of the compressed I_0 with the LSb of the output. The bit-extension block upon the guidance of W passes through the lower 6 bits and fills in the upper 2 bits accordingly. Here it zero extends the value to 8b since this imap is known to have only positive values. If the layer had signed imap values, the extender would sign-extend instead. As a result, the value `0010 1100`, the original I_0 is sent to the multipliers. OFS is updated as before: $OFS = (0{+}5{+}1)$ MOD $8 = 6$. Since this does not exceed 8 we will not read a value from the imap buffer in the next cycle. A new width field is read into W by the end of the cycle. This is the width for *BBlock* 1.

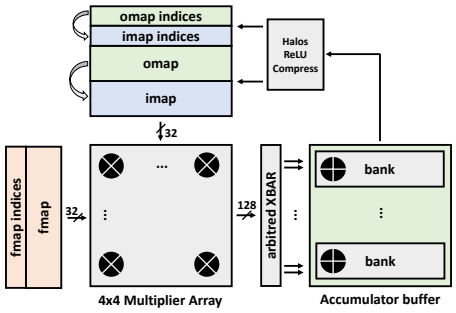


Figure 9. SCNN processing engine organization.

In cycle 3, as Figure 8d shows, *OFS* instructs the shifter to slide the (L,R) by 6 positions. The extender block passes through the 5 least significant bits since W is `100` zero-extending to 8b. (OFS) will be updated to $(6{+}4{+}1)$ MOD 8, and since this exceeds 8, L and R will be swapped, and the next imap row will be loaded into L in the next cycle.

E SCNN

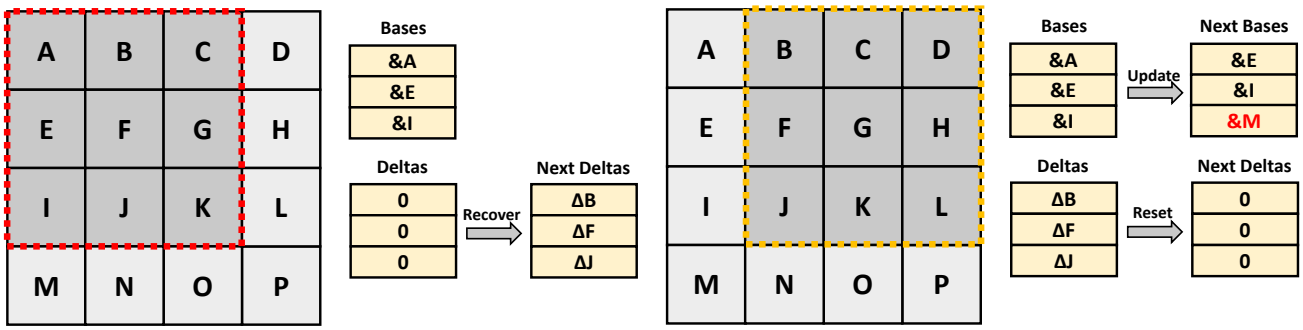
SCNN excels at processing the convolutional layers of pruned neural models. The inputs to a convolutional layer are K fmaps of dimension $S{\times}R{\times}C$ (height, width, channel), an $H{\times}W{\times}C$ imap where typically $H \gg S$ and $W \gg R$ and a stride s. The fmaps are statically known values (*weights*), whereas the imaps are runtime calculated values (*activations*). The output is an $\left(\lceil\frac{H-S}{s}\rceil+1\right){\times}\left(\lceil\frac{W-R}{s}\rceil+1\right){\times}K$ omap (*activations*). Each omap value is calculated as the 3D convolution of an fmap with an equally sized *window* of the imap. Each fmap produces the omap values for one channel by sliding the window over the imap using the stride s along the H and W dimensions. The 3D convolution entails the pair-wise multiplication of an fmap element with its corresponding imap element, followed by the accumulation of all these products into the omap value. Each 3D convolution is equivalently the sum of C 2D convolutions on each input channel.

Figure 9 shows the organisation of an SCNN tile and complements Section 3.1. The tile has three buffers holding imaps (and omaps), fmaps, and partial sum omap accumulators. A spatial dataflow performs all 2D convolutions for all windows of a single channel of the imap at a time. SCNN builds on the observation that in convolutional layers the product of any fmap value with any imap value from the same channel contributes to some omap value. At maximum throughput, the tile processes 4 imap and 4 fmap values all from the same channel and calculates the products for all 16 possible (imap,fmap) pairs. It then directs via a crossbar all these products into their corresponding accumulator. The accumulator buffer is organized into 32 banks to reduce conflicts that occur when multiple products map onto accumulators in the same bank. To take advantage of sparsity, the imap and the fmap omit zero values storing non-zero values as $((value),(skip))$ pairs where $(skip)$ is the number of zero values omitted after each. By using these $(skip)$ fields SCNN deduces each value's original position and maps the products to their respective accumulators. *Boveda* compresses values after SCNN eliminates zeros using the Post-Processing Unit (PPU). Thus, zeros do not affect *hileras* organisation.

F Processing Convolutional Layers in TensorCores+

Convolutional Layers: N.Samples-Height-Width-Channel (NHWC) memory mapping is commonly used to increase data locality for convolutional layers. Compared to other layers, the added challenge for convolutional layers is the need to initiate accesses to multiple, often overlapping windows. Without loss of generality, let us consider a channel-first output stationary dataflow where each window is processed in channel-width-height order. We will use the term COLUMN to refer to all imap values with the same (width,height) coordinates. To compute a single omap our dataflow will access the values within a COLUMN sequentially and then access other COLUMNS in width-height order. *Boveda* can group values into *BBlocks* sequentially along each COLUMN, adhering to the NHWC mapping.

However, each COLUMN's starting position is no longer a linear function of its (width,height) coordinates. A naïve solution is to keep pointers to each COLUMN (2D coordinates of the first channel). This is excessive since each COLUMN is needed during the processing of a few windows (e.g., for a 3×3 fmap, each column will be accessed 9 times). Furthermore, a typical first layer with imap dimensions $230{\times}230$ imap would need around 52k pointers to process the whole layer. This overhead would make the compression impractical. Instead, we made two observations: a) windows typically overlap and thus the starting position of each COLUMN will be encountered while processing an earlier window due to well-known access pattern, and b) address mapping is known beforehand (NHWC), hence we know which addresses are consecutive. By these two observations, *Boveda* reduces the number of pointers explicitly stored as metadata while "recovering" the rest during processing and keeping them around only as long as necessary. Figure 10 shows an example 2D convolution with *Boveda* compression using reduced pointers for a 5x5 imap, 3x3 fmap, and a stride of 1. This example is equivalent to a 3D convolution due to channels that are stored first in NHWC. During the computation of the first window in Figure 10a, we "recover" the starting position of the second column. This is possible due to the dataflow's well-known access pattern and the linear accesses from A-C, E-G, and I-K. Also, since the distances between addresses are relatively small; the starting



(a) First window

(b) Second window

Figure 10. Example convolution with *Boveda* compression for a 5x5 imap, 3x3 fmap, and a stride of 1.

position of a column is stored as a delta to the first column. This "recovering" operation is repeated for all the windows in a row of windows but the last one. After computing the last window in a row, the next window to start is located in the next row. This is the case of Figure 10b. Hence, we retrieve the starting position of the window from the second observation. We know that the next address after L is M due to the address mapping we are using, and the deltas are rested to zero. This pointers reduction can be performed for other dataflows and address mappings. We generate the pointers on-the-fly for convolutional layers whose filter dimensions are larger than the stride. There are layers in ResNet having 1x1 kernels and stride of 2; for those, we keep all pointers. This is affordable since the layers are small.

The number of pointers stored along the imap depends on the imap and fmap dimensions and the number of windows: we use $max(H \times \lceil \frac{windows}{(H-S+1)} \rceil)$ where H, S and $windows$ respectively the imap rows, fmap rows, and the maximum number of windows we wish to process concurrently. On-chip we need two sets of registers. One for holding the current set of points and one to "recover" the next set. For example for a layer with an 230×230 imap and a 3×3 fmap, storing around 700 pointers is enough to support concurrent processing of more than 200 windows. Since each fmap is read once per window, *Boveda* also keeps a pointer per fmap. The overhead is small, and except for depthwise separable convolutions, even the smallest filters are of 3×3 width and height and several tens of channels deep. Finally, *Boveda* stores a base address and all other pointers as offsets rather than storing absolute pointers

G BOVEDA OVER THE TPU

This section details how *Boveda* can be incorporated into the first generation Tensor Processing Unit (TPU) (not enough is publicly known about later generations). The TPU is a large accelerator designed for datacenters. It uses a large systolic array of 256×256 MAC units and aims to hold all imap values on-chip to maximize systolic array utilization. To keep the systolic array utilised, the TPU's on-chip memory buffers can sustain two wide reads per cycle. For this purpose, it uses a Unified Buffer of 24MB for imaps, while another 4MB are used to accumulate omaps. Fmaps are streamed from off-chip DRAM with a weight stationary dataflow in a FIFO fashion. Four on-chip 64KiB tiles reduce memory synchronization latency for the fmaps.

A Systolic Data Setup module between the Unified Buffer and the Systolic array sets up the imap to match the TPU dataflow. The imap read order can be adjusted to best fit each layer. Fmaps are read in blocks of 64k and loaded over 256 cycles into the Systolic Array, which is double-buffered. Loading the weights into the compute matrix requires 256 reads of 256×8b weights. TPU's dataflow preloads a matrix of 64K weights, and feeds the imaps in a wave-fashion. Every cycle, imap values are moved to the next column of the systolic array so that each imap value is multiplied by all the weights in the row. Similarly, partial omaps are moved to the next row to be accumulated. As a result, every column contributes to the same omap. This dataflow needs 256 cycles for an imap to meet all fmap values and an extra 256 cycles for an omap to accumulate with all rows.

***Boveda* extension:** We insert a *Boveda* decompressor between the imap and fmap buffers and the systolic array, and compressors at the output. Since weights are read in chunks of 256×8b to feed the systolic array, they are kept compressed until they need to be loaded. Since *Boveda* arranges values in *BBlocks*, the per value overhead of a 256 wide read is the same as SCNN or the Tensorcore+. However, we use *BBlocks* of 16 to avoid broadcasting the same *BBlock* width to 256 decompressors. Imaps are decompressed before the Systolic Data Setup Module to support all dataflows. Imaps are kept compressed in processing other depending on the layer architecture. Compressors are located after the Activation Pipeline to compress the imap for the next layer before storing them in the Unified Buffer. As with Ten-

sorcore+ *Boveda* uses pointers and appropriate alignment where needed.

H General-Purpose Compression Methods Considered

Figure 4 reported the memory footprint in bits for the whole network and with different compression algorithms relative to the baseline. First, the figure compared *Boveda* to Frequent Pattern Compression (Alameldeen & Wood, 2004a) (FPC) and Bit-Plane Compression (Kim et al., 2016) (BPC) which are cache compression schemes for general-purpose systems. Both target value width in addition to other properties. Since programmers tend to use 32b variables regardless of actual value range needs, FPC detects whether values can be stored in power of two-sized containers (4b being the smallest). On average, it reduces footprint by 15%. This is mostly from removing zeros. BPC preprocess the 32x32b input matrix by calculating deltas, bit-transposing the matrix, and XOR-ing neighbours. This is done to reduce entropy of bits. Then, the zero 32b rows of the matrix are run-length encoded, while for the non-zeros, FPC-like patterns are applied. Ignoring alignment and padding, BPC achieves 40%, mostly from removing zero 32b rows.

Also, the figure compared to Dictionary Sharing (Panda & Seznec, 2016) (DISH) and Base-Delta-Immediate (Pekhimenko et al., 2012) (BΔI), which target value content. DISH applies 8-dictionary encoding to 64B input blocks. If different blocks have the same dictionary values, they can share the overhead. Moreover, when a block cannot be compressed within 8 entries, the same process is repeated, comparing only the top bits. If the range is still to large, it is not compressed. On average, it reduced its footprint to 75%.

BΔI exploits the low-dynamic range of values in programs (neighbouring values tend to be close in value). It operates on chunks of 64 bytes and reduces width at a byte granularity. It represents values as deltas of 4,2 or 1 bytes from either zero or the first value of 8, 4 or 2 bytes. All zero chunks are represented as 1 byte plus metadata. This byte granularity is too large for neural networks. At best, it reduces its footprint by 7% ResNet50S-OA where it takes advantage of zero values.

We evaluate a variant of *Boveda*, *Boveda*-BΔI, which incorporates elements from BΔI: It applies the per value compression method of BΔI but at a smaller granularity. The compression options are: all bits are zero and delta sizes of (8b, 4b, and 2b). As in *Boveda*, it packs values in *hileras* so that decompression can be processed in parallel and without requiring a large crossbar at the output. The base is set to be 1 byte, while we reduce the working set of values to *BBlocks* of 8. *Boveda*-BΔI achieves 40% compression on average

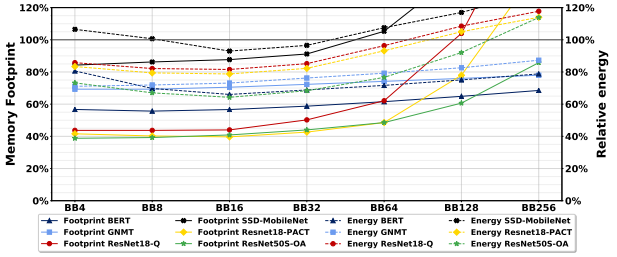


Figure 11. Memory Footprint and Energy reduction for different *BBlocks* sizes

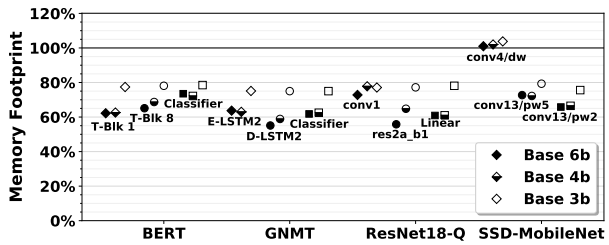


Figure 12. Effectiveness with Smaller Datatypes

ignoring the overheads of width and pointer metadata. This is close to what *Boveda* achieves. However, decompressing values with *Boveda*-BΔI is considerably more complex and requires more energy than *Boveda*. For example, decompressing a block needs 8 additions in parallel, plus broadcasting the base across all of them. Compression is also more involved: it performs all compression possibilities in parallel before choosing the best. *Boveda* both achieves a better compression rate and is simpler to implement.

I BBlock: Sensitivity Study

TPU uses a 256×256 systolic array that is fed through wide 256-value on-chip. *Boveda* however can be configured with a smaller *BBlock* size to better balance costs. The larger the *BBlock* size the lower the metadata overhead, yet the higher the probability that a higher magnitude value will be included reducing compression effectiveness. Moreover, the longer the distance the metadata has to be broadcast. The smaller the *BBlock* size the higher the compression rate and the shorter the distance the per *BBlock* metadata needs to be communicated. However, a smaller *BBlock* size increases metadata overhead and requires support for more independent accesses to the buffers. Given that the buffers will be banked, supporting the required independent accesses is unlikely to be a significant cost. Accordingly, this section studies how *BBlock* size impacts *Boveda* performance by varying the *BBlock* size from groups of 4 to 256 values.

Figure 11 reports memory footprint (solid lines) and energy reduction (dotted lines) per model for TPU on-chip memories with different *BBlock* sizes. For clarity, only a subset of the networks is shown. Memory footprint and energy reduction are normalised to the baseline TPU without *Boveda*

compression. In general, larger *BBlock* sizes present lower compression rates, and in some cases they even increase footprint due to the padding. Since metadata is relatively small, *BBlocks* of 4, 8, and 16 show similar compression, except for SSD-MobileNet where increasing the *BBlock* increases the memory footprint. SSD-MobileNet has several small layers which benefit with smaller *BBlocks* as this reduces padding overhead. In contrast, the sweet spot for energy efficiency is a *BBlock* of 16. While the compression rate is virtually identical with other smaller *BBlock* sizes, using a *BBlock* size of 16 uses fewer and wider memory reads and fewer metadata reads.

J QUANTISATION: SENSITIVITY STUDY

To investigate *Boveda*'s potential effectiveness should quantisation to even smaller bitwidths becomes possible by generate synthetic 6b, 4b (we studied PACT), and 3b models by scaling existing 8b layers to fewer bits while maintaining the relative distribution of values. Figure 12 shows ideal compression rates for a representative subset of these layers compressed in *BBlocks* of 8. The results show that *Boveda* remains effective for 4b layers. For 3b layers, on occasion *Boveda* fails to reduce its footprint or even expands it.

In general, *Boveda*'s compression rate depends on the value distribution and is given by:

$$Comp. = 1 - \frac{\sum_{bl=Bmin}^{Bmax} P(X = bl) \times bl + \frac{\lceil \log_2(Bmax) \rceil}{BBlock}}{Bmax} \quad (1)$$

where *Bmax* is the maximum bit length, $P(X)$ the probability to have a certain bit length given by the value distribution, and *Bmin* is 2 for signed values and 1 otherwise. For signed values, maximum compression is achieved when P(X = 2) = 1. For 3b and a group size of 8, maximum compression is limited to 25%, while for 4b it is limited to 43.75%. This formula ignores padding and pointer overhead which depends on the dataflow, accelerator, and layer dimensions.