A OUR HARDWARE APPROACH

This section presents the *Schrödinger's FP* hardware encoder/decoder units that efficiently exploit the potential created by our quantization schemes. Without the loss of generality, we describe compressors/decompressors that process groups of 8 FP32 values. This section assumes that the reader is aware of prior work that demonstrated why it is possible and desirable to encode tensor values using variable length containers when storing them to external DRAM, e.g., (Han et al., 2016c; Judd et al., 2016a; Han et al., 2016a).

Overview of Hardware: At high-level, our hardware compressors/decompressors transparently encode/decode tensor values just before the memory controller. When values are stored to external DRAM, the encoders efficiently encode the values to use as few bits as necessary. When values are read back from external DRAM, the decoders, expand the values to the original format. This way the rest of the on-chip memory hierarchy and compute units can remain as-is.

Compressor: The compressor accepts one row of 8 numbers per cycle. In the compressor's first stage, it subtracts the fixed bias from the exponents. The resulting differences along with mantissas are then processed by 8 Packer units and a width detector as shown in Figure 9. The mantissa quantizer method, whether *Quantum Mantissa* or *BitWave*, provides the same mantissa length for all values. Each value within the row is encoded using the same number of bits, calculated as the sum of the provided mantissa bitlength and the bitlength needed to store the largest exponent difference across the row. The width detector, as its name suggests, will detect how many bits are needed to represent the exponents of the entire row. This step is accomplished by performing an OR operation on all 8 exponent values, and then detecting the leading 1. It will output a 3b number to the packer and compressor output. The exponent lengths need to be stored as metadata per row. These are stored separately, necessitating two write streams per tensor; both streams are sequential, thus DRAM-friendly. Furthermore, because there are only 3 bits per group of 8 values, a single access to this metadata structure yields metadata for multiple groups. Accesses to this metadata structure will thus be much less frequent than that for the value containers.

To avoid wide crossbars when packing/unpacking, values remain within the confines of their original format bit positions as per the method proposed in Proteus (Judd et al., 2016b). In contrast to Proteus, however, here every row uses a different bitlength, the values are floating-point, the bitlengths vary during runtime and per row, and we target training. Each packer, shown in Figure 11, takes a single FP32 number masks out unused exponent and mantissa bits, and rotates the remaining bits to a position to fill in the output row. The mask is created based on the exp_width and man_width inputs. The rotation counter register provides the rotation count which is updated to (exp_width+man_width) every cycle. The (L,R) register pair is used to tightly pack the encoded values into successive rows. They are needed since a value may now be split across two memory rows. This arrangement effectively packs the values belonging to each column tightly within a column of 32b in memory. Since each row is the same total bitlength, the 8 packers operate in tandem filling their respective outputs at exactly the same rate. As a result, the compressor produces 8×32b at a time. The rate at which the outputs are produced depends on the compression rate achieved, the higher the compression, the lower the rate.

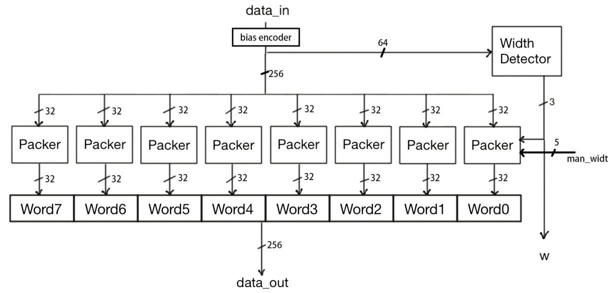

Figure 9: Compressor.

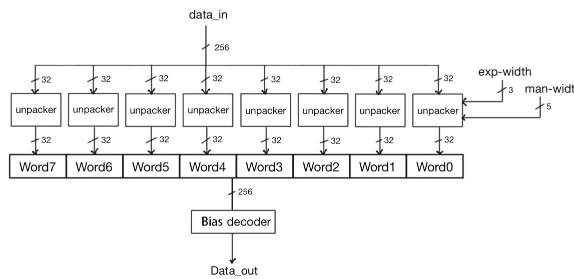

Figure 10: Decompressor.

Decompressor: As Figure 10 shows, the decompressor mirrors the compressor. The inputs to the unit are a 3b exp_width, a 5b man_width, and a 8×32b compressed data input. Since the data is compressed, a single row of 8×32b will typically contain data from more than one original uncompressed row of FP32 numbers. The compressed data values are packed into 8 virtual columns withing each row. Accordingly, each of the 8 virtual columns of 32b is fed into a dedicated unpacker.

Each unpacker, shown in Figure 12, has a wide 64b register that is internally divided into L and R registers of 32b. They are used in a similar fashion as the corresponding registers of the packer unit. At any point in time, one of the registers is used to accept a new row of 32b packed

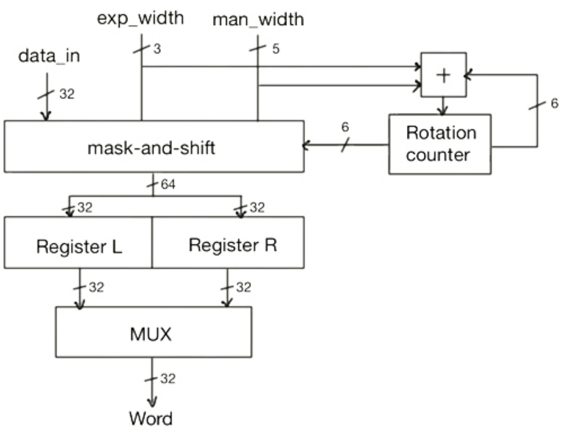

Figure 11: Packer.

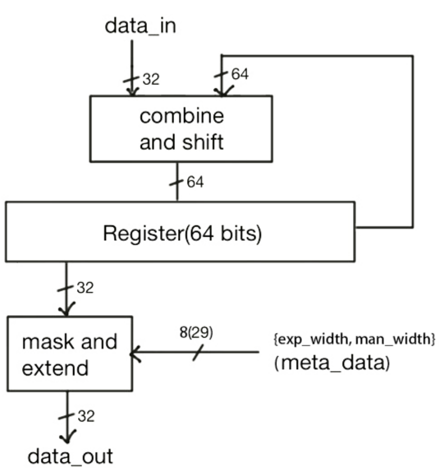

Figure 12: Unpacker.

Table 3: Hardware Area Overhead.

module	area per unit (um^2)	unit number	total area (mm^2)
compressor	31575.60	16	0.505
decompressor	37133.28	16	0.594
accelerator	38533.68	8000	308.27

data whereas the other contains whichever bits from the previous row of 32b have not been used yet. The combine-and-shift will combine the input data and previous data in the register then shift to the left. The number of shifted bits is determined by the exponent and mantissa lengths of this row. The 32b data on the left of the register are taken out and shifted to the right (zero extending the exponent). Finally, the unpacker reinserts the mantissa bits that were trimmed during compression. Since each row of data uses the same total bitlength, the unpackers operate in tandem consuming data at the same rate. The net effect is that external memory see wide accesses on both sides.

B HARDWARE EVALUATION METHODOLOGY

Best practices for the evaluation of custom hardware architectures necessitate exploration and validation first via analytical modelling or via cycle-accurate simulation. Since training these networks takes several *days* on actual hardware, cycle-accurate simulation of the full process is impractical. To estimate performance and energy, we use the best practice approach by analytically modelling the time and energy used per layer per pass of a baseline accelerator. To do so, we use traffic and compute counts collected during the aforementioned full training runs. We record these counts each time a layer is invoked using PyTorch hooks. We model time and energy for memory accesses via DRAMSIM3 (Li et al., 2020). For modeling on-chip structures we use CACTI (HewlettPackard) for the buffers and layout measurements for the compute units and the *Gecko* compressors/decompressors. We use a commercial 65nm process to model the processing units and *Gecko* hardware. We implement the units in Verilog and perform synthesis via the Synopsys Design Compiler and *layout* using Cadence Innovus with a target frequency of 500MHz. Synthesis uses Synopsys' commercial Building Block IP library for the target tech node. We estimate power via Innovus using traces over a representative input sample to model properly signal activity. We used nominal operating conditions to model power and latency. There are two *Gecko* compressor/decompressor units per channel.

Due to the complexity and time cost of cycle-accurate hardware simulation, we have opted for an estimated time and energy consumption analytical model based on the proposed hardware description and the compressor-decompressor architecture. To compute the analytical model, we first analyze the network and retrieve its structure (layer input and output sizes, kernel sizes for convolutional layers, stride, bias and padding). We then calculate the compute operations that will happen for the general batch size (N) in both the forward and backward pass, as well as the number of parameters that must be stored in memory for activations, weights and gradients.

To take advantage of data reuse where possible we perform the forward pass in a layer-first order per batch. This allows us to read the weights per layer only once per batch. For the backward pass, we utilize the on-chip buffers for mini-batching with a layer-first order over a mini-batch of samples. Mini-batching reduces overall traffic by processing as many samples as possible in a layer-first order avoiding either having to spill gradients or reading and writing weights per sample per layer. The number of samples that can fit in a mini-batch depends on the layer dimensions and the size of the on-chip buffer.

Both $\mathrm{S}FP_{\mathrm{Q}}$ and $\mathrm{S}FP_{\mathrm{B}W}$ sample bitlengths per batch to a log file for both mantissas and exponents. These bitlengths are used to compute the number of mini-batches that can fit at every training step per layer on chip. Based on the number of sampled mini-batches (K) we compute the memory footprint generated on the forward pass for each method. After this, we calculate the footprint that stays on-chip and can be loaded from on-chip for the backward pass, and the footprint that goes to off-chip and has to be loaded to on-chip again for it. Based on these memory accesses, we use DRAMsim to simulate the number of compute-cycles that take the memory accesses to finish and we use the maximum cycles between compute and memory as the time constraint to calculate total computation time in the proposed hardware.

To calculate energy consumption and efficiency, we use the information gathered in terms of on-chip memory access cycles, off-chip memory access cycles and compute cycles. We estimate energy consumption for all components including the compressors and decompressors. We use the following equations to estimate energy consumption for our methods (all symbols are defined in Table 4):

$$\begin{gathered} E_{forward} = E_{compute\,fwd} + E_{offchip\,in\,act\,mem} + \\ E_{offchip\,wgt\,mem} + E_{offchip\,out\,act\,mem} + E_{onchip\,in\,act\,mem} + \\ E_{onchip\,wgt\,mem} + E_{onchip\,out\,act\,mem} + E_{read\,ops\,mem} + \\ E_{decomp\,act} + E_{decomp\,wgt} + E_{comp\,act} \end{gathered} \tag{17}$$

$$\begin{gathered} E_{backward} = E_{compute\,bck} + E_{offchip\,in\,act\,mem} + \\ E_{offchip\,wgt\,mem} + E_{onchip\,in\,act\,mem} + \\ E_{onchip\,wgt\,mem} + E_{read\,ops\,mem} + \\ E_{decomp\,act} + E_{decomp\,wgt} \end{gathered} \tag{18}$$

where,

$$E_{offchip\,in\,act\,mem} = \frac{MemCh \times P_{DRAM}}{Freq_{compute}} \times Cycles_{offchip\,in\,act} \tag{19}$$

$$\begin{gathered} E_{offchip\,wgt\,mem} = \frac{MemCh \times P_{DRAM}}{Freq_{compute}} \times \\ (Cycles_{offchip\,wgt} + Cycles_{offchip\,wgt\,grad}) \end{gathered} \tag{20}$$

$$E_{offchip\,out\,act\,mem} = \frac{MemCh \times P_{DRAM}}{Freq_{compute}} \times Cycles_{offchip\,out\,act} \tag{21}$$

$$E_{onchip\,in\,act\,mem} = Cycles_{onchip\,in\,act\,write} \times P_{onchip\,write} \tag{22}$$

$$E_{onchip\,wgt\,mem} = Cycles_{onchip\,wgt\,read} \times P_{onchip\,read} \tag{23}$$

$$\begin{gathered} E_{onchip\,out\,act\,mem} = Cycles_{onchip\,out\,act\,read} \times P_{onchip\,read} + \\ Cycles_{onchip\,out\,act\,write} \times P_{onchip\,write} \end{gathered} \tag{24}$$

$$E_{decomp} = P_{decomp\,(comp\,ratio)} \times \frac{Cycles_{comp\,to\,decomp}}{Freq_{compute}} \tag{25}$$

$$E_{comp} = P_{comp\,(comp\,ratio)} \times \frac{Cycles_{decomp\,to\,comp}}{Freq_{compute}} \tag{26}$$

$$E_{decomp\,act} = E_{decomp\,act(comp\,ratio)} \tag{27}$$

$$E_{decomp\,wgt} = E_{decomp\,wgt(comp\,ratio)} \tag{28}$$

$$E_{comp\,act} = E_{comp\,act(comp\,ratio)} \tag{29}$$

Table 4: Symbols definition table.

Symbol	Definition
$E_{compute\,fwd}$	Energy consumption of the compute module for the entirety of the computations in the forward pass
$E_{compute\,bck}$	Energy consumption of the compute module for the entirety of the computations in the backward pass
$E_{offchip\,in\,act\,mem}$	Energy consumption of the offchip memory transfers for the network input activations
$E_{offchip\,wgt\,mem}$	Energy consumption of the offchip memory transfers for the network weights
$E_{offchip\,out\,act\,mem}$	Energy consumption of the offchip memory transfers for the network output activations
$E_{onchip\,in\,act\,mem}$	Energy consumption of the onchip memory transfers for the network input activations
$E_{onchip\,wgt\,mem}$	Energy consumption of the onchip memory transfers for the network weights
$E_{onchip\,out\,act\,mem}$	Energy consumption of the onchip memory transfers for the network output activations
$E_{read\,ops\,mem}$	Energy consumption of loading operations from memory
$E_{decomp\,act}$	Energy consumption of decompressing activations in the decompressor
$E_{decomp\,wgt}$	Energy consumption of decompressing weights in the decompressor
$E_{comp\,act}$	Energy consumption of compressing activations in the compressor
$P_{decomp\,(comp\,ratio)}$	Power consumption by the decompressor when loading data from offchip memory at a specific compression ratio (see Table 5)
$P_{comp\,(comp\,ratio)}$	Power consumption by the compressor when writing data to offchip memory at a specific compression ratio (see Table 5)
$MemCh$	Number of available memory channels
P_{DRAM}	Power consumption of offchip DRAM
$Freq_{compute}$	Clock frequency of the hardware accelerator
$Cycles_{offchip\,in\,act}$	Compute cycles taken to read input activations from offchip memory
$Cycles_{offchip\,wgt}$	Compute cycles taken to read weights from offchip memory
$Cycles_{offchip\,wgt\,grad}$	Compute cycles taken to read weight gradients from offchip memory
$Cycles_{offchip\,out\,act}$	Compute cycles taken to read output activations from offchip memory
$Cycles_{onchip\,in\,act\,write}$	Compute cycles taken to read input activations from onchip memory
$Cycles_{onchip\,wgt\,read}$	Compute cycles taken to read weights from onchip memory
$Cycles_{onchip\,out\,act\,read}$	Compute cycles taken to read output activations from onchip memory
$Cycles_{onchip\,out\,act\,write}$	Compute cycles taken to write output activations to onchip memory
$P_{onchip\,write}$	Power consumption of a word write to onchip memory
$P_{onchip\,read}$	Power consumption of a word read from onchip memory
$Cycles_{comp\,to\,decomp}$	Compute cycles taken to decompress compressed data
$Cycles_{decomp\,to\,comp}$	Compute cycles taken to compress data

Table 5: $P()$ terms: Power consumption as a function compression ratio.

Compression ratio	Compressor power (mW)	Decompressor power (mW)
0.143 - 0.263	10.87	13.84
0.264 - 0.388	12.18	14.72
0.389 - 0.513	12.65	15.97
0.514 - 0.638	13.44	15.76
0.639 - 0.763	14.98	15.42