Journal of Systems Architecture 112 (2021) 101839

Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

A survey On hardware accelerators and optimization techniques for RNNs☆

Sparsh Mittal[a,*], Sumanth Umesh[b]

[a] Department of Electronics and Communication Engineering, IIT Roorkee, India
[b] Department of Electrical Engineering, IIT Jodhpur, India

ARTICLE INFO

Keywords:
Recurrent neural networks
Deep learning
GPU
FPGA
ASIC
Pruning
Parallelization
Low-precision

ABSTRACT

"Recurrent neural networks" (RNNs) are powerful artificial intelligence models that have shown remarkable effectiveness in several tasks such as music generation, speech recognition and machine translation. RNN computations involve both intra-timestep and inter-timestep dependencies. Due to these features, hardware acceleration of RNNs is more challenging than that of CNNs. Recently, several researchers have proposed hardware architectures for RNNs. In this paper, we present a survey of GPU/FPGA/ASIC-based accelerators and optimization techniques for RNNs. We highlight the key ideas of different techniques to bring out their similarities and differences. Improvements in deep-learning algorithms have inevitably gone hand-in-hand with the improvements in the hardware-accelerators. Nevertheless, there is a need and scope of even greater synergy between these two fields. This survey seeks to synergize the efforts of researchers in the area of deep learning, computer architecture, and chip-design.

1. Introduction

Unlike feed-forward networks such as CNNs (convolution neural network) or MLPs (multilayer perceptron), RNNs have a recurrent connection where the output of the previous timestep forms an input to the present timestep. RNNs store the temporal information in the form of memory and are able to capture long-term dependencies between input symbols. Due to these features, in recent years, RNNs have shown phenomenal success in several sequence learning tasks such as machine translation, language processing, image captioning, scene labeling, action/speech recognition, time-series forecasting, image/music generation, etc. RNNs power popular applications such as Google's voice search and Apple Siri. In fact, in 2016 and 2020, RNNs accounted for 29% and 21% of the TPU (tensor processing unit) workloads run in Google datacenters [1,2], which highlights the importance of RNNs.

A key challenge in the use of RNNs is their high computation and latency overheads. RNNs work on sequences of variable-length inputs/outputs and their inference latency depends on the length of the inputs. The dependencies between the input symbols prevent parallel computation of different symbols because of which they have a lower degree of parallelism than CNNs [3]. Also, the dominant computation-pattern of RNNs is MVM, which has low data-reuse. Due to these reasons, RNN inference incurs high latency. For example, a quantized version of LSTM (long short term memory) based neural machine translation model takes 1322 seconds on a pair of Haswell CPUs [4]. Compared to an LSTM, a bidirectional LSTM requires twice the bandwidth and duplicating the hidden layer for processing the inputs in the opposite direction. Although RNNs generally have fewer parameters than CNNs [5,6], activations consume a much larger fraction of memory in RNNs than they do in CNNs [7]. Further, RNN training involves a large number of epochs for convergence, which increases the training time and computation/memory overheads. Due to these factors, hardware acceleration of RNNs is not merely attractive but even imperative. It is also evident that many optimizations proposed in the context of CNNs [8] cannot be directly applied in the context of RNNs. To address this need, many recent works have proposed accelerators for RNNs.

Contributions: In this paper, we present a survey of GPU/FPGA/ASIC-based accelerators for RNNs. A recent paper has reviewed CPU-based techniques for RNNs [9]. Fig. 1 provides an overview of the paper. Section 2 provides background and classification of research works on key parameters. Section 3 reviews RNN accelerator architectures and Section 4 discusses many widely-used optimization techniques. Section 5 discusses the techniques for reducing the compute/memory/latency overheads of RNNs.

We review both commercial products and research techniques. We highlight the similarities and differences between different research works. Although many of the works arch over categories, we summarize a work in a single section only. This survey includes papers published between years 2014 to 2020. It is important to note that research in both RNN algorithms and accelerators is happening at a rapid race. Due to this, some of the comparative analysis and conclusion may no longer

☆ Support for this work was provided by Semiconductor Research Corporation.
* Corresponding author.
E-mail addresses: sparshfec@iitr.ac.in (S. Mittal), sumanth.2@iitj.ac.in (S. Umesh).

https://doi.org/10.1016/j.sysarc.2020.101839
Received 25 May 2020; Received in revised form 26 June 2020; Accepted 16 July 2020

Fig. 1. Organization of the paper.

hold. Due to this reason and the fact that different works use different evaluation strategies, we primarily focus on the key insights of the papers and present only selected quantitative results. For the same reason, while summarizing a paper, we have also shown its year of publication. Section 6 lists the avenues for future research and Section 7 provides concluding remarks.

The goal of this survey is to bridge the understanding of researchers from two fields: deep-learning algorithms and hardware architecture. The survey aims to inform researchers in deep-learning algorithms about the potential and limitations of various hardware platforms for accelerating the RNN algorithms. At the same time, it seeks to motivate and even challenge the researchers in the hardware architecture domain to innovate and optimize the computing platforms according to the needs of RNN algorithms. As RNN algorithms grow bigger and get applied in increasingly complex tasks, more than ever, there is a need of such synergistic efforts, and hence, we believe that our survey paper is timely. Table 1 shows the acronyms used in this paper.

2. Background and overview

2.1. A quick intro to LSTMs and GRUs

LSTM: A recurrent layer has an LSTM cell, which is executed repeatedly for every symbol in the input sequence. Fig. 2 shows the design of an LSTM cell. A cell has a cell-state and multiple gates for processing the information. At every timestep, a cell operates on two vectors: input vector (x_t) and output of the previous timestep (h_{t-1}), which is also called the hidden vector. A cell has four gates for recurrently updating the cell-state and computing the output. (1) The input gate (i_t) determines the impact of the current input on the cell-state. (2) The "forget gate" (f_t) removes the unimportant information from the current cell-state (3) The cell-update gate (g_t) decides the "input information" that is taken as the candidate for updating the cell-state (4) The "output gate" (o_t), which computes the output and decides what to pass-through.

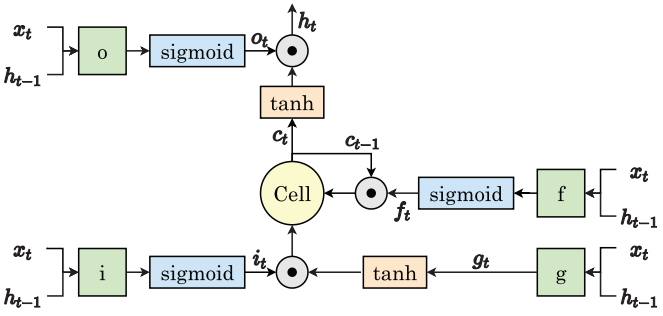


Fig. 2. Design of an LSTM cell [10].

Let 'w' be the "weight matrix" and 'b' be the "bias vector". Then, an LSTM network can be described using the ensuing equations:

$$i_t = sigmoid(w_{xi}x_t + w_{hi}h_{t-1} + b_i) \quad (1)$$

$$f_t = sigmoid(w_{xf}x_t + w_{hf}h_{t-1} + b_f) \quad (2)$$

Table 1
Acronyms used in this paper.

Acronym	Full-form	Acronym	Full-form
ADMM	alternating direction method of multipliers	MFE	multifunction engine
AF	Activation function	MKL	Math Kernel Library
BNN/CNN/ DNN/RNN	binarized/convolution/deep/recurrent neural network	MLP	multiplayer perceptron
BRAM	block RAM	MMM/MVM	matrix matrix/vector multiplication
BWP/FWP	backward/forward propagation	MRF/VRF	matrix/vector register file
CSC/CSR	compressed sparse column/row	NMT	neural machine translation
DMA	direct memory access	NPE	neural processing engine
DPE	dotproduct engine	NZ	non-zero
EW	element-wise	PCIe	peripheral component interconnect express
FC	fully connected	PE	processing engine
FMA	Fused multiply add	ReLU	rectified linear unit
GRU	gated recurrent unit	RF	register file
HLS	high-level synthesis	ShM	shared memory
ICI	Intercore interconnect	SIMD	single instruction multiple data
IPU	Intelligence processing unit	SM	Streaming multiprocessor
LSTM	long short-term memory	TPU	tensor processing unit
LUT	look-up table	VSM	vector-scalar multiplication
MAC	multiply accumulate	VVM	vector-vector multiplication

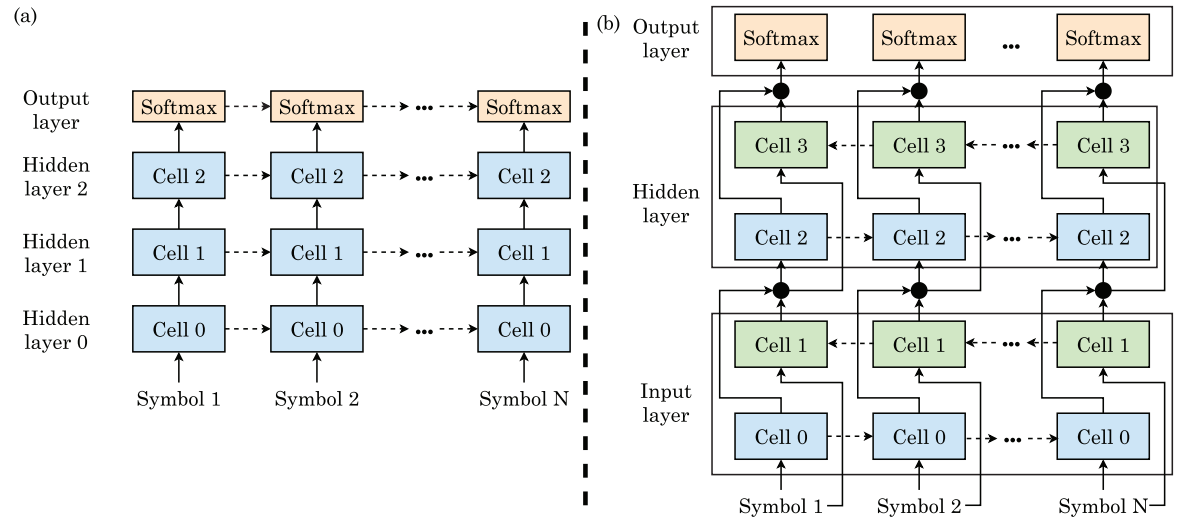


Fig. 3. (a) A uni-directional LSTM network (b) a bi-directional LSTM. Dotted-arrows show recurrent connections and solid arrows show forward connections.

$$o_t = sigmoid(w_{xo}x_t + w_{ho}h_{t-1} + b_o) \quad (3)$$

$$g_t = tanh(w_{xg}x_t + w_{hg}h_{t-1} + b_g) \quad (4)$$

$$c_t = f_t \odot c_{t-1} + i_t \odot g_t \quad (5)$$

$$h_t = tanh(c_t) \odot o_t \quad (6)$$

The computations performed by different gates are almost similar. The computation of every gate involves two MVMs: one for the forward connection and one for recurrent connection. Thus, in each timestep, an LSTM cell performs eight MVM operations. The LSTM computation has two types of dependencies: (1) intra-sequence: the activations of a gate need to be ready before updating the cell or generating the output (2) inter-sequence: computation of a timestep depends on the output of the previous timestep (h_{t-1}).

In a bidirectional RNN, there are two recurrent layers connected to the single output layer. One layer receives input from the future to the past, whereas another layer receives it from the past to the future. This bidirectional nature helps in properly understanding the context, which improves accuracy. Figs. 3(a) and 3(b) show the architecture of uni-directional and bi-directional LSTMs (respectively). Notice that the LSTM cell has two kinds of connections, viz. recurrent and forward connections. Several works have evaluated bi-directional LSTMs [5,7,11–14].

GRU: Most works reviewed in the paper study LSTM and only a few works have studied GRU [14–16]. GRUs [17] have been proposed only recently, but the interest in them has been growing over time. While LSTM has four gates, GRU has three gates: reset, update and output. Thus, the number of parameters and computations in a GRU cell is nearly 0.75 × that of an LSTM cell. GRU does not have a cell-state and uses the hidden state for transferring the information. GRUs have lower training time than LSTMs on less training data. LSTMs are able to remember longer sequences than GRUs and hence, they are better for tasks that need to capture long-term dependencies. However, there is no clear winner between LSTM and GRU for all the tasks.

2.2. A classification of research works

Table 2 organizes the works based on the deep learning phase (inference/training) and the framework used by them. It also mentions the tool used for programming the FPGAs. It further lists the computing platform used for evaluation or comparison purposes and the optimization objective of different works.

Table 3 shows the application areas for which different techniques were evaluated and the dataset for which training/inference was performed.

3. RNN Accelerator architectures

The FPGA/ASIC accelerators can be classified in two categories based on the abstraction they provide: (1) Some accelerators [1,3,12,14,16,37,39,43–45,55,70,72,78,89] provide a compute-substrate, such as engines for performing MVM/MMM and EW-operations. Their compute-substrate and the interconnect may not have a one-to-one correspondence with the RNN gates/dataflow, but their computing power can be harnessed for a broad range of NNs. Examples of these accelerators are Google's TPU or Microsoft's Brainwave. GPU-based accelerators also fall into this category. (2) Some accelerators [13,29–31,33–36,38,41,46,56,68,69,75,85,90] are specialized for RNNs, especially LSTMs. They offer the abstraction of gate units. For example, they may have four dedicated gate units, each for processing one gate of LSTM, along with the specific hardware for performing other operations such as AF and EW-operations. These accelerators may not run other NNs easily or efficiently, but they provide high efficiency on RNNs.

In a hardware accelerator, the core-computation engine decides the "unit of computation" that can be performed in one go. The computation engine may be either vector-scalar multiplication (VSM) [85], vector-vector multiplication (VVM) [33], matrix-vector multiplication (MVM) [3,41,43,45,55,75,78,86,89] or matrix-matrix multiplication (MMM) [1,27,30,39,72]. The architectures based on these engines are discussed in Section 3.1 through 3.4. In terms of the convention used in BLAS (Basic Linear Algebra Subprograms) library [91], these computations are either scalar or vector-only computations (level-1), matrix-vector computations (level-2) or matrix-matrix computations (level-3). Thus, both VSM and VVM engines perform level-1 computations. A low-level engine provides higher flexibility but incurs large latency while executing higher-level computations. A higher-level engine reduces bandwidth needs by exploiting data-reuse and hence, can provide high throughput. A higher-level engine may remain unutilized when it performs low-level computations, or in case of sparse computations, small-size problems or with small batch-sizes. In this section, we also review the architectures based on multiple FPGAs (Section 3.5), those for bi-directional LSTMs (Section 3.6) and n-dimensional LSTMs (Section 3.7).

Table 2
A classification based on machine-learning phase, framework, computing platform and optimization objective.

Category	Reference
Machine learning phase	
Training	[7,18–27]
Inference	[5,7,11,12,27–46,46–66]
Both	[3,13,67–77]
Deep-learning framework	
TensorFlow	[3,5,10,12,21,22,28,31,39,42,48,52,58,59,70,72,78,79]
Pytorch	[7,23,54,61,74,77]
Keras	[5,33,51,72,79]
Others	Theano [14,50], TensorRT [3], Caffe [12,76], Torch7 [41,45], MXNet [24]
FPGA programming tools/language	
Vivado HLS	[5,10,13,14,38,42,46,48–51,53,74,76]
OpenCL	[39,55,72]
Others	Verilog [16], System Verilog [43,79], Xilinx SDx HLS [70,73], Quartus Prime [43,44,80], MATLAB [50], System C [20]
Computing system/platform	
GPU	Volta GPU [22,23,25,59,60,65], Tesla P100 [23], Tesla M40 [15,21], Tesla K40 [26], Pascal Titan X [24,64], GTX Titan X [16], Titan X [57], Quadro P2000 [23], Jetson TX1 [41,61], Adreno GPUs [58]
FPGA	Xilinx Virtex 6 [20], Xilinx Virtex 7 [45,46,51,73,76,78,79], Xilinx Virtex UltraScale [14,49,81], Xilinx Zynq 70xx [10,38,41,42,45,45,48–50,53,74,75,78], Xilinx Zynq UtraScale [13,47], Xilinx Kintex UltraScale [5,40,70,72,73,82], Xilinx Kintex-7 [71], Altera Stratix V [3,16,39,55], Altera Stratix 10 [3,44,80], Altera Cyclone 5 [71], Altera Arria 10 [3,16,43,44,52,83]
ASIC	[1,3,7,11,12,16,19,29–31,33,34,36,37,49,66,67,69,71,84–86]
Others	Multi-FPGA [5,44], FPGA+ASIC [80], CPU+FPGA [44], IBM TrueNorth [71]
Optimization objective	
Energy	[3,5,10,13,16,29–31,34,36–39,41,43,46,48,49,53,54,56,61,66,68,70,71,71,74–76,78,81,83,85–87]
Performance	Nearly all

Table 3
Benchmark/application area of different works and the datasets used by them.

Category	Reference
Benchmark/Application areas	
DEEPSPEECH	[11,23,37,57]
Deepbench benchmark suite	[61,80]
Speech recognition	[7,11,12,23,29–35,37,40,43,52,54,57,60–62,67,68,71,75,77,79]
Neural machine translation	[1,11,14,22,24,28,33,34,44,59–61,65,79]
Language modelling	[19,22,24,31,36,41,43,63,67,69,77,86]
Image captioning	[22,29,34,65]
Sentiment analysis	[11,18,25,34,79,83]
Others	Natural language processing [29,67], video action recognition [77], video classification [33], sentence completion [20], EEG signal processing [49], question answering [81], in-hospital mortality rate prediction [88], keyword spotting [86]
Datasets	
Penn Treebank Corpus	[16,19,20,22–24,31,32,39,43,61,62,67,69,77]
TIMIT	[30,31,35,40,43,50,52,54,55,62,70,71,73,77]
WMT	WMT 15 [11,34,59], WMT 16 [22,28,65]
Others	TED-LIUM [11,12,34], MSCOCO [22,34,65], LibriSpeech [7,11,37], AN4 speech dataset [23,67], Stanford treebank [25,64], bAbI dataset [61,81], Google speech commands [68], THCHS-30 [86], SVHN [71], MSVD [28], SST Dataset [18] IWSLT 15 [28], DIBCO [74], Fraktur dataset [56], Smartphone sensor dataset [58], Wikitext-2 [24], TIDIGITS [66], MIMIC 3 [88], VCTK [77], UCF101 [77]

3.1. Architectures based on VSM engine

Yazdani et al. [85] (2019) present an accelerator for LSTM networks. The accelerator has a pipelined design with four stages. The PE and add-reduce units perform MVM while the activation unit performs AF. "Cell-update unit" uses the results from four gates for updating the cell-state and producing the hidden states for the next timestep. The input/hidden buffer is designed using SRAM since it is modified frequently and requires low latency. The "weight buffer" is architected as a multi-banked eDRAM memory. The latency of eDRAM is similar to that of SRAM [92,93] and the weights are written only once for every LSTM layer. They also store the cell-state and intermediate results on-chip.

The PE uses "vector-scalar multiplication" (VSM) as the core kernel and realizes VVM and MVM by combining VSM in different dimensions. A K-width VSM engine multiplies an input/hidden element with K elements of a row of the weight matrix. Previous works (e.g., [3]) that use VVM as the core kernel require two-levels of reduction, whereas the proposed work requires only one-level of reduction. By allocating P VSM units in a row-wise or column-wise manner, they can generate MVM engines of various dimensions for performing gate-computations in a tiled manner. The add-reduce unit is designed using a tree-adder. Pipelining of all the levels of this tree brings its effective latency to one cycle. The activation MFE performs floating-point computations. The sigmoid function is computed in three steps: exponentiation, addition and division. By pipelining, its effective latency is also reduced to one cycle. The cell-updater has all the functionalities of activation MFE plus a dot-product unit. On pipelining, in each cycle, the cell-updater computes K/4 elements of hidden outputs, which combine the output of 4 gates.

Fig. 4 shows various scheduling strategies used by different works. In the sequential strategy [3,40], the gates are serially computed. The

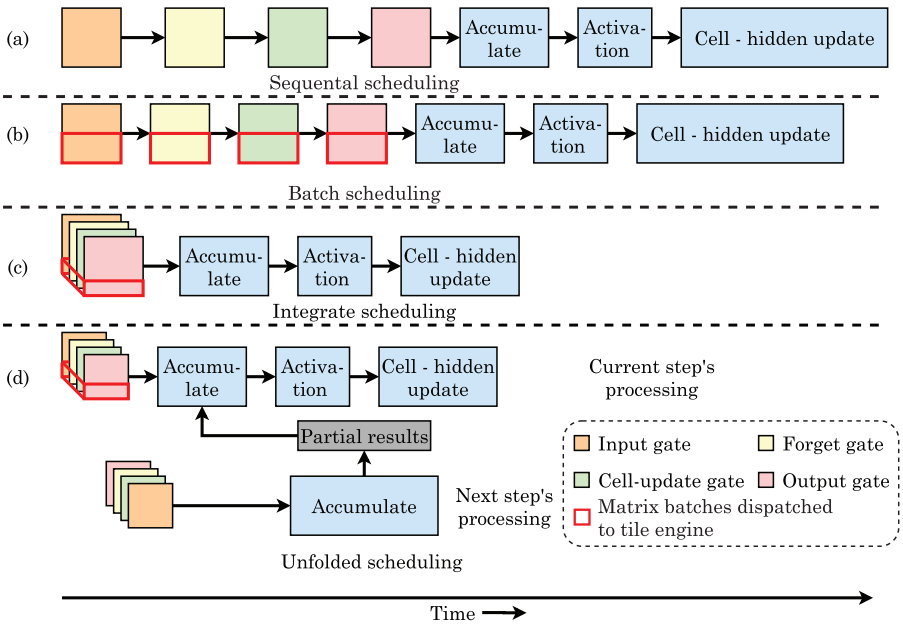


Fig. 4. (a) Serial scheduling (b) Batched scheduling (c) Intergate scheduling (d) "Unfolded" scheduling [85].

batch-strategy is a variant of the sequential strategy that computes a batch of every gate at a time. This allows pipelining the entire LSTM computation. In the intergate strategy [33,70], the MAC resources are shared for the multiplication of all the gates simultaneously. The intergate strategy overlaps the update processes of cell-state and hidden-state.

They propose a scheduling strategy called unfolding. Here, the input MVM of every timestep is processed and its result is stored in a buffer. Then, the output of the hidden MVM is accumulated with the input results stored in the buffer. Then, AF is applied and the cell-state and hidden-state are updated. Since the input vectors are not mutually dependent, in the unfolding strategy, the processing of cell-state and hidden-state at time t is overlapped with the computation of input MVM of time $t+1$. While the intergate strategy hides only intra-timestep dependency, the unfolding strategy hides both intra-timestep and inter-timestep dependencies. Due to this, PE utilization is greatly improved. In order to improve the efficiency of memory accesses, the weights corresponding to the input and hidden states are separately stored. To reduce the padding requirement, the tile size is adapted upon reaching the last row. Also, the MVM tile-engine is reconfigured based on the dimension of every LSTM model. Their accelerator provides high performance and energy efficiency for a range of LSTM models and resource constraints.

3.2. Architectures based on VVM engine

Silfa et al. [33] (2018) present an architecture for accelerating large LSTMs. For large LSTMs, all the weights cannot be stored on-chip and on the other hand, storing all the weights in off-chip memory incurs large overhead. Their proposed design balances these tradeoffs by storing the weights of only one LSTM layer in the on-chip memory. This is possible since performing either a forward or backward pass on a layer by evaluating the entire input sequence requires only the weights of that particular layer. Since input sequences have many elements, the weights can be reused for those elements. To further reduce the memory overhead, both input sequences and weights are linearly quantized to 8-bits. Their proposed accelerator is shown in Fig. 5(a). It has four PEs, each of which handles one of the LSTM gates. Having a dedicated PE for each gate is helpful since the majority of cell computation does not require information exchange between the gates. Since every gate evaluates two MVMs for every input element, the load on four PEs is balanced. However, at the end of cell state computations, there is some data-exchange between PEs, which is shown by the arrows in Fig. 5(a).

Due to the data-dependencies of LSTMs, interim results of a layer for a whole input sequence need to be saved. They use a dedicated on-chip buffer for storing them. This buffer is split into two portions, which store the output of the current and previous layers, respectively. This is because the previous layer's output can be discarded only when the entire input sequence has been evaluated. Each PE has two key units, as shown in Fig. 5(b): DPE and MFE. The DPE splits two vectors into multiple equal-sized sub-vectors and then computes an integer dot-product between them. In each timestep, two sub-vectors are multiplied and their results are accumulated over time. The MFE performs remaining operations such as peephole computations, activation, and linear quantization in floating-point arithmetic.

Even after these optimizations, on-chip memory needs of some workloads remains quite high. To address this challenge, they propose a technique which can be understood as follows:

Baseline scheme: Here, in a gate, all the neurons are evaluated for one element (x_t) of the input sequence. Thus, the neurons in a layer are sequentially evaluated first for x_t and h_{t-1}, and then for x_{t+1} and h_t.

Proposed technique: They note that the recurrent links need to be processed in a fixed sequential order to respect the data-dependencies. By contrast, the forward connections from one layer to the next can be evaluated in any order because all the outputs from the last layer are already present. Based on this, their technique computes forward connections in an order that is suitable for ensuring maximum reuse of weights. Specifically, their technique proceeds in two phases. In the first phase, evaluation of all neurons is done using forward links for the entire input sequence $(x_t, \ldots, x_n)$ as the input. Then, interim results are saved. In the second phase, computation of all neurons is done in the sequential order for the recurrent connections $(h_{t-1}, \ldots, h_{n-1})$.

Notice that all "feed-forward networks" that consume x_t do not require the previous output of the layer. Hence, their technique partially evaluates every neuron in a layer for the whole "input sequence" and then proceeds with recurrent links. In effect, their technique lowers the reuse distance of a subset of weights, as shown in Fig. 6. Once the reuse for that subset is over, it can be evicted and thus, at any point of time, forward weights of only one neuron need to be stored in the on-chip memory. However, the storage requirement of recurrent weights (W_h) is not reduced. The limitation of this approach is that it needs extra

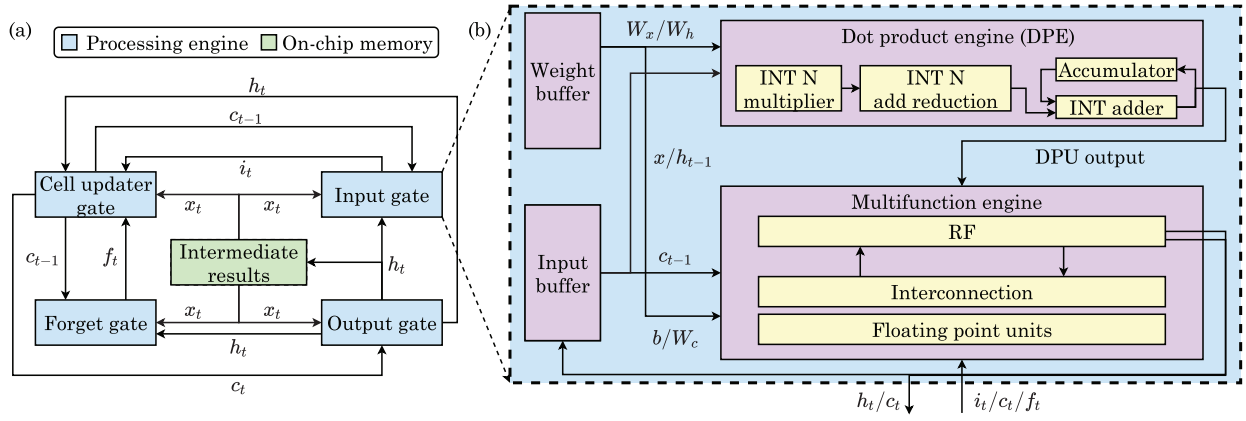


Fig. 5. (a) Accelerator proposed by Silfa et al. [33] for LSTM with peephole connections (b) PE architecture.

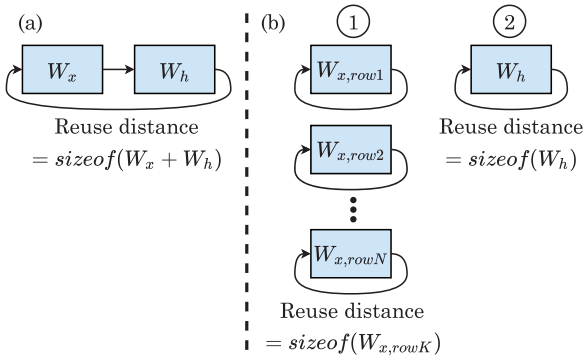


Fig. 6. Reuse distances in the (a) baseline scheme and (b) the technique proposed by Silfa et al. [33].

memory for storing the partial results of all neurons of a layer. The partial outputs are linearly quantized to overcome this requirement. Nonetheless, these values have to be transformed back to floating-point for adding it with the output of recurrent links for obtaining the final results of a gate. Their technique achieves high performance and energy efficiency.

3.3. Architectures based on MVM engine

Fowers et al. [3] (2018) present the architecture of Brainwave, a cloud-scale FPGA-based accelerator for DNNs, including MLPs, 1D/2D CNNs and LSTM/GRU RNNs. Brainwave optimizes the latency of individual requests, which, in turn, leads to high throughput. In other words, Brainwave optimizes throughput without sacrificing individual latency. The Brainwave neural processing engine (NPE) uses a single-threaded SIMD "instruction set architecture" (ISA). It has compound operations for operating on N-length vectors and $N \times N$ matrices. Vectors and matrices are handled as different datatypes and separate RFs are used for storing them. These RFs are called vector RF (VRF) and matrix RF (MRF), respectively. The NPE datapath is designed as a coprocessor which uses a "scalar core" for issuing instructions to the datapath through an "instruction queue". The "scalar core" orchestrates the NPE control flow based on the input, which is crucial for RNNs with variable-length timesteps.

NPE uses "explicit instruction chaining" where dependent instruction sequences pass values directly between successive operations. This allows exploiting pipelining and obviates the need for RFs or dependency-checking. NPE allows operating on multiples of the native dimension, which eases the parameterization of models. It also reduces the number of instructions, e.g., for one of their GRU models, one instruction can issue above 7 million operations. Fig. 7 shows the NPE design. It seeks to run the instruction chains on a continuous stream of vectors passing through the function units. Movement of data between the memory elements is managed by the "vector handling network". The scheduler sends the control signals to the function units and vector handling network.

They use an MVM engine as the key computation engine due to the flexibility it provides. During inference, model weights remain constant and hence, their storage is specialized. Fig. 7 also shows the design of MVM engine. It has multiple "matrix-vector tile engines" for implementing a native-sized MVM. A tile engine has multiple DPEs, each of which multiplies the input vector with a row of the matrix tile. A DPE has lanes of parallel multipliers connected to an accumulation tree. Figs. 8(a) and (b) show the mapping of vector-matrix multiplication to an MVM engine having two tiles, two dot-product engines and four lanes. An MVM engine parallelly executes the dot-product between an input vector broadcast from VRFs and the matrix rows stored in the distributed MRFs. Overall, the NPE leverages parallelism in four ways: across-MVMs, MVM-tiling, across the rows of a tile and across the columns of a row. In each cycle, the MVM engine performs 2 x #TileEngines x #DPEs x #Lanes computations. Every input to each of the DPE engines is provided by a dedicated memory port in the MRF to maximize throughput with MVM engine.

MFEs perform vector-vector operations and AFs. For supporting longer sequences of vector operations, two or more MFEs can be chained. For performing dot-products, they use a "block floating-point" (BFP) format where 5b exponents are shared across 128 independent sign and mantissa values. By using its variant, they reduce mantissa to just 2 to 5 bits with minor accuracy loss. Thus, the BFP has 1 sign-bit, 5 exponent bits and only 2 or 5 mantissa bits. It can be seen in Fig. 7 that the matrix-vector multiplier converts the 16-bit float (FP16) vectors to BFP, multiplies it with the matrix and converts the BFP back to FP16. Use of BFP requires only few additional training epochs but no hyperparameter-tuning.

Multiple low-precision computations can be packed in a single "digital signal processor" (DSP). They implement Brainwave architecture on a Stratix 10 FPGA and evaluate it with LSTM and GRU benchmarks using a batch size of one. Brainwave latency remains below 4ms for all benchmarks and the peak throughput is nearly 36 Teraflops. Also, Brainwave achieves high improvement in throughput and latency over the latest GPU.

Nurvitadhi et al. [80] (2019) perform a comparative evaluation of RNN execution on GPU, FPGA and FPGA + ASIC. They model Microsoft's FPGA-based Brainwave architecture [3] in SystemVerilog. In their simulation framework, they choose the Brainwave configuration providing the highest performance, within the FPGA resources present, for three precisions: INT4, INT8 and FP32. For INT4 and INT8 pre-

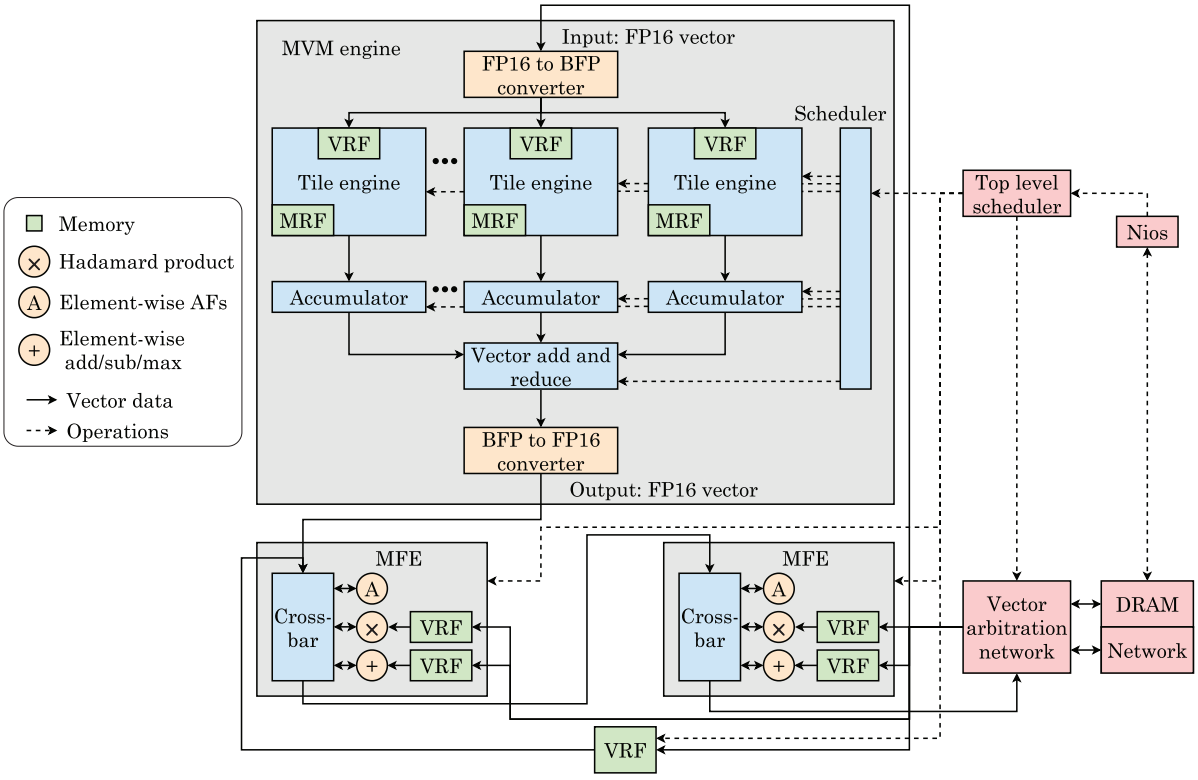


Fig. 7. NPE design in the Brainwave architecture [3].

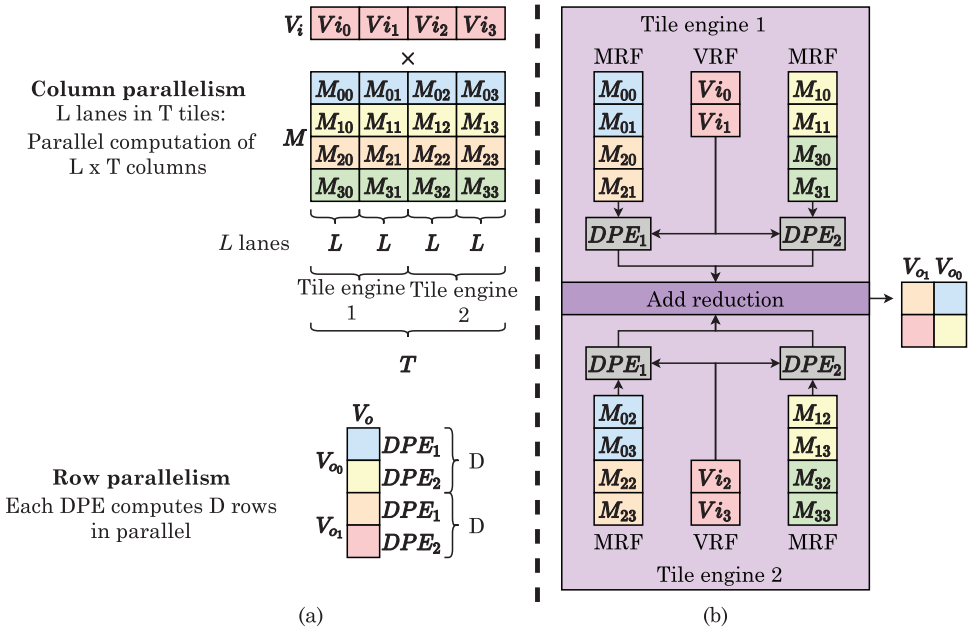


Fig. 8. (a)Parallelism in vector-matrix multiplication through tiles, dot-product engines and lanes [3,80] (b) Mapping a vector-matrix multiplication to an MVM engine.

cisions, they pack six and four multiplications in each DSP block, respectively.

Stratix 10 FPGAs allow low-latency and low-energy integration with multiple ASICs in a single package. They propose a "TensorRAM architecture" which is shown in Fig. 9(a). It has fast RFs and SRAMs and multiple near-memory compute-engines that match the memory bandwidth. It can work in either memory mode, similar to a RAM with fixed read/write latency or in computation mode, where computations are performed near SRAMs. In computation mode, every PE works on the operands from the RF and those from its corresponding SRAM bank. Their deep-learning accelerator (implemented in FPGA) offloads its MVM engine computations to TensorRAM (implemented in ASIC). This improves area and performance efficiency. The higher frequency and memory/compute capability of ASIC allows achieving higher performance. Also, it reduces the resource requirement of FPGA and releases it for performing other functions. The operations of the proposed "TensorRAM" architecture in memory mode and computation mode are shown in Figs. 9(b) and 9(c), respectively.

They evaluate RNN, LSTM and GRU benchmarks from DeepBench. As for the computing systems, they evaluate a Volta GPU, their accel-

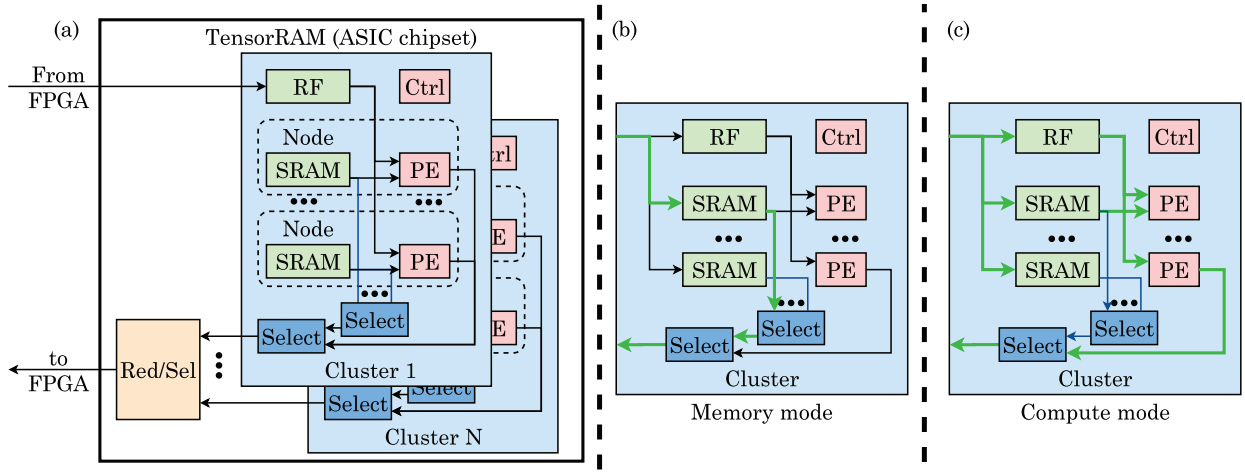


Fig. 9. (a)TensorRAM architecture proposed by Nurvitadhi et al. [80] (b)Operation in memory mode (c) Operation in computation mode.

erator implemented on Stratix 10 FPGA and FPGA-ASIC design, where ASIC implements the TensorRAM architecture. They use the persistent approach where the model is stored on-chip in BRAM of FPGA or caches and RF of GPU [94]. Persistent execution is used in Microsoft's Brainwave and NVIDIA's cuDNN. They find that the FPGA achieves lower latency than the GPU and they achieve 57% and 6% (respectively) of their peak performance. Further, FPGA + ASIC design achieves even higher energy efficiency and performance than the GPU. The total on-chip memory of FPGA + ASIC is twice that of a high-end GPU or FPGA. It is important to note that TensorRAM uses INT8 precision, whereas they evaluate Volta GPU using FP32 precision. Volta GPU also supports FP16 and a FP16 implementation can provide up to 8 × throughput than a FP32 implementation.

Li et al. [14] (2019) accelerate NMT using FPGA. The NMT uses bidirectional GRUs as the encoder and "attention with beam search" as the decoder. Since the encoder accounts for a negligible fraction of total operations, they focus on accelerating the decoder. Their design stores bias and weight values in off-chip DRAM and all the interim results in the on-chip buffers. As data propagates through different layers, the buffers of one layer are reused to store the interim results of the next layer. Sharing computation resources (DSPs) between layers avoids resource-idling, yet it requires a large number of MUXes. To balance this tradeoff, they develop an IP for MVM kernels which has input/output buffers of pre-determined size. This IP is used for all the MVM kernels. The IP has multiple PEs consisting of MAC units with registers and an adder tree. Large MVMs are split to match the size of the input/output buffer of the IP. In order to match the dimension of beam-search algorithm, two types of MVM templates are designed. The first one handles single MVM and the second one handles multiple (e.g., five) MVMs that have the same weight. In the encoder and the first stage of decoder, a single MVM is performed in every timeslot.

If the width of the beam-search algorithm is five, then up to five decoding processes are executed that share the same weights. Hence, if these decoders are run simultaneously, these weights need to be loaded only once from the off-chip memory, which improves data-reuse. For the MVMs of the beam-search algorithm, the second MVM template is used. Since the PE has internal registers, they pipeline every PE to improve throughput. They find that the performance of one pipelined PE can reach that of multiple PEs run in parallel that use the same set of buffers. Hence, for pipelined design, they use a single PE for handling multiple pieces of data, but for the non-pipelined design, the number of buffers is matched to the number of PEs. They implement their technique on an FPGA and show that it achieves high performance.

Chang et al. [41] (2017) present three architectures for accelerating RNNs on FPGA. (1) In the first architecture shown in Fig. 10(a), data is streamed from off-chip memory to PEs. Its primary building block is the "gate engine" which performs MVM and AFs. Every gate has two MACs which concurrently compute $W_x x_t$ and $W_h h_{t-1}$. These results are added and passed through an EW AF. The inputs are temporarily stored in a "vector memory". The interim results from gate engines are stored in FIFO buffers. These results and c_{t-1} vector obtained through DMA are used by the EW block to compute the final result. c_t and h_t are stored in main memory. Since 4 DMA ports are used, 2 gate engines (i.e., 4 MAC units) can run concurrently. Although this design achieves high MAC utilization, it uses only 4 MACs in parallel and is bottlenecked by off-chip bandwidth.

(2) The second architecture, shown in Fig. 10(b), stores all data in on-chip memory. Its operation is similar to the first architecture, except that all the matrix rows are saved in "weight memory" and 128 gate engines are used for simultaneously multiplying all the matrix rows with the vector. The non-linear activation engine is shared between all gate engines. At the time of initialization, all the weights are loaded into on-chip memory using DMA. This architecture has high performance and low bandwidth requirement, however, since the on-chip memory size is limited, this architecture cannot be used for RNNs with large weight matrix.

(3) Their third architecture, shown in Fig. 10(c) addresses the limitations of previous architectures by storing only part of weight data into on-chip memory. It uses a gate grid that consumes data from weight memory and vector memory. Both memories are double-buffered, which helps in overlapping computation with communication. However, due to limited data-reuse in RNNs, a complete overlap may not be possible. The gate grid has multiple MACs, each of which processes a different row of the weight matrix. The MACs in a row operate in SIMD fashion. Every MAC column has access to one bank of weight memory and a single vector memory is shared between all the MACs. This architecture offers high scalability. In terms of decreasing performance per unit power, the designs are second, first and third. All these designs achieve higher performance than the implementations on TX1 GPU.

3.4. Architectures based on MMM engine

Jouppi et al. [1] (2017) present the architecture of Google's tensor processing unit (TPU) (version 1). The core computation engine of TPU is an MMM unit that has a 256x256 array of MACs. A MAC operates on 8b unsigned/signed integers. The 16b results are stored in the accumulators, which hold 4096, 256-elements, each 32b wide. Fig. 11 shows the block diagram of the TPU. In every cycle, the MMM unit generates one 256-element partial sum. TPU works at half-speed when working

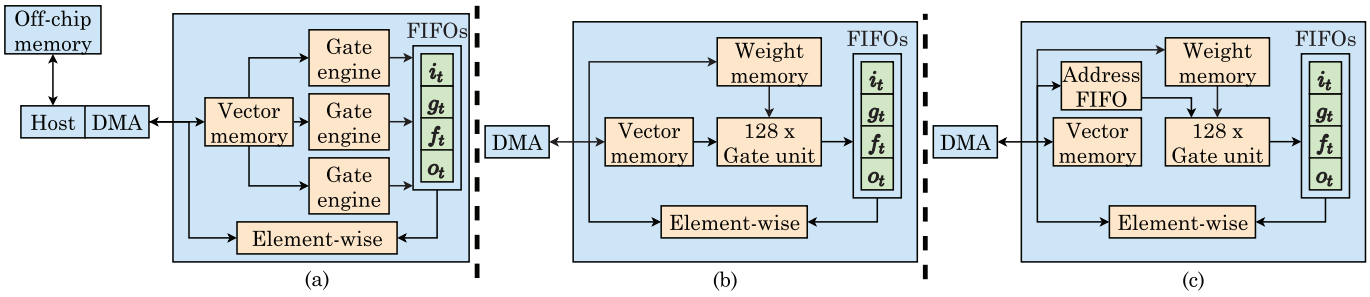


Fig. 10. RNN accelerators based on (a) streaming of data from off-chip memory to PEs (b) storing all the data in on-chip memory (c) storing only part of the weight data in on-chip memory [41].

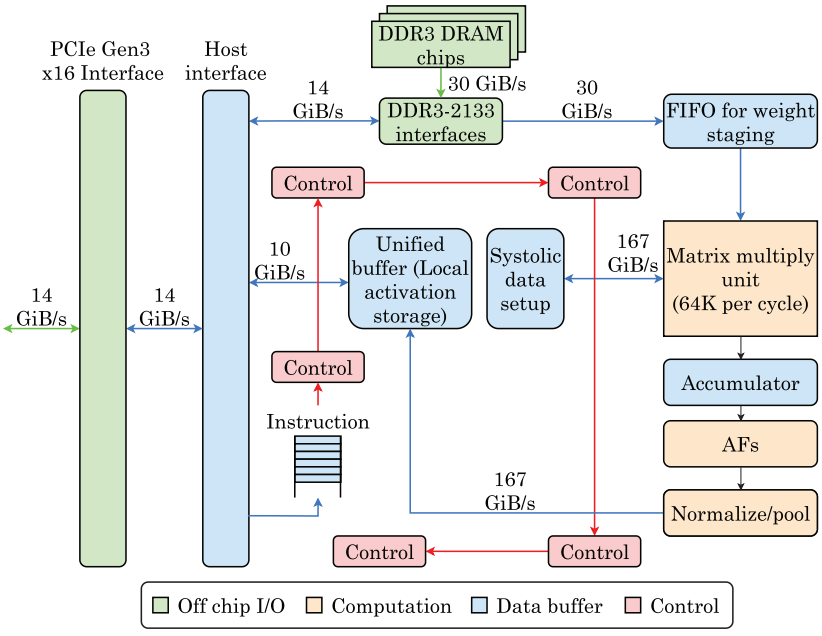


Fig. 11. Block diagram of TPU [1].

on 8b activations and 16b weights (or vice versa) and at quarter-speed if both span 16b. MMM unit can perform either matrix-multiplication or CONV. A "weight FIFO" is used for staging the weights to the MMM unit. It fetches the weights from the off-chip memory. The interim results are stored in the "unified buffer" which provides inputs to the MMM unit. As for the chip-layout, MMM unit, unified buffer and accumulators take nearly 70% of the chip's total area. Control circuitry takes only 2% area of the chip. TPU connects to the host (CPU) via a slow PCIe interconnect and to save its bandwidth, TPU uses "complex instruction set computer" (CISC) ISA with nearly 12 instructions.

TPU processes a DNN one-layer a time. TPU achieves high throughput by fully-utilizing the MMM unit. TPU has a 4-stage pipeline for hiding the latency of matrix-multiply instructions. In order to reduce the energy of accessing the unified buffer, the MMM engine uses systolic array style computation. Here, the input activations stream from the left and the weights stream from the top. A 256-element MAC operation propagates from the matrix in a diagonal direction. Pipelining of control and data hides the latency of reading inputs and writing output to the accumulators. TPU can run programs written in TensorFlow. It can run most of the DNN models completely, which reduces the overhead of data-transfer latency.

As for benchmarks, they use two LSTMs, two MLPs and two CNNs. They compare TPU with a Haswell CPU (released in 2013) and a K80 GPU (2014). The LSTMs and MLPs are memory-bound whereas CNNs are compute-bound. In the roofline models, the ceiling of TPU is higher than that of both Haswell CPU and K80 GPU. Although the peak performance of TPU is 92 teraops/sec, the two LSTMs achieve only 3.7 and 2.8 teraops/sec throughput. By contrast, one of the CNN achieves 86 teraops/sec. The reason is that while running LSTMs, TPU frequently stalls for fetching and shifting the weights. Hence, a large fraction of MACs in the MMM engine remains unused. Also, there are stalls due to accessing input data over PCIe and read-after-write hazards. Compared to the K80 GPU, TPU has 25 × MACs and 3.5 × on-chip memory, even though TPU has a much lower area than the K80 GPU. Finally, TPU has higher inference performance and performance/watt compared to Haswell CPU and K80 GPU. It is important to note that the TPU operates on 8b integers whereas both Haswell CPU and K80 GPU results have used 32b floating-point. A limitation of TPU is that it has poor "energy proportionality", which implies that even at 10% load, it consumes close to 90% of the power it consumes when fully loaded.

Jouppi et al. [2] (2020) discuss the architecture of TPUv2/v3. TPUv1 operates on 8b integers and hence, it is useful for inference only. TPUv2/v3 use BF16 and hence, are useful for both training and inference. Fig. 12(a) shows the architecture of TPUv2/v3 and Fig. 12(b) shows the simplified floor-plan of TPUv2. To meet the requirements of training, TPUv2 chips are connected as a 16x16 2D torus. Each chip has four "inter-core interconnect" (ICI) links which has a bandwidth of 496Gbits/s in each direction. Use of a fast interconnect and splitting the minibatch over the nodes reduces the tail latency and makes it possible to use synchronous parallel training and accelerate training. Similar to TPUv1, TPUv2 also has a large 2D systolic array (called MXU) to perform MMM, except that TPUv2 has two such cores in each chip. TPUv2 uses large batch sizes (e.g., 256 to 8192) to achieve high throughput. Since training uses multiple processors, use of two cores per chip avoids

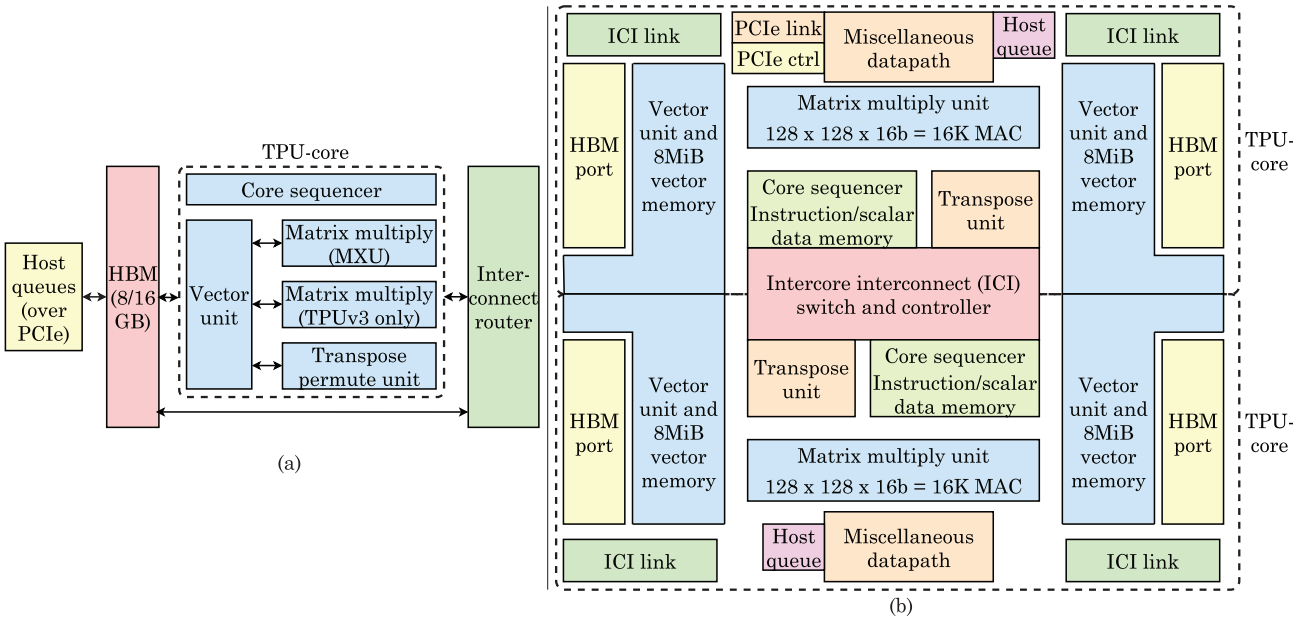


Fig. 12. (a) Block-diagram of TPUv2/v3 (b) a simplified floor-plan of TPUv2.

the high latency of a single large core, and is also easier to program than having multiple cores per chip.

TPUv2 uses "high-bandwidth memory" (HBM) which has 20 × the bandwidth of TPUv1. Thus, unlike TPUv1, TPUv2 is not memory-bound. The "core-sequencer" brings "very long instruction word" (VLIW) instructions from the on-chip instruction memory, performs scalar operations using a data memory and forwards vector instructions to the VPU. The 322b VLIW instruction can issue multiple operations, such as scalar/vector ALU, vector load/store, etc. The XLA compiler is responsible for scheduling load from instruction memory through independent code. The "vector processing unit" (VPU) executes vector operations using a on-chip vector memory and vector registers. The VPU performs streaming of data to and from the MXU. The VPU supplies data to vector memory by leveraging multiple operations per instruction and the vector functional units. To implement batch normalization, they divide it into vector additions and multiplications across the batch and one inverse-square-root computation (implemented by the transcendental unit). To reduce the number of vector operations, they made the ALUs and registers 128x8 instead of 128x1. The "transpose permute unit" performs 128x128 matrix tranpose, reduce and permute operations of the VPU lanes.

TPUv1 has 256x256 systolic array, which is not fully utilized for sparse DNNs or small-size DNNs. To partially alleviate this limitation, TPUv2 reduces the size of systolic array to 128x128 and double the number of MXUs. Thus, the total number of MACs is reduced from 64K in TPUv1 to 32K in TPUv2. While one 256x256 MXU takes nearly same area as four 128x128 MXUs, the latter has higher utilization for small CONV operations. This is because the smaller MXU sees less idling for small-size MMMs. Although sixteen 64x64 MXUs have even better utilization, but their area overhead is also high. In TPUv2, the MXU takes 16b FP inputs and generates 32b FP products. The remaining computations are in 32b FP, excepting the outputs going as an MXU input which are transformed into 16b FP.

Table 4 compares the architectural features of TPU versions and Volta GPU. A TPUv3-based supercomputer can use 1024 chips. TPUv3 requires liquid cooling. TPUv2/v3 support "BF16" format which uses 8 exponent and 7 mantissa bits. Compared to IEEE-754 FP16 format, BF16 format has lower area and energy, and also simplifies the software. Jouppi et al. also present a compiler, named XLA, for programming the TPU. Since TPU is a in-order machine, XLA takes care of overlapping the activities in computation/network/memory. The multithreading used in GPUs leads to high overhead and is not required for DNNs. The memory access pattern of DNNs is serial and easy to prefetch and hence, the simple systolic array architecture is more suited for DNNs. On CNN/RNN/MLP/transformer applications, TPUv2/v3 provided much larger performance than TPUv1. On scaling to 1024 chips, TPUv3 provides 96% to 99% of linear speedup.

In NVIDIA GPUs, the Volta tensor cores are able to perform matrix-multiply accumulate operation ($D = A \times B + C$) between 4x4 matrices having in a single cycle. Here, A and B are FP16 and their product is FP32. C is also in FP32 precision. The Turing tensor cores add the support for INT8 and INT4 precision modes also. Compared to CUDA cores, tensor cores have higher performance at the cost of precision. Also, while CUDA cores are capable of executing the entire CUDA program, the tensor cores are specialized at performing only the matrix multiply accumulate operation. When the size of matrices is a good fit for the supported basic WMMA shapes of tensor cores, the tensor cores deliver much higher performance than the CUDA cores (e.g., up to 8 × [65]). However, when this condition is not met, tensor cores provide poor performance [95]. Also, tensor cores are customized for dense matrix multiplication only [65]. Due to their potential for high performance, use of tensor cores is highly promising for RNNs.

Jia et al. [27] (2019) review the architecture of "intelligence processing unit" (IPU) from Graphcore, which is illustrated in Fig. 13. IPU seeks to efficiently execute fine-grained operations over numerous parallel threads. This makes IPU suitable for applications with irregular computations, memory-accesses and control-flow. IPU exploits "multiple instruction multiple data" parallelism. An IPU has 1216 PEs and operates at 1.6 GHz. Every IPU has 10 IPU links for communication across IPU processors and two PCIe links for communicating to the host CPU. In IPU, threads operate on small data-blocks exploiting "multiple instruction multiple data" parallelism. Different threads may have different code and execution flow, without incurring performance overheads. Each core is sufficiently complex to execute totally different programs.

Similar to CPUs and GPUs, IPUs obtain high performance by oversubscribing threads to cores. Every IPU PE supports 6 hardware threads and thus, there are 7296 threads in the IPU. Different threads execute in a static round-robin manner. The threads are multiplexed onto shared resources for achieving high throughput. Each IPU PE has customized pipelines termed "accumulating matrix product" (AMP). AMP boosts

MMM and CONV operations. In each cycle, AMP can complete sixteen FP32 operations or sixty-four mixed-precision operations. AMP is similar to the tensor-core of V100 GPU.

Each PE has one core and 256KB local memory, which is managed as scratchpad and not cache. Cores incur a fixed latency in accessing their local memory, regardless of the access pattern. The IPU also has RF and an on-chip interconnect for communication between PEs. IPU does not use any shared memory, rather it uses only distributed local memory with every core. The local memory is designed with SRAM which has an aggregate bandwidth of 45 TB/s and a latency of 6 cycles. This is vastly superior to the DRAM characteristics.

A limitation of IPU is that its total memory capacity is smaller than both CPU and GPU. However, this is compensated by the fact the latency of IPU's local memory is lower than that of shared memory in T4 GPUs and is close to that of L2 caches in recent CPUs. Each IPU chip has 304MB (=1216*256KB) memory, which is much higher than the on-chip memory of CPU and GPU. An IPU board has two IPU chips. Models requiring more than 608MB (=2*304MB) memory can be mapped to multiple processors and boards. Also, several optimizations can be used for mitigating the memory capacity limitations, such as data-compression [96], repeating lightweight computations instead of storing their results, using small minibatch-size, and using memory-efficient algorithms. This is especially suited for RNNs which are trained better with small minibatch sizes. On the DeepBench LSTM RNN benchmark, IPU provides higher throughput than P100 GPU [97].

Apart from irregular workloads, IPU also performs well on the regular, vector/matrix-based programs. The theoretical peak single-precision throughput of IPU is 31.1 TFlops/s, which is higher than that of V100 GPU (~5.7 TFlops/s [2,27]). However, for mixed-precision operations, their throughput is comparable (124.5 TFlops/s for IPU and 125.0 TFlops/s for V100 GPU). Further, on both IPU and GPU, the actual performance depends on various factors.

Conti et al. [30] (2018) design a chip for accelerating LSTMs. LSTMs perform three kinds of operations: MVM, EW vector operations and AF. Their proposed chip can perform these functions and stores the state-parameters on-chip. Also, a large amount of on-chip SRAM buffer is used for storing the weights. The code for computing the product between a weight matrix W of size $P \times Q$ and a vector X of size Q is as follows:

```
for p from 1 to P //row loop
for q from 1 to Q //column loop
z += W[p,q] * X[q]
```

Their chip executes the row loop on the parallel engines and the column loop in a sequential manner. Fig. 14(a) illustrates the block diagram of their chip. N parallel engines are used for simultaneously executing all the iterations of the row loop. Every engine has a memory unit for storing weights, registers for temporarily buffering i_t, f_t, o_t and c_t, a MAC unit and two LUTs for implementing AFs. x_t and h_t are stored in a bank with N registers. In every cycle of the column loop, based on the iteration-ID, one item each from input state and hidden state are sent to all the engines. Fig. 14(b) shows the operation sequence. State-variables are stored in 8b fixed-point and the MAC unit uses 16b. When the computation of a layer begins, weights are loaded and the inputs are serially streamed in.

They note that scaling a single engine is not feasible because the LSTM units are tied to the registers through multiplexers. Hence, the value of N cannot be increased beyond a few hundred units. To process larger networks, they connect multiple engines as tiles and distribute the RNN computations in a systolic architecture. Fig. 14(c) shows an example of this for a 3x3 array. Here, the weights are divided across different connected chips. The input state is divided into N-length vectors and every vector is broadcast vertically across a column. The updated value of internal states is obtained by summing up the output of every row. The rightmost column computes h_t, which is sent to the columns for the subsequent iteration. Their chip provides much higher energy efficiency than FPGA designs and other ASICs such as TPU. The performance of their chip is lower than that of TPU because of the different feature sizes (65nm vs 28nm).

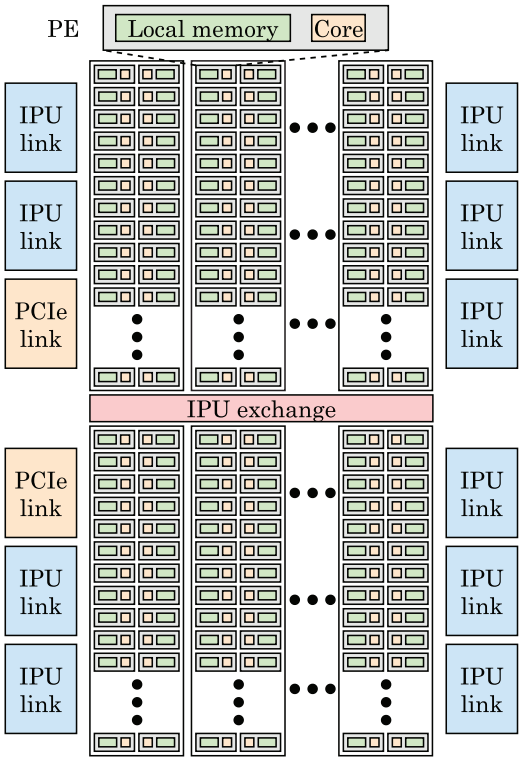


Fig. 13. The architecture of IPU from Graphcore [27].

Table 4
Architectural characteristics of different TPU versions and Volta GPU [2].

Feature	TPUv1	TPUv2	TPUv3	Volta
Peak TFLOPS/chip	92 (8b int)	46(16b)/3(32b)	123(16b)/4(32b)	125(16b)/16(32b)
Maximum chips/supercomputer	NA	256	1024	Varies
Peak PFLOPS/supercomputer	NA	11.8	126	Varies
TDP (Watt/chip)	75	280	450	450
TDP (Watt/supercomputer)	NA	124,000	594,000	Varies
Die size (mm^2)	< 331	< 611	< 648	815
Clock rate (MHz)	700	700	940	1530
Chip technology	28nm	> 12nm	> 12nm	12nm
On-chip/off-chip memory size	28MiB/8GiB	32MiB/16MiB	32MiB/32GiB	36MiB/32GiB
Memory bandwidth (GB/s/chip)	34	700	900	900
MXU/core and MXU size	1 and 256x256	1 and 128x128	2 and 128x128	8 and 4x4
Cores/chip and chips/host	1 and 4	2 and 4	2 and 8	80 and 8/16

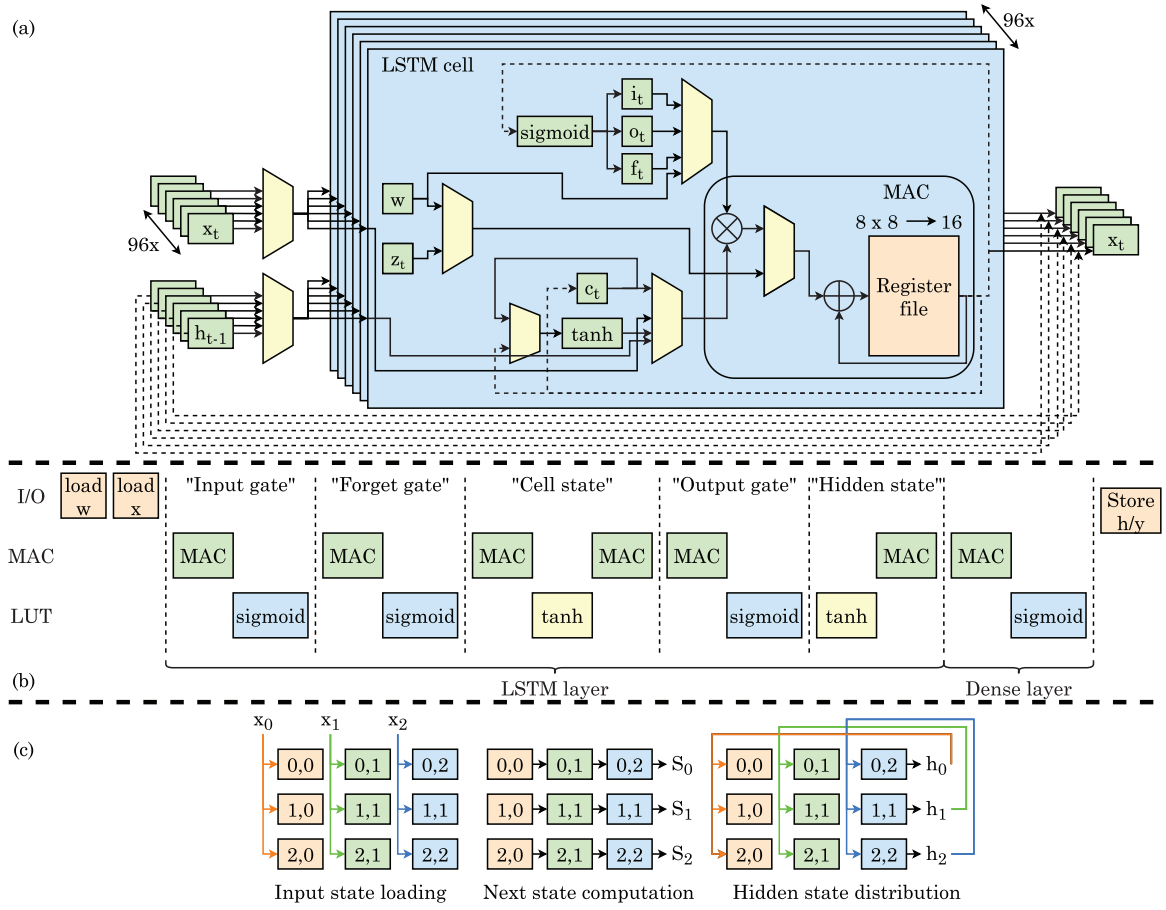


Fig. 14. (a) Block diagram of chip proposed by Conti et al. [30] (b) Sequence of operations (c) Working of a 3 × 3 "systolic array" while loading the input state x_t, calculation of the latest i_t, o_t, f_t, c_t, h_t state values, and distribution of the latest h_t.

Liu et al. [72] (2019) present a cloud-based FPGA accelerator for LSTM networks. Here, a host sends LSTM inference requests to the FPGA accelerator in the cloud. Although the dominant computation pattern of LSTM is vector operations, vector-operations lead to low throughput, which is especially unacceptable in a cloud scenario. They propose concatenating multiple input vectors from different "input sequences" to form an "input matrix". Grouping input vectors into matrices improves bandwidth utilization. This converts the LSTM operations into matrix operations, such as matrix-matrix multiplication and addition, EW matrix multiplication and EW matrix AF. In summary, there are five distinct operations: MAC, +, *, tanh and sigmoid. They propose a dataflow architecture for handling all these operations, as shown in Fig. 15.

Here, the number of PEs equals the LSTM size, which is decided by the dimension of the weight matrix. MatrixA is the input matrix. MatrixB is the weight matrix in MMM or the input matrix in gate computations and in EW computations. For executing matrix computations, the loop is unrolled across the rows of the MatrixB or the columns of MatrixA. Each PE accesses one value from a column of MatrixA or a row of MatrixB. For MMM, the previous PE also needs to write the output to the next PE for performing MAC operation. Different PEs work in a pipelined manner and process different input sequences. A single PE also processes multiple input sequences. All PEs work on various phases of the operation for different input sequences. MatrixA and MatrixB are divided across columns and rows, respectively. They are stored in LUTRAM in the FPGA. This allows PEs to work without stalling due to dependencies. Reception of input on FPGA, FPGA computations and dispatching of the output to the host happen in a pipelined manner. This hides the transmission latency completely. Their technique achieves high performance.

3.5. Architectures for multi-FPGA systems

Nurvitadhi et al. [44] (2019) present a multi-FPGA architecture for accelerating neural machine translation. They connect FPGAs to CPUs through the PCIe slots to implement a software-programmable overlay. CPU and FPGAs are programmed in C/C++ using a unified software framework. CPU kernels are optimized using vectorization with AVX-512. They implement Brainwave NPE [3,80] on the FPGA. For programming the FPGA, the NPE program is compiled into a binary and loaded into the NPE using a software API. They also use a tool for modeling the system-level performance. Using this, the number and types of FPGAs required for meeting the performance constraint can be found. They execute only FPGA-friendly functions on the FPGA and run the remaining irregular computations on the CPU. The model parameters are stored in a distributed manner across on-chip memories of 8 FPGAs. This allows the FPGA to leverage high on-chip memory bandwidth for performing matrix operations. PCIe communication between CPU and FPGA is required only for transferring the vectors. CPU uses FP32, whereas FPGA uses INT8 for MMM and INT27 for vector operations.

They note that the latency of data transfer to one, two and four FPGAs is nearly the same since every FPGA has its own PCIe connection. For vector sizes up to 32KB, data-transfer to/from four FPGAs incurs below 8μs latency, which is only up to 1.4 × the latency of data transfer to a single FPGA. They study three strategies for partitioning the matrices on multiple FPGAs: (1) Splitting the matrix by row blocks and concatenating the partial results (2) splitting the matrix by column blocks and reducing the partial results and (3) a mix of both approaches. Assuming the number of row blocks and column blocks are R and C (respectively), a reduction is not required for an R × 1 configuration, but a 1 × C con-

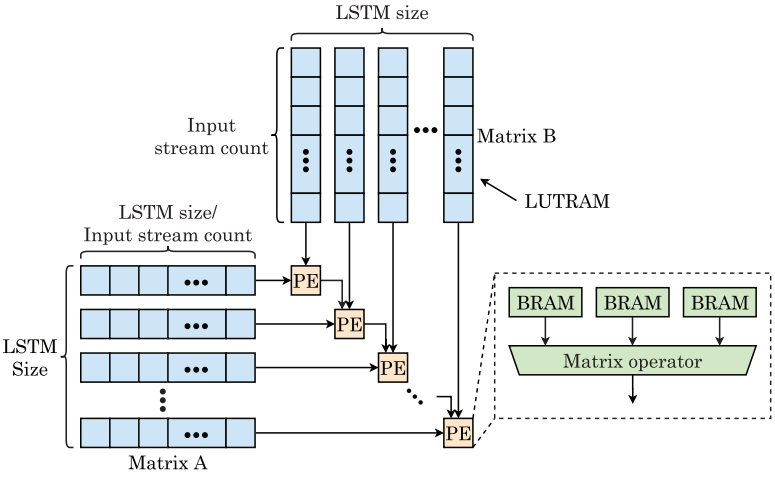


Fig. 15. Architecture proposed by Liu et al. [72].

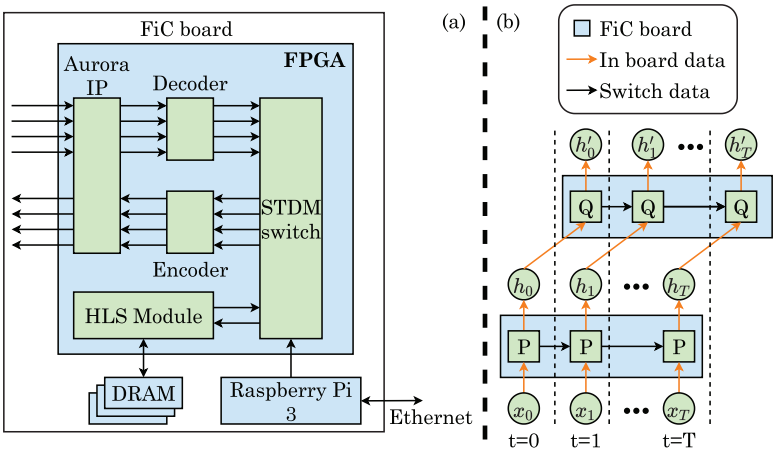


Fig. 16. (a) Block diagram of a "flow-in-cloud" (FiC) board in the technique of Sun et al. [5] (b) Two-layer stacked RNN implementation using two boards.

figuration uses column-block strategy and as such, needs (C-1) reductions on CPU. Overall, data-transfer and CPU computations take a large fraction of time, whereas FPGA matrix operations consume negligible latency. Since the row-blocking strategy does not need CPU reductions, it incurs lower latency. Column-blocking is useful for skewed matrices. With increasing problem size and FPGA-count, data-transfer latency becomes a smaller fraction of overall latency.

For processing LSTM, they execute an entire LSTM subset on a separate FPGA and use CPU to combine the results at the end of every iteration. Their implementation scales well with increasing number of FPGAs and achieves below-2ms latency even with the largest LSTM on 8 FPGAs. They finally implement NMT on their platform, such that LSTM and matrix-vector operations are mapped to FPGA and other operations such as vector operations, AFs, softmax and input/output embedding are mapped to CPU. This application involves a complex pattern of communication and executes many kernels on the CPU. For an NMT model with 100M parameters that processes 50-step input sequences on 8 FPGAs, their technique achieves below-10ms latency. Upon scaling from 1 to 8 FPGAs, the model size increases by 8×, yet the latency increases only by 2×. This is because the FPGA computation time stays constant, whereas CPU computation time increases linearly and the communication time increases sub-linearly.

Sun et al. [5] (2019) present a multi-FPGA architecture for accelerating deep RNNs. Here, multiple medium-scale FPGAs are connected with high-bandwidth serial links. FPGA boards have "static time division multiplexing" (STDM) switches. A switch has three ports: one for accepting the input data and two for sending the output data to other boards. Every layer is mapped to a "flow-in-cloud" (FiC) board. An HLS tool is used for implementing the RNN cells with arbitrary-precision fixed-point numbers. The eight MVM operations of an LSTM cell are performed parallelly by eight MVM modules of the FPGA. The block diagram of a FiC board is shown in Fig. 16(a). Fig. 16(b) shows the pattern for processing a two-layer stacked RNN. Here x_t is a vector, and P and Q are LSTM cells. Initially, the first layer is computed. When the vector of hidden states h_0 is computed, it is sent to the next FiC board through the switch. Simultaneously, h_0 and x_1 are used for computing h_1, the next hidden state vector. Thus, the computation of h'_{t-1} and h_t happens almost simultaneously, barring the packet processing latency in the switch. By scaling the number of FiC boards, any number of layers can be stacked while achieving near-constant execution time.

Their technique can also accelerate bidirectional RNNs, since the FWP and BWP layers can execute independently on separate boards. By stacking them, a deep bidirectional RNN can be designed. An FC layer is mapped to another board, which receives the FWP and BWP states from two other boards. With an increase in the number of layers, the latency of their technique increases only slightly. Their technique achieves high performance and energy efficiency.

Note that Nurvitadhi et al. [44] map an entire LSTM subset on each FPGA, whereas Sun et al. [5] map different layers to different FPGAs.

3.6. Architectures for bi-directional LSTMs

Rybalkin et al. [38] (2017) present an accelerator for bi-directional LSTM with the "connectionist temporal classification" algorithm. They store weights and inputs in 5b fixed-point, internal state and interim re-

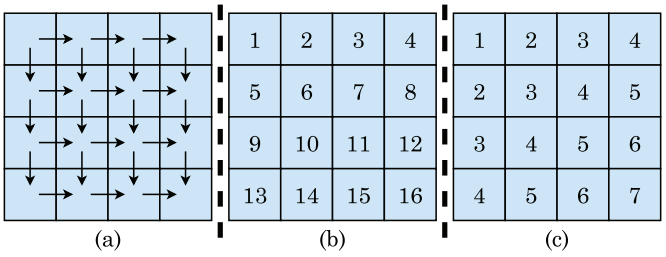


Fig. 17. (a) Pixel dependencies in a 2D-LSTM [74] (b) pixel wise processing order (c) Diagonal wise processing order.

sults in 16b, except that 32b is used for the softmax function. Only one LSTM cell is instantiated to reduce the resource usage and unrolling is done for all multiplication units. Thus, different cells are processed sequentially, but all the inputs to a cell are parallel computed. On applying pipelining, in each cycle, one neuron is processed.

However, in this approach, the pipeline stays idle between the time it accepts input of a memory cell and the time of completion of its processing. To mitigate this challenge, they interleave the processing of forward and backward columns. Starting with the first column, the forward computation of (N-1)th neuron is done, and then, backward computation of 0th neuron is done. Finally, when (N-1)th backward neuron enters the pipeline, 0th forward neuron is supplied with data from the last timestep. At this point, a new image column enters the pipeline. Thus, the pipeline is fully utilized. With this optimization, the memory bandwidth requirement is only half of that with the duplicate datapath, although the weight memory size is not reduced. The throughput remains one neuron per clock cycle.

The computation of the output layer begins when the first N outputs from the forward hidden layer are available. This avoids the need for large buffers. Also, one neuron of the output layer is instantiated with N multiplication units and a reduction tree. This approach halves the latency and the requirement of multiplication units. They also interleave "initial writes" and "later reads" of the values from the column, which reduces the memory requirement by half. Specifically, after writing the last value of the central column from the backward hidden layer, their technique stops writing and begins reading the values. Also, it adds these values with the new outcomes from the output layer corresponding to the forward hidden layer values from the central column.

When no batching is used, they propose parallel execution of different neuron functions on different compute units. When batching is used, different compute units execute different images in parallel. They evaluate their technique on an FPGA. On optical character recognition problem, their technique achieves orders of magnitude higher throughput than a contemporary technique.

3.7. *Architectures for* n-*dimensional LSTMs*

Puigcerver et al. [6] (2017) show that an *n*D-LSTM is between 3X to 11X smaller than an isoaccuracy CNN. However, the number of computations in a single layer of *n*D-LSTM may be one or two orders of magnitude more than that in a CONV layer. Thus, the complexity of an *n*D-LSTM comes from its long sequential execution. Fig. 17(a) shows the dependencies between different pixels. A pixel is dependent on both the previous column and the previous row. To meet these dependencies, the processing can be performed in either row-major order (Fig. 17(b)) or in the diagonal-wise order (Fig. 17(c)). Another challenge in *n*D-LSTM is that multidimensional recurrences prohibit parallelization and slow-down the training and inference speed, even on GPUs. Also, back-propagation over recurrent connections increases the memory requirement.

Rybalkin et al. [74] (2020) leverage the flexibility of FPGA in designing custom datapaths and supporting low bit-width operations for developing an accelerator for 2D-LSTM. They evaluate their technique using two benchmarks: handwritten-digit recognition on MNIST and historical document image binarization. The topologies used for these benchmarks are shown in Fig. 18. The authors note that in 2D-LSTM, computations performed in four directions are independent. As such, they propose interleaving the computations of these directions. In the beginning, the first pixel from every direction is processed, then the second pixel and so on. This scheme avoids the idling of pipelining hardware as long as a sufficient number of sequentially computed LSTM cells are available. Since different directions are sequentially processed, the hardware overhead does not increase by $4\times$. Their technique performs parallelization at the level of LSTM cells and across the dot-product operations for both input data and recurrent state. These unrolling factors are shown as PE_LSTM, SIMD_INPUT and SIMD_RECURRENT, respectively. Similarly, unrolling is done for the FC layer. Also, they quantize weights and activations.

Their proposed accelerator is shown in Fig. 19. Here, the Mem2Stream unit performs DMA reads of input data and converts the transaction into the AXI stream. The pixels are quantized so that the bus width is a multiple of pixel width. Also, pixels are concatenated to the size of a single bus transaction. The width converter divides a transaction into different pixels and concatenates them. The data width converters in recurrent path change the width of hidden-layer output from PE_LSTM to SIMD_RECURRENT. Buffers are used for matching the inputs and outputs. The output layer implements the FC layer with unrolling factors chosen to match the throughput of the previous layer. For handwritten-digit recognition, the outputs of different directions need not be matched for a pixel. Hence, the outputs are simply accumulated in a one-element buffer for every output unit. By contrast, for image binarization, outputs from different directions for the same pixel need to be matched. Hence, a large-enough buffer is used for every output unit. These buffers have separate read-write ports that allow reading/writing partial sums in a pipelined style. Since processing happens in pixel-wise manner and different directions are sequentially processed, there is no contention between read and write addresses in a cycle. Their design provides higher accuracy than contemporary FPGA accelerators. Also, it provides higher energy efficiency and lower latency than a GPU implementation.

4. Techniques for optimizing RNN accelerators

In this section, we discuss pipelining (Section 4.1), parallelization (Section 4.2), batching (Section 4.3) and scheduling (Section 4.4). We also review techniques that repeat computations for saving memory (Section 4.5) and that seek to choose the optimal data-layout (Section 4.6).

Table 5 shows the compute-related optimization techniques used by various works. The table shows the strategy used for implementing the activation function (AF). The AF can be implemented using LUTs or CORDIC units [98]. Since $tanh(x) = 2sigmoid(2x)1$, a single LUT may also be used for both these AFs. However, this approach incurs a high area and latency overheads. As an alternative, AF can be realized using piecewise linear functions which have low implementation overhead. Also, the loss of accuracy can be minimized by choosing a large enough number of segments. Table 6 shows the memory-related optimizations used by various works.

4.1. *Pipelining techniques*

Sicheng Li et al. [20] (2015) accelerate an RNN for language model on an FPGA. Since MVM and AF take the largest fraction of compute-time, they focus on accelerating these functions. The technique of Boxun Li et al. [99] (2014) divides the computation of the FWP phase into two phases: "input to hidden layer" and "hidden to output layer". This is shown in Fig. 20(a). Also, it unfolds RNN in the time domain by pipelining the computations of Q (e.g., $Q = 2$) previous timesteps. The dataflow of this pipelined structure is shown in Fig. 20(b). Sicheng Li

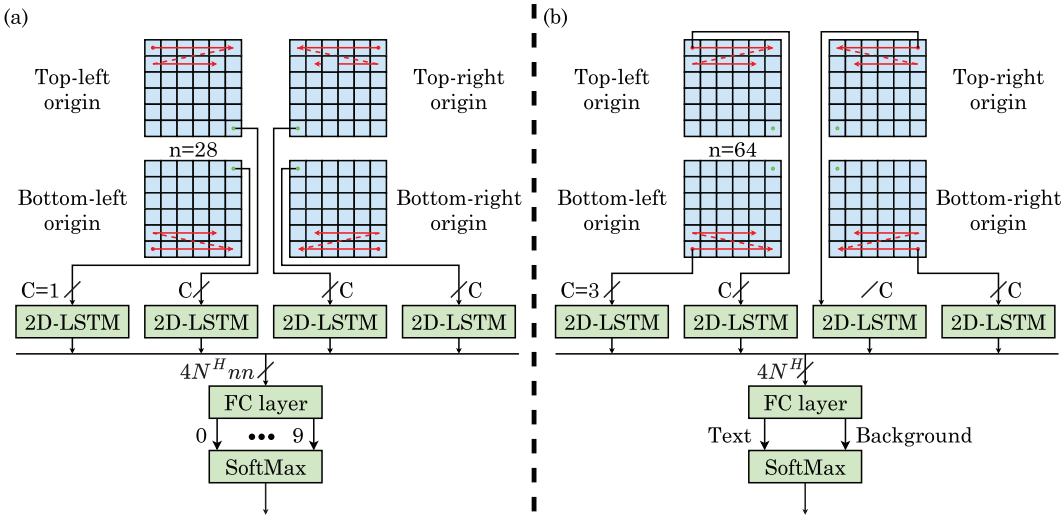


Fig. 18. Topology proposed by Rybalkin et al. [74] for (a) image classification and (b) image binarization.

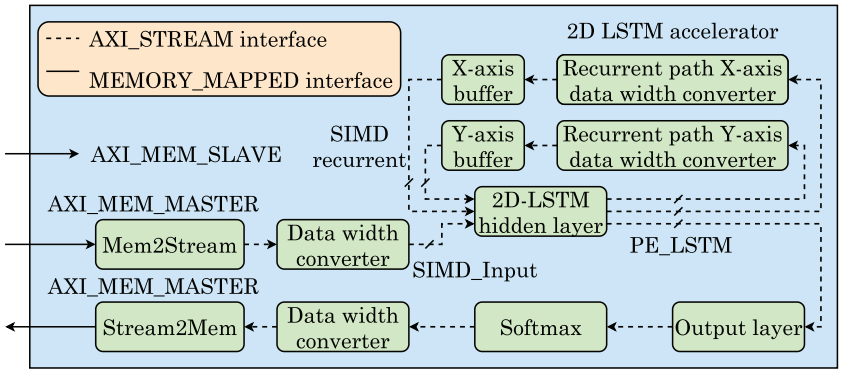


Fig. 19. Accelerator proposed by Rybalkin et al. [74].

et al. [20] note that the throughput of these two stages is not balanced. Let the number of nodes in the input and hidden layer of a RNN be X and that in the hidden layer be H. Then, the computation complexity of the first and second stages is $O(H \times H)$ and $O(H \times X)$, respectively. The value of X ranges between 10,000 to 200,000 depending on the vocabulary size, whereas that of H ranges between 100 to 1000. As such, the second stage has much higher latency than the first stage, leading to an unbalanced pipeline.

Sicheng Li et al. [20] focus on accelerating the second stage of the pipeline. They compute the hidden layer in a serial manner and the output layer in a parallel manner. The proposed architecture is shown in Fig. 20(c). Assuming that the degree of parallelism (Q) is 4, X=10,000 and H=100, their technique provides 3.86 times speedup over the de-

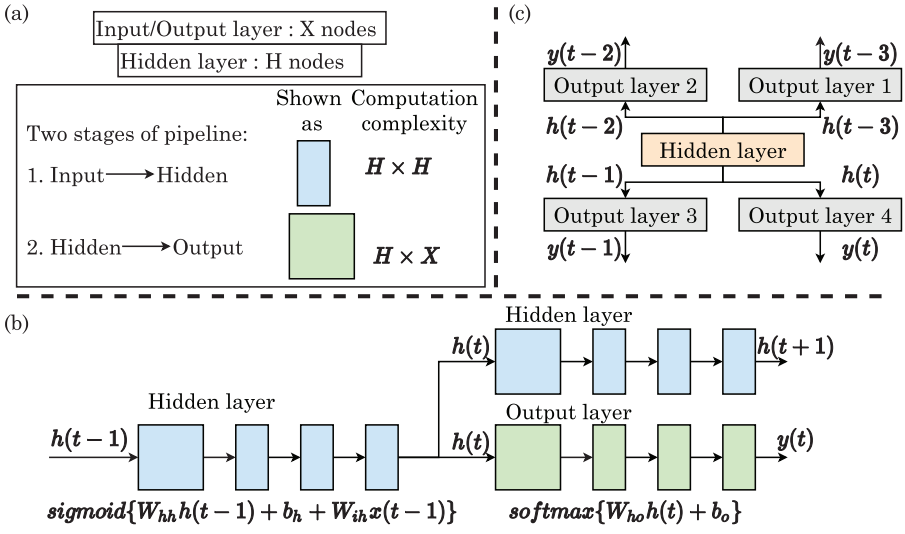


Fig. 20. (a) Stages of the pipeline and the corresponding complexity (b) Dataflow of the two-stage pipelined RNN [99] (c) RNN architecture proposed by Sicheng Li et al. [20] with parallel computation of the output layer (Q=4).

sign of Li et al. [99]. An increase in Q improves the acceleration and allows processing more time steps in each iteration. This can also increase accuracy by sustaining a higher amount of historic information. A downside of this approach is that it increases the hardware resource requirement. Based on the error tolerance of RNNs, Li et al. [20] propose lowering the precision of W_{ho}, the weight matrix between hidden and output layers. Specifically, on storing W_{ho} with 16 bits while keeping the hidden and output layers in 64 bits, the output quality of a fixed-point implementation becomes same as that of a floating-point design.

Their FPGA-based accelerator has two types of computation cores: one for hidden layers and one for output layers. Except for the AF, these computation cores are identical. Every computation core is composed of multiple PEs. In a MVM, each PE handles a row of the weight matrix. They note that the amount of data required for controlling the RNN training is minimal. Hence, they store the training data in CPU memory and transfer it to FPGA in each training step. They also propose techniques for balancing the memory access load across multiple PEs. They store weight matrices and bias vector in separate buffers to enable their reuse, which reduces the memory access overhead.

On the sentence completion workload, FPGA-based implementation of their technique achieves higher training accuracy than the conventional class-based recurrent network. Also, it provides much higher performance than a "math kernel library" (MKL)-based implementation on a Xeon E5-2630 CPU. For small RNNs, their FPGA implementation provides higher performance than a GeForce GTX580 GPU implementation. This is because the GPU is bottlenecked by the divergence due to branch-prediction [100]. Also, GPU implements AFs exactly, whereas FPGA implements them in approximate manner which is more efficient. However, for large-scale RNNs, their FPGA implementation achieves lower performance than the GPU implementation because of limited memory bandwidth of FPGA.

Que et al. [78] (2019) note that in conventional MVM implementations, at any time, computation is performed on the whole vector of (x_t, h_{t-1}) and entire row of weights. This, however, leads to stalls since the new hidden vectors need to be computed before beginning the subsequent timestep to respect the data dependency. Thus, new MVM can begin only after the pipeline has become empty. This approach is shown in Fig. 21(a). In their proposed technique, in the beginning, only Q elements of x_t vector are used (where Q is decided by the amount of parallelism), whereas h_{t-1} is not used. As shown in Fig. 21(b), computations are done on all elements in the corresponding columns of the weight matrix. Thus, computation of (x_{t+1}, h_t) can begin without requiring emptying of the pipeline for obtaining h_t. This removes pipeline stalls.

Further, they split the "weight matrix" into multiple tiles and batch the input activations, while accounting for data-dependencies. This allows the tiling of computations, which addresses the issue of limited on-chip memory. Batching allows reuse of weights for the next MVM computations and it also reduces the number of memory accesses. Batch size is chosen so that computation time exceeds the communication time. In their technique, the parameters of all types of gates are included in the matrix. Tiling is done in a manner that each tile also includes the parameters of all types of gates.

Their computing engine is shown in Fig. 21(c). In each iteration, a single tile is transmitted to buffer0 and buffer1 in the on-chip memory, which work as the "double-buffers". These buffers also store the chunk of activations of input vectors. The PEs perform MVM between the weight matrix and partial input vector. The partial results are accumulated through the inter-tile linking and finally stored into a FIFO for use in the next tile. Thus, this FIFO allows accumulating the results of different tiles. The length of this FIFO equals the batch-size. After processing of all blocks, the final result is produced on the "interconnection unit" where it is reshaped for subsequent processing. Remaining computations such as AFs are performed in the post-processing unit. Their technique scores high on performance per watt metric.

4.2. Parallelization techniques

Zhang et al. [61] (2018) note that MVM kernel (namely Sgemv) accounts for a large fraction of LSTM latency due to the large number of off-chip accesses. For each cell's execution, the MVM kernel needs to be launched, which operates on the weight matrix. The MVM kernel at the current cell operates on h_{t-1}, which depends on the previous cell in the same layer. Due to this, Due to the inter-cell dependencies, MVM kernels across the cells cannot be merged into one MMM kernel. For MMM kernel, memory loading is required only once per layer, whereas for MVM kernel, it is required for every cell's execution. Hence, the amount of loaded data could be 100 × that of the original data.

They propose techniques to mitigate memory bottlenecks in RNN execution on mobile GPUs. They note that in tanh and sigmoid functions, the output changes linearly with input in the range [-2, 2], whereas for the remaining values of input, the output is nearly independent of the

Table 5
Compute-related optimizations.

Category	Reference
Pipelining	[1,3,5,7,10,11,13,14,19,20,29,29,33,34,34,36,38,40,43,45–47,49,51–53,55–57,59,60,65,67,69,70,72,73,78,80,85,99,99]
Enabling parallelization by breaking dependencies	By exploiting sensitive/insensitive ranges of AF [61], by exploiting correlation information to speculatively parallelize input sequences [29]
Increasing parallelism	Fusing the cells from the sub-layers into cell-groups [61], fusing multiple MMMs into one [15], concatenating the inputs for multiple time-steps [15], using CUDA streams to simultaneously execute multiple MMMs [15], parallely accessing different banks of ShM [62]
Batching	[1,3,13,15,16,19,21–24,28,32,34,36,38–40,43–45,47,50–52,52,57,59,60,60–64,69,74,78,85]
Memoizing	Transpose of weight matrix [15], output of a neuron [11], results of multiplications of feed-forward weights with all the word encodings in the vocabulary [16]
Heuristics/tools	Roofline model [1,42,78], graph coloring [39], greedy algorithm [60], JIT compilation [25]
Techniques for reducing computations	
Reducing number or overhead of multiplications	Logarithmic number system for converting multiply into addition [84], replace multiplications with shift operations [32], approximate multiplier [36,68]
CNN architecture-level techniques	Use of ReLU AF [7,48], FFT CONV [70,71,73], singular value decomposition [42], knowledge distillation [7]
Value-based techniques	Skipping the computation of $W_x x_t$ whenever $W_h h_{t-1}$ is highly negative [7]
Realizing AF using	
Piece-wise linear approximation	[10,20,29,36,41,45,46,53,55,67,70,78,84]
Piece-wise polynomial approximation	[72,89,90]
LUT	[5,30,38,49,69,75]
Exponential computations	[85]

Table 6
Memory-related optimizations.

Category	Reference
GPU memory optimizations	Storing RNN parameters in GPU RF [25,57,60] and activations in ShM [57,60], avoiding ShM bank-conflicts [60], transferring intermediate data to host memory during training [21], distributed training [57], use of CUDA stream [15,59]
Reusing memory	Reclaiming the memory allocated to hidden state and cell state and reusing it for the next cell [58], reclaiming and reusing the memory for subsequent timesteps [24,60], increasing weight similarity and reuse using quantization [86]
Double-buffering	[1,7,39–41,43,46,67,72,76,78,78]
Extra computations to save memory	[21,24,27,67]
Loop-optimizations	Tiling [3,10,39,42,44,46,54,76,78], loop unrolling [46,54,72]
Other memory optimizations	Sharing of weights between decoding processes in the beam-search algorithm [14], sharing exponent across 128 sign and mantissa values [3], near-memory computing [80]
Changing data-layout or storage locations	
Choosing optimal layout	Using row-major or column-major layout to improve cache efficiency in GPU [24], using channel-major layout for avoiding data-duplication and achieving bandwidth efficiency [39]
Choosing storage location	Changing the locations of NZ values for reducing shared-memory conflicts in GPU [60], storing the weights of different gates in consecutive locations [85], reshaping weight matrices to achieve contiguous access with tiling [46], interleaving multiple accesses to reduce bandwidth wastage [76]
Adaptive techniques	
Choosing architectural optimization	Using double-buffering for inputs and intermediate results but not for weights (due to their larger size) [73], adapting the tile-size in different rows [85], using different datatypes and RFs for vectors and matrices [3]
Choosing memory technology	designing input/hidden buffer using SRAM and weight buffer using eDRAM [85]
Choosing RNN cell	choosing between LSTM or GRU based on accuracy constraint [73]

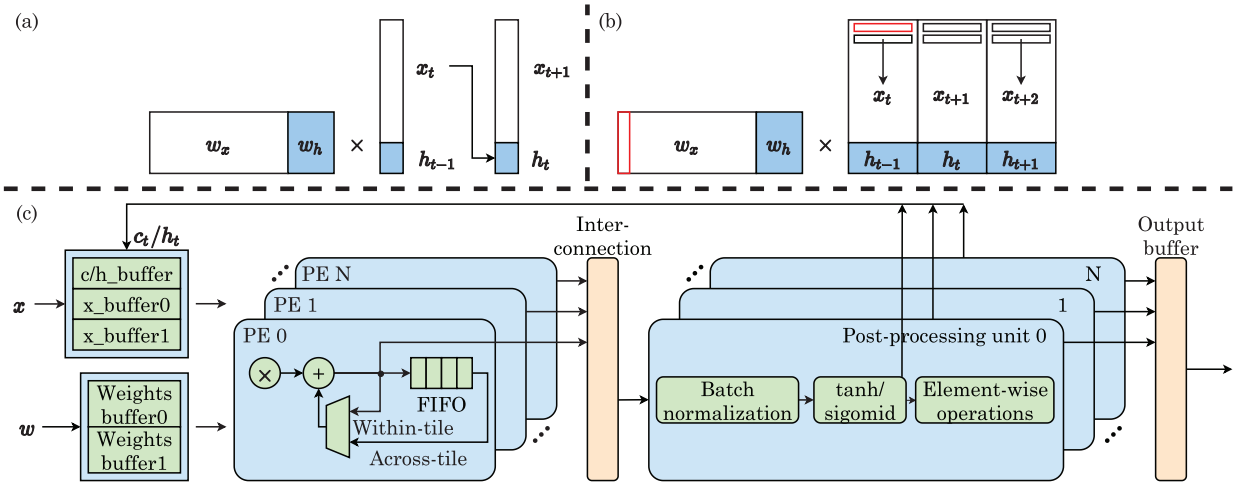


Fig. 21. (a) In the conventional method, there is data dependency between successive time-steps (b) The technique of Que et al. [78] removes this dependency and allows operation of pipeline without stall (c) Architecture of the proposed computing engine [78].

input. These ranges are termed sensitive and insensitive ranges, respectively, as shown in Fig. 22. Based on this observation and the range of the output of tanh and sigmoid, they find the conditions under which h_{t-1} has no impact on the value of f_t, i_t, c_t, and o_t, or in other words, the output of the previous cell has no impact on the computation of the current cell.

Based on the knowledge as to where there is no (or only weak) dependency between the cells, they divide an LSTM layer into multiple independent sub-layers. Fig. 23(a) shows 8 cells with both strong and weak dependencies. The execution of these sub-layers is parallelized. Since complete removal of weak dependency links between sub-layers may lead to a large loss of accuracy, they add a predicted link to the first cell of every sub-layer for recovering the accuracy, as shown in Fig. 23(b). The boundaries of sub-layers are chosen by breaking those links whose impact-score is lower than a threshold. The sub-layers are parallelized by fusing the cells from the sub-layers into cell-groups. Each cell-group has one cell from each layer. For instance, in Fig. 23(c), the LSTM layer is split into 4 sub-layers, organized into 3 cell-groups. Let S be the number of cells in a group. They note that on increasing S, the performance first increases and subsequently decreases due to the limited ShM bandwidth of mobile GPU. The bandwidth utilization is 100% at $S = S_{opt}$. At $S > S_{opt}$, kernel reconfiguration is required for not exceeding the bandwidth limit, which changes the thread-count so that execution time is increased. This increase is not offset by a reduction in memory accesses due to an increased S value. To mitigate this issue, their technique ensures that for different groups, S value is nearly the same and no group has more than S_{opt} cells, as shown in Fig. 23(d). The value of S_{opt} is found using offline execution.

They further observe that in every cell, the largest memory consumption comes from the matrix $U_{f,i,c}$. The output gate o_t has a strong influence on h_t, so a near-zero element in o_t makes the corresponding element in h_t near-zero, regardless of the value of element in c_t. Hence, the corresponding rows in U_f, U_i and U_c matrices that determine c_t have no impact on h_t. Based on it, they propose not loading these unimportant rows, which also avoids any computations on them. This technique

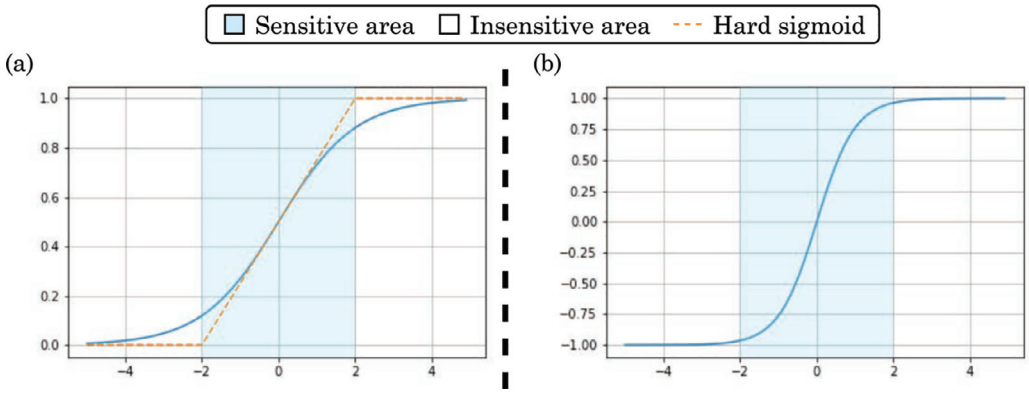


Fig. 22. Illustration of sensitive and in-sensitive regions in (a) sigmoid AF and (b) tanh AF [61].

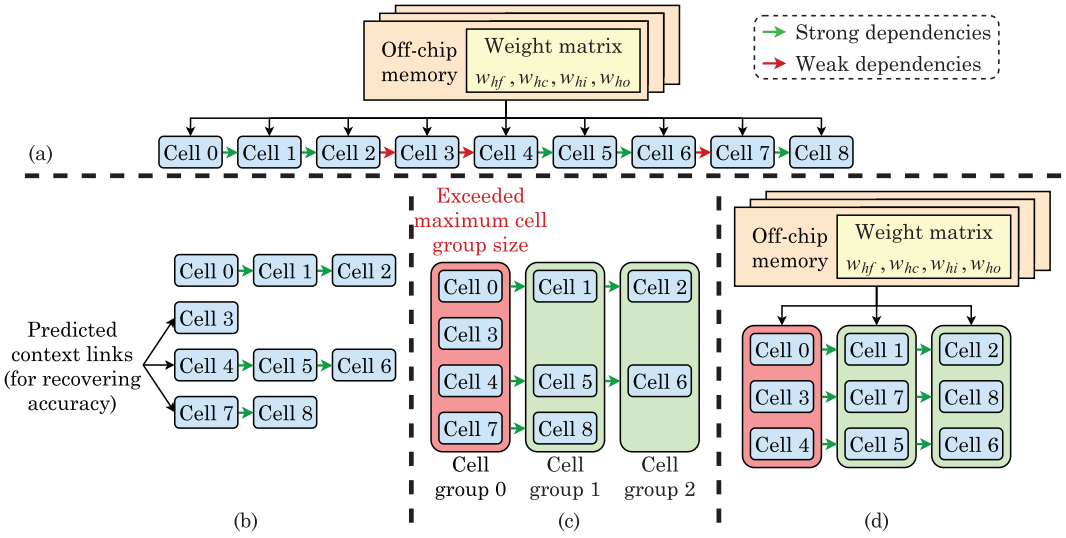


Fig. 23. (a) Sub-layers with strong and weak dependencies [61] (b) predicted context links (c) formation of cell-groups (d) Alignment of cell-groups.

makes some values of the state vector c_t close to zero, which has a negligible impact on the accuracy since the subsequent cell uses forget gate for filtering the state vector from the previous cell. This technique is applied at runtime since different rows are skipped in different LSTM cells. They also propose techniques for handling the branch divergence resulting from row-skipping. Their technique provides a significant improvement in performance and energy-saving with only minor accuracy loss.

U Ouyang et al. [29] (2017) present a technique for parallelizing LSTM-RNN computations. In RNN, H_t is used for computing the index into the vocabulary which stores category items such as words of a sentence. For sentence completion, the RNN may output a predicted word stored in the vocabulary corresponding to the index. The index of the current word is used for predicting the index for the next word. This results in a dependency between two iterations. An iteration of RNN can be split into three tasks, viz., generation of input data, processing of core layer and FC layer. These tasks are executed by the data-generation unit (DGU), core-computation unit (CCU) and FC-computation unit (FCU), respectively. Due to the dependency between these tasks, successive iterations have to be executed serially, as shown in Fig. 24(a).

They note that different words have semantic correlation, e.g., if the present word is “He”, the following word is more likely to be “is” than “teach”, “are” or “your”. These likelihoods can be computed from the language model. The correlation between the words translates into correlation between the adjacent indices. Based on this, their technique prefetches indices that show a high semantic correlation. This allows DGU of the next iteration to work in parallel with FCU and CCU of the current iteration. If the next index is the same as that predicted from semantic correlation, then the processing continues, as shown in Fig. 24(b). In case of a wrong prediction, the progress of DGU is discarded and the computation is repeated with the correct index, as shown in Fig. 24(c). Overall, by virtue of achieving highly parallel and pipelined execution, their technique achieves high performance and energy efficiency for natural-language processing, speech-recognition and image captioning tasks.

Appleyard et al. [15] (2016) present techniques for optimizing RNN on GPUs. They note that a naive implementation that implements every function viz., MMM, EW addition, sigmoid and tanh as a separate kernel achieves poor performance. In FWP, LSTM has four MMMs operating on the x_t vector and four MMMs operating on the h_{t-1} vector. They fuse four MMMs into a single MMM, which increases the parallelism. Further, they use CUDA streams to run both the fused MMMs above concurrently. This doubles the available parallelism in each RNN cell and is especially effective for small-sized MMMs. Since all pointwise operations are independent, they are also fused into a single large kernel.

At the level of a layer, inputs for multiple timesteps are concatenated, which leads to larger MMM. Also, since a weight matrix is used repeatedly, transposing it once improves the performance of subsequent computations. Finally, they parallelize the computations of multiple recurrent layers. As for the scheduling scheme, the next work to be scheduled is the one with the least edges to traverse before the first recur-

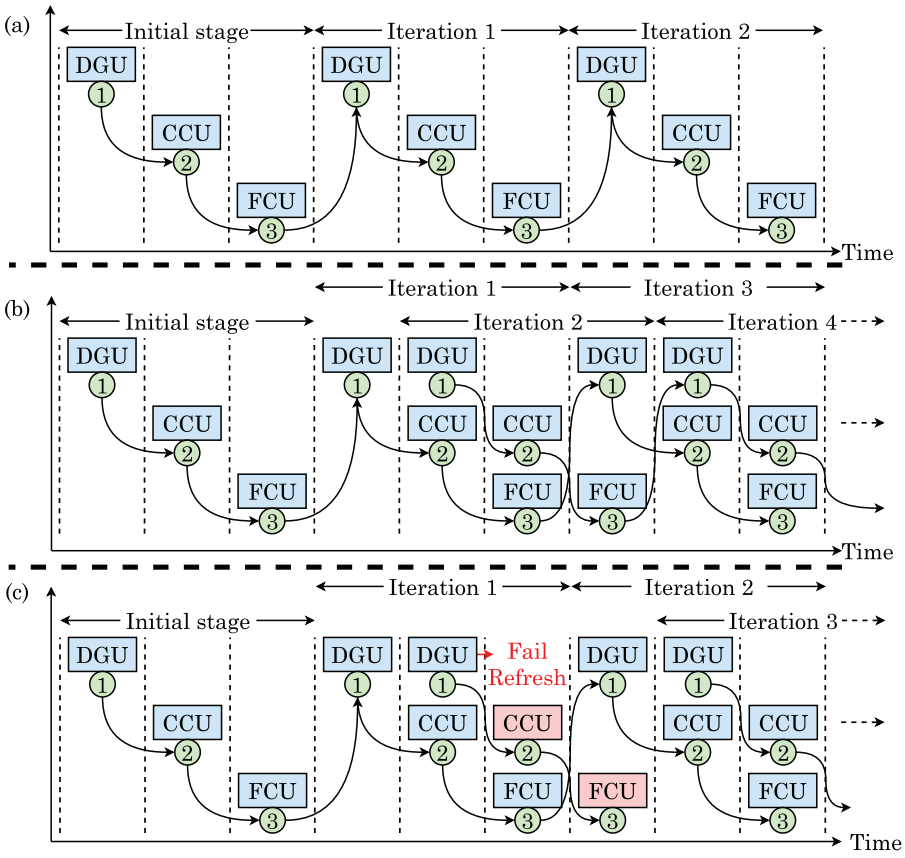


Fig. 24. (a) Baseline serial LSTM processing (b) Semantic correlation based parallel processing [29] (c) Processing in case of failed predictions.

rent cell is reached. Thus, in a 2D organization of cells, the cells that are launched are in a diagonal direction. Their optimizations provide a significant speedup, especially when the hidden state size and/or mini-batch size are small because such networks have a smaller amount of inherent parallelism. These optimizations are incorporated into the fifth version of the cuDNN library.

Peng et al. [79] (2019) note that the use of FPGA resources for designing a single-core accelerator reduces efficiency while handling multiple requests. They propose a multi-core accelerator design where each "core" can handle a separate RNN inference task. Their accelerator can operate in three styles: multi-programming, multithreading and helper-core. (1) In multi-programming style, two cores can process different inference tasks. However, if only a single task is available, one core stays idle. (2) In multithreading style, two cores can simultaneously work on a single inference task. Multithreading style exploits parallelism across LSTM layers and timesteps. Fig. 25(a) shows the dependency diagram and execution sequence of LSTM. Evidently, due to data-dependencies, the speedup from multithreading remains small.

They note that in an LSTM, the sequence length greatly exceeds the number of layers. For example, a 4-layer LSTM may process a sequence of length above 20. Based on this, they propose a scheduling scheme, where the computation tasks of every timestep are grouped into multiple threads. Mapping of threads to cores is done in interleaved fashion and then, scheduling is done while respecting data-dependencies. Specifically, odd/even threads are mapped to core 0/1, respectively. The execution proceeds at the granularity of timeslots: every core advances one step for the current thread in every timeslot. Fig. 25(b) shows the working of this technique for a 3-layer LSTM and a sequence length of 3. Here, core 0 handles threads 1 and 3, whereas core 1 handles thread 2. Execution completes in 6 timeslots and core 1 stays idle in 1st, 5th and 6th timeslot. When the sequence length and hence, the number of threads is large enough, the time period during which core 1 remains idle is a very small fraction of the overall time.

(3) Helper-core style exploits parallelism inside a layer. They observe that the number of computations and the amount of data transferred for MVM is much larger than the EW multiplications. Also, the eight MVMs performed in a layer are mutually independent. The helper-core style decouples the MVMs involving x_t and h_{t-1} and maps them to two cores. The main core and helper-core execute MVMs involving h_{t-1} and x_t, respectively. The helper core moves ahead of the main core and computes MVMs involving x_t, which allows the main core to compute MVMs involving h_{t-1} and EW operations. In every timeslot, a predetermined number of tasks are scheduled for every core. Only when both cores complete a timeslot, they advance to the next timeslot and in this way, the cores are synchronized.

They evaluate their techniques on six benchmarks: "image-captioning", "sentiment classification", "activity recognition", "text generation", "speech recognition", and "translation". The first three are single-layer LSTMs, whereas the last three LSTMs have two or three layers. Multiprogramming (MP) achieves the best performance for all the benchmarks except for the translation benchmark, where helper-mode performs the best. In general, in decreasing order of the amount of time spent in idle mode, the modes are helper-core, multithreading and MP. In MP, two cores independently run the same benchmark and compete for the same memory bus. In image-captioning and translation, memory operations in single-core execution take over 50% of the runtime, so effective memory latency is increased due to the contention at the bus. This reduces the performance of MP on image-captioning and translation.

The poor performance of helper-core mode comes from the fact that assigns all EW operations to the main core, which leads to load-imbalance and synchronization delays between two cores. This imbalance is further aggravated for benchmarks for which x_t vector is smaller than h_t vector. Conversely, when x_t vector is larger than h_t, the imbalance is mitigated. For example, the translation benchmark has the largest layer-size, and the computation-load of MVM increases quadrat-

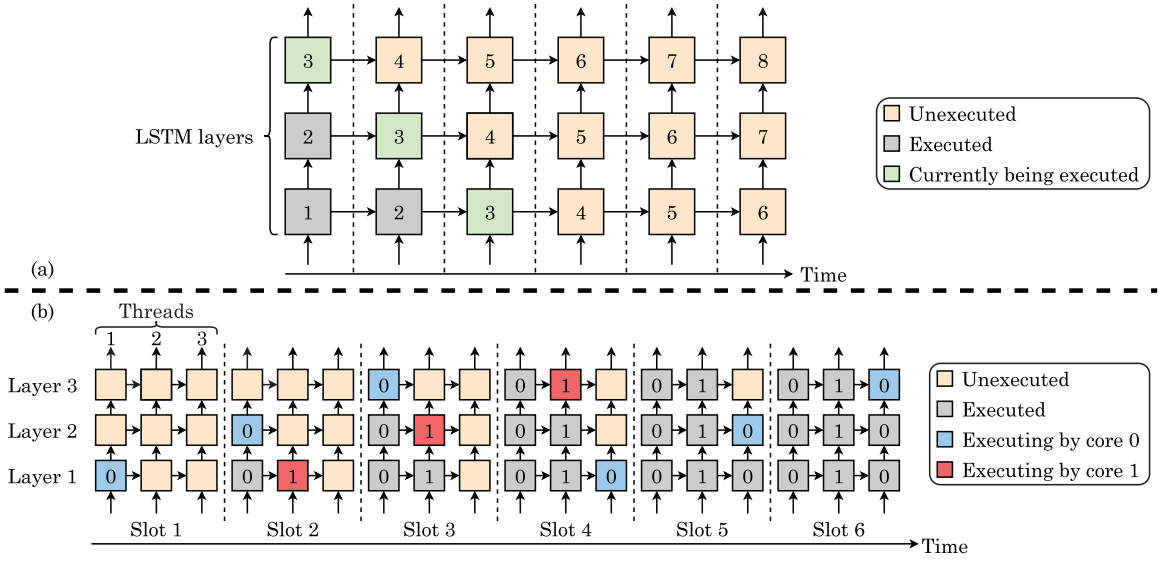


Fig. 25. (a) Execution sequence of LSTM (b) Multi-threaded scheduling [79].

ically, whereas that of EW multiplication increases linearly. Hence, the computation-load of EW multiplication is insignificant in comparison to that of MVM. This reduces the synchronization delays in translation and hence, helper-core mode provides the highest performance on translation. For translation, MP and multithreading are bottlenecked by memory bandwidth, but helper-core mode can reuse the parameters from on-chip buffers.

Nurvitadhi et al. [16] (2016) note that in GRU, a majority of computations are performed in dense-MVMs. GRUs have three computation gates, each of which performs two MVMs in each timestep, one involving feed-forward weights and another involving recurrent weights. Since the weights do not change during inference, they propose memoizing the results of multiplications of feed-forward weights with all the word encodings in the vocabulary. This allows removing half of the MVM operations in GRU and hence, brings close to 50% saving in inference latency. MVMs still account for the majority of the remaining latency and hence, they accelerate it using FPGA.

They divide a matrix into blocks depending on the available resources. Different blocks are processed in sequence, whereas rows and columns of a block are processed in parallel. Their architecture has a memory read and a memory write unit, along with groups of fused multiply-add (FMA) units. A single FMA unit processes a row of a block by computing the product of the input vector elements with row matrix elements. The final accumulated results from two FMAs are added by the reduction unit and this generates the final output vector element (VecOutN). Their technique can exploit fine-grain parallelism in small/medium matrices by virtue of distributing matrix data on multiple on-chip RAMs and carefully moving data to utilize all the FMAs. Due to this, their technique achieves high throughput and energy efficiency.

4.3. Batching techniques

Gao et al. [59] (2018) note that while CNNs have fixed computation patterns, RNNs perform recursive computations. Hence, the "unfolded dataflow graph" of an RNN is not fixed but changes based on the input. Thus, inputs decide the depth of recursion, which makes it difficult to use batching. Previous techniques perform batching of unfolded graphs, which is termed as "graph batching". This approach is shown in Fig. 26(a). It harms latency because different inputs do not come at the same time. Also, for inputs with different sizes, in the combined graph, all operators cannot be fully batched and thus, improvement in throughput remains small.

Since RNN computation is composed of different numbers of similar compute-units, they propose batching at the granularity of subgraphs. Two subgraphs of identical topology are said to be of the same type if their parameter weights are identical and their inputs have the same shape and count. If subgraphs of the same type have no mutual-dependency, their batching can be done. As an input arrives, its computations are broken into multiple subgraphs and the group of common subgraphs are selected for batching, as shown in Fig. 26(b). The batched subgraphs may come from different or the same request. This allows immediate batching of an incoming request with ongoing requests, without waiting for the previous requests to finish. Also, a request is returned to the user as soon as its last subgraph is finished. Thus, batching of a short request with a long request does not delay the short request. They allow a user to express the recurrent architecture of an RNN at the computational granularity of a subgraph. The maximum batch size for different cell-types is different and it is found through offline benchmarking.

On GPU, they use CUDA streams for asynchronously launching the kernels while respecting their data-dependencies. Further, they add a "signaling kernel" to the end of every task. When a task is done, the signaling kernel increments a signal variable. A thread running on the host (CPU) reads this variable and this allows asynchronously notifying the host about completion of a task. Compared to MXNet and TensorFlow, their technique achieves lower latency and higher throughput for Seq2Seq and LSTM models.

4.4. Scheduling and synchronization techniques

Cao et al. [58] (2017) present techniques for optimizing RNN execution on mobile GPUs. They note that the use of a CUDA-like approach in mobile GPU makes RNN execution 4 × slower than the execution on Nexus5 mobile CPU. This is because mobile GPUs have unified (and not discrete) memory and fewer cores than the desktop GPUs [101]. Hence, the workload is factorized into several tiny units, which leads to significant scheduling overhead. Their technique uses the RenderScript framework [102], which allows breaking down the computation in suitable work units that are automatically parallelized on GPU cores. For example, assume that in each gate, a 1x32 input is multiplied with a 32x120 weight matrix, as shown in Fig. 27(a). The CUDA-like approach, shown in Fig. 27(b) may break this computation into 120 units, where every unit multiplies the input with one of the columns. At each time, 12 units are scheduled and there are a total of 120 function calls. By contrast, in the technique of Cao et al., shown in Fig. 27(c), the task is divided into

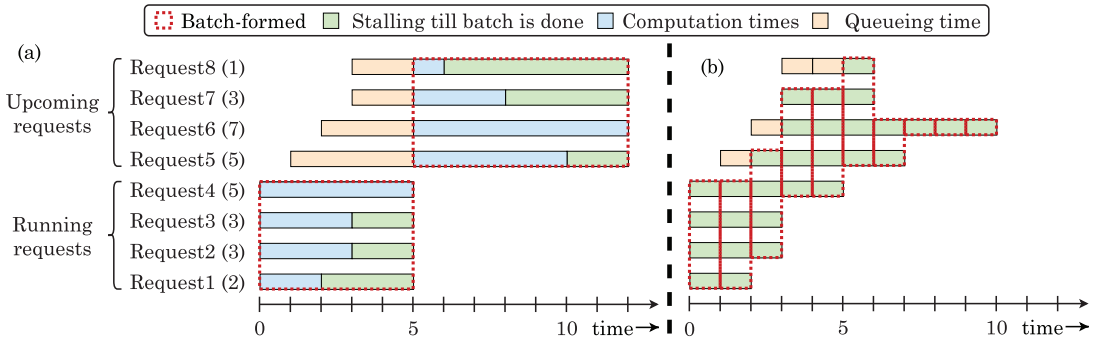


Fig. 26. (a) Graph batching (b) Cellular batching proposed by Gao et al. [59].

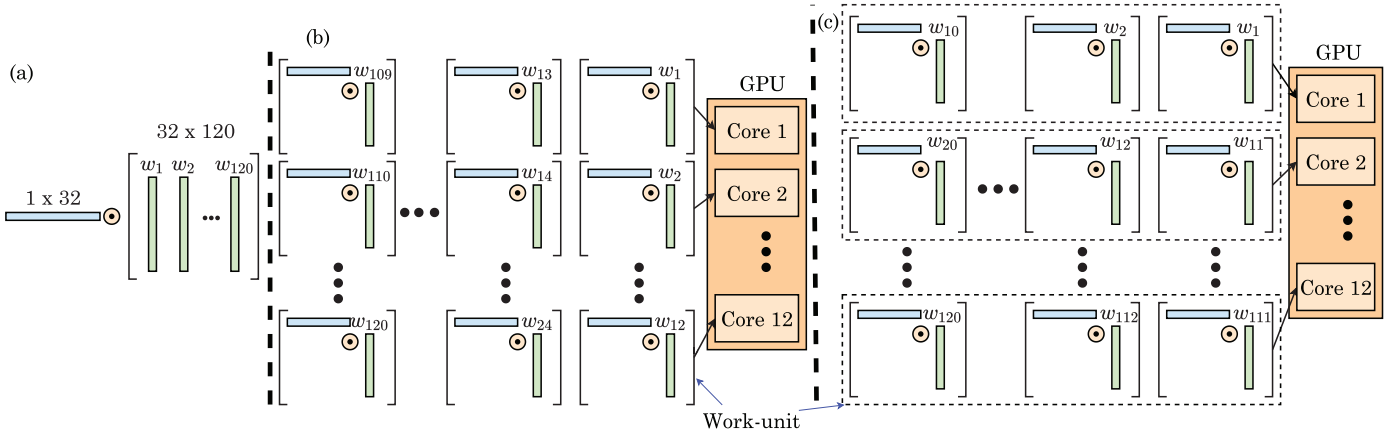


Fig. 27. (a) Vector-matrix multiplication in an LSTM gate (b) GPU computation using CUDA (c) GPU computation approach proposed by Cao et al.[58].

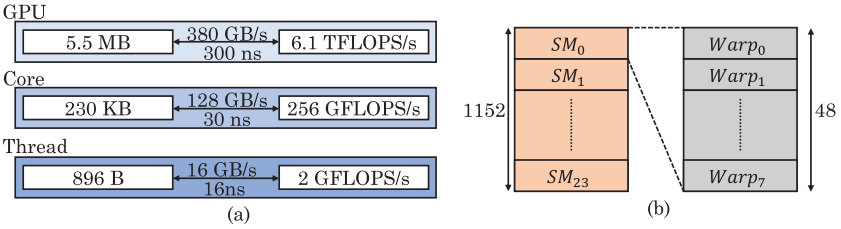


Fig. 28. (a) An abstract view of TitanX GPU microarchitecture (b) Tiling of recurrent weight matrix. Every streaming multiprocessor (SM) computes a 48x1152 block row where it reads 1152 activations for the present time-instance and writes 48 activations for the subsequent time-instance. In each streaming multiprocessor, every warp computes a 6x1152 block row and all 8 warps can access the activations for the current time-instance in ShM. A warp has 32 threads.

12 work-units, each of which computes 10 vector products. This lowers the scheduling overhead.

Their technique also reclaims the previously allotted memory for reusing it, which leads to memory saving. When a cell completes its execution, the memory allocated for the hidden state and cell state is reused for the next cell. They test their technique on an RNN trained for predicting activities based on sensor inputs. Their technique brings 4 × speedup on mobile GPU compared to the single-threaded Nexus5 CPU implementation. When RenderScript is used for doing multi-threading only on CPU, the performance achieved is 70% of the mobile-GPU performance. Finally, with increasing load on GPU, the RNN execution latency increases further.

Diamos et al. [57] (2016) present a technique to improve RNN training performance by storing RNN weights on-chip. They note that due to the serial dependence between time-instances, explicit synchronization is required. Also, the recurrent weight matrix needs to be loaded from memory at every time-instance. To optimize this operation, they use the “multi-bulk-synchronous-parallel” machine approach. This approach models the resources of a GPU, viz., number of cores, bandwidth, synchronization overhead and cache/memory capacity. The GPU is modeled hierarchically, as shown in Fig. 28(a). Then, beginning with the lowest level of the hierarchy and moving upwards, individual neurons are divided into logical modules. Also, modules are connected such that (1) parameters corresponding to the neurons chosen for a processor can be accommodated in the cache or memory for a particular level of memory hierarchy, (2) intra/inter-module communication latency is nearly equal to the compute latency of neurons of a module and (3) synchronization latency due to inter-module connections is nearly equal to the compute latency of neurons of a module. Thus, their technique seeks to balance the cost of computation, data-transfer and synchronization.

They divide the “recurrent weight matrix” into groups of contiguous rows. A single streaming multiprocessor works on a row that obviates the need to perform an inter-streaming-multiprocessor reduction for computing the activations for a group of rows. This approach is demonstrated in Fig. 28(b). The limitation of this approach is that it needs higher global memory bandwidth compared to an approach that divides the matrix into tiles. Their technique loads weights into RF [103] and the input activations from the previous timestep into ShM. After this, the computation is performed for every row, the result is written for the present timestep and a barrier is performed between all streaming multiprocessors. Software pipelining is done to overlap computations with communication.

On TitanX GPU, every thread can access 1KB memory. Out of this, they reserve 896B for storing recurrent weights and the remaining for

the interim computations. These RNN weights are reused over multiple timesteps, which avoids the need to fetch them from DRAM. They further note that achieving synchronization between GPU cores by launching multiple kernels leads to high overhead. This is because the synchronization latency becomes much higher than the compute latency of a single timestep. To mitigate this issue, they optimized the implementation of a global barrier that can be totally overlapped with the computations for a single time-instance. With minor changes, their technique can also be applied to LSTM and GRU based RNNs. Their technique provides high throughput with small mini-batch size (e.g., 4), reduces the requirement of activation memory, and shows good strong-scaling of RNN training for up to 128 GPUs. Overall, their technique can train RNN models with more than 100 layers.

Zhu et al. [60] (2018) extend the technique of Diamos et al. [57](2016) for sparse RNNs. In sparse RNNs, each NZ weight is stored in the format of < index,value > and it is stored on-chip. All NZ elements processed by a thread should belong to the same row. Also, the degree of sparsity and size of the hidden layer determines the number of thread-blocks. This is in contrast to the case of dense RNNs, where the number of thread blocks equals the number of streaming multiprocessors. They note that due to the sparsity, the reuse of activations is reduced. Also, ShM access by threads of a warp can lead to bank conflicts if they are directed to the same banks [104](2019). As such, the performance of sparse RNNs is bottlenecked by the bank-conflicts and ShM throughput.

To mitigate these issues, they propose four strategies. (1) They pad rows with less than the maximum number of NZ weights with < index,0 > values. This ensures that all rows have the same number of weights. This strategy improves performance by avoiding overheads of handling sparse matrix. (2) In order to achieve weight reuse between samples, 4 activations from different samples are batched together. `ld.shared.v4` instruction is used to read these activations from ShM simultaneously. Since the addresses of these activations are contiguous, they are stored in different ShM banks, which reduces bank conflicts. However, this increases the requirement of ShM capacity, which reduces the maximum size of the hidden layer. For the case of large hidden layers, `ld.shared.v2` instruction can be used, which reduces ShM usage at the cost of increased bank-conflicts.

(3) They change the weight-layout to reduce ShM bank-conflicts. Every thread stores K < index,value > pairs and computes `sum+= value[i]* activation_in_SharedMemory[index[i]]`, one by one. They propose changing the locations of NZ values in each row, which creates an access pattern for ShM with reduced bank-conflicts. This reordering is required only in case of change in the sparsity pattern of the network. (4) To ensure correct ordering of work of different thread-blocks, they suggest using either Lamport timestamps or global synchronization. Use of Lamport timestamps provides higher performance, except that for very large layer-sizes, global barriers are better due to lower requirements of ShM. Their optimizations provide significant speedup on pruned RNNs and allow much larger models to fit on the GPU.

4.5. Repeating computations to save memory

Zheng et al. [24] (2018) note that in LSTM RNN based NMT model, attention layers have a small input/output size, but the intermediate values have a large size. These values are stored in memory as feature maps for later reuse during BWP phase, leading to memory bottleneck. Their first proposed technique avoids storing these values. Instead, only input values are stored and during the BWP phase, the computation of FWP phase is repeated using the saved inputs. The increase in latency due to this repetition is offset by reduction in data transfers to main memory. However, during BWP phase, these extra computations may need a large workspace. They note that in LSTM RNNs, computations must be performed one time step at a time. Based on this, they reuse the same workspace for all the time steps.

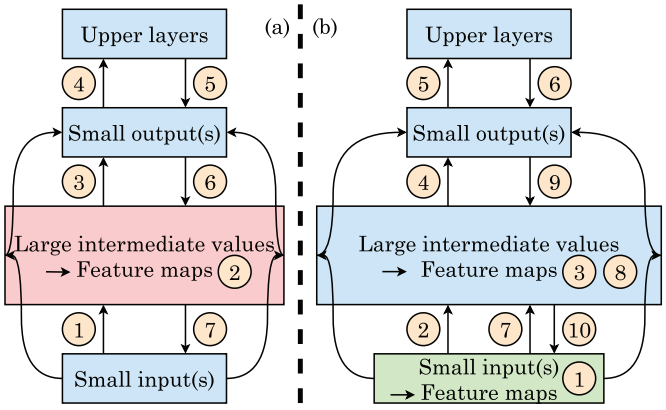


Fig. 29. (a) Conventional backpropagation (b) Partial forward propagation proposed by Zheng et al. [24].

Fig. 29 contrasts conventional backpropagation and the scheme proposed by Zheng et al. [24]. In the conventional scheme, during FWP phase, the output is computed based on the input values and interim values are stored as feature maps (①-④). In the BWP phase, these stored feature maps are used for backpropagating the output gradients (⑤-⑦). In their proposed technique, during FWP phase, interim values are not stored as feature maps; rather, the inputs are stashed (①). When BWP arrives at the outputs of the current layer, the inputs are again forward-propagated to the interim values (⑦-⑧). These are used for performing regular BWP computations (⑨-⑩).

They further note that FC layers take a large fraction of latency in NMT models. In FC layers, the dimension of X (input matrix) is B × H, where $H > B$ since the hidden dimension exceeds the batch size. Also, the dimension of W (weight matrix) is 4H × H. Due to the skewed nature of both the matrices, computing $Y^T = WX^T$ takes less time than $Y = XW^T$ since the former computation is able to achieve higher cache efficiency. Based on this, to optimize FC layers, they propose transposing the data between row-major and column-major layouts. In general, finding an optimal layout is NP-hard. However, they note that FC layers in LSTM RNNs have the same dimensions across all time-steps. Hence, this problem is greatly simplified such that a decision between row-major and column-major layouts is required for only one FC layer and the same layout is sure to be optimal for subsequent layers. Their techniques reduce memory utilization of the training phase and improve the performance of both inference and training phases.

Meng et al. [21] (2017) propose two techniques to mitigate the memory limitations of GPU while training Seq2Seq models. During FWP phase, feature-maps with large reuse time are migrated to CPU memory, which frees the GPU memory. These feature-maps are timely prefetched during BWP phase. Feature-maps with short reuse time are not migrated to avoid CPU-GPU data-transfer overhead.

Their second technique reduces the memory-requirement of Seq2Seq models. These models have encoder, decoder and attention layers. The attention layers need much larger memory than encoder/decoder which are usually RNNs. Let the encoder outputs be the keys and decoder output of previous time steps be the query. For calculating the "attention scores", intermediate results are needed during the BWP phase. Although these interim results take the highest amount of memory, their calculation only requires the summation of keys and queries. Hence, they propose not saving these interim results during the FWP phase and recomputing them during the BWP phase, as shown in Fig. 30. Note that the prefix 'd' in "dKeys" and "dQuery" indicate the correspond-

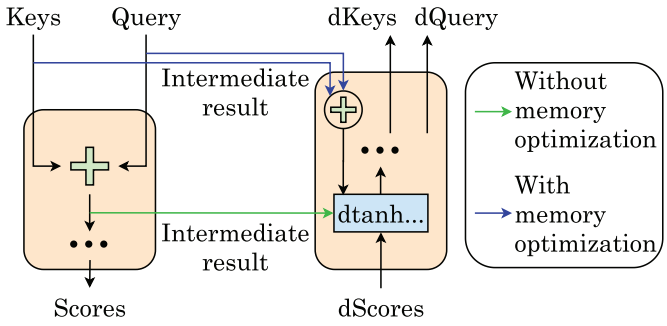


Fig. 30. The second technique (based on re-computation) proposed by Meng et al. [21].

ing gradients. They test this technique for both the attention layer and LSTM/GRU layers. Their technique allows increasing the batch size and training a much larger network on GPU. In terms of training speed, their first technique leads to performance loss, whereas the second technique improves performance.

4.6. Choosing optimal data-layout

Guan et al. [39] (2017) propose a tool and workflow for mapping DNNs specified in TensorFlow to FPGA. Their compiler extracts the topology and computations of the DNN. Then, the hardware implementation is optimized and parameterized. Based on this, the schedule and configuration of the hardware kernel are produced. C++ code is generated from the kernel schedule, which runs on the CPU. Based on the kernel configuration, the FPGA code is produced by creating "RTL-HLS hybrid templates". Here, an efficient compute engine is developed using RTL, and the control logic for it is implemented using OpenCL-based HLS. This combines the best of both RTL and HLS based approaches.

The computation of a layer is modeled as a MMM. For hardware implementation, they allocate a single MMM kernel, which is customized for executing multiple layers. The MMM kernel is implemented using Verilog and tiling is performed to achieve data-reuse. Since the MVM operation in LSTM layers is just a particular case of MMM operation, the MVM operation is easily realized by the MMM kernel. However, since MVM operation cannot achieve reuse of weights, they perform batching of input vectors.

They allocate data buffers in DRAM to form a memory pool. The problem of buffer reuse is cast into graph coloring problem and solved with a polynomial-time heuristic. They note that discontinuous and continuous accesses to DRAM provide a bandwidth of 1 GB/s and 8 GB/s, respectively. Hence, they propose techniques for optimizing effective DRAM bandwidth for different layers. For CONV layer, they propose storing the input data in channel-major layout. To understand the reasoning behind this, consider an example where we have 8 input channels of 3x3 dimension (i.e., 72 elements) and a 2x2 CONV kernel. The input is shown in Fig. 31(a).

- For using `im2col` kernel, the CONV operation can be lowered into MMM operation. As shown in Fig. 31(b), the matrix is divided into 4x2 tiles, then flattened and stored in DRAM. However, this leads to data-duplication between adjacent tiles, increasing the number of elements stored to 128.
- The row-major layout avoids data-duplication. However, as shown in Fig. 31(c), it requires two DRAM bursts for fetching the first tile. This degrades the bandwidth.
- The channel-major layout, shown in Fig. 31(d), requires flattening every row to channel-major order. This averts duplication and ensures access to contiguous elements. Also, unlike the im2col or row-major layout, it does not require data-reorganization for offloading the output.

Their technique outperforms hand-coded FPGA implementation and also scores high on performance per joule metric.

5. Techniques for simplifying RNN accelerators

RNNs have inherent resilience to inexactness [105](2016), which provides scope for using approximate computing for reducing the hardware overheads or RNNs. In this section, we review the techniques for hardware-aware pruning (Section 5.1), accelerators for sparse RNNs (Section 5.2), techniques based on circulant matrices (Section 5.3), techniques that seek to leverage similarity and significance information (Section 5.5) and techniques that use variable-precision (Section 5.4).

5.1. Techniques for hardware-aware pruning

Table 7 presents a classification of techniques for pruning and using block-circulant matrices. As for the criterion for pruning, although all pruning schemes prune values with low magnitude, some techniques also use additional criteria, such as achieving a certain sparsity ratio or a connectivity pattern. These attributes are highlighted under the heading "criterion for pruning".

Cao et al. [43] (2019) note that while unstructured pruning leads to a hardware-inefficient design, coarse-grain pruning compromises model accuracy. To balance these factors, they propose a "sub-row balanced sparsity" (SBS) technique. Every matrix row is divided into multiple sub-rows, each having an equal number of NZ values. Fig. 32 compares SBS with fine-grain (unstructured) and coarse-grain (block-level) pruning techniques, where all the three techniques achieve 50% sparsity. In unstructured pruning, the smallest 50% weights are pruned. Coarse-grained pruning is performed at the granularity of 2x2 blocks and the block magnitude is taken as the average value of its elements. SBS applies unstructured pruning in each sub-row separately. This allows a sparse-MVM kernel to leverage parallelism across the rows and across the sub-rows. Also, by virtue of applying unstructured pruning in each sub-row, SBS preserves weights of higher magnitude, which helps maintain accuracy. By contrast, coarse-grained pruning may preserve some small weights and prune-away some large weights since it prunes at the granularity of blocks. While pruning an example LSTM network to 90% sparsity, fine-grain pruning, coarse-grain pruning (with 4x4 block) and SBS preserve on-average 100%, 28% and 85% of the large weights across the four input weight matrices.

Fig. 33 (a) shows the working of a dot-product operation in SBS. The computations of different sub-rows happen in parallel, but those of the NZ elements in a sub-row happen in series. Since the number of NZ elements is equal in all rows and sub-rows of the matrix, SBS achieves load-balancing and its memory accesses are regular. While computing a "partial dot-product", SBS accesses only one element from every sub-row. Hence, each sub-row can be stored in a separate BRAM for achieving conflict-free memory access and high bandwidth.

They further propose a sparse encoding technique that leverages the balanced characteristic of SBS. It uses a value array and an index array. The value array stores NZ values in a row-major style as shown in Fig. 33(b). In every row, initially, the first NZ elements in every sub-row are stored, then the second element of every sub-row and so on. This reorganization exposes parallelism across sub-rows since consecutive K (=number of sub-rows) elements can be accessed in parallel for performing computations. The index array stores the index of NZ values inside each bank. When each sub-row is stored in a different BRAM block in an FPGA, these indices become the physical addresses in the BRAM block. Experiments show that for sparsity ratios above 40%, the accuracy achieved by SBS is the same as that of unstructured pruning and higher than that of coarse-grain pruning. At sparsity ratios below 40%, all the three pruning strategies achieve almost equal accuracy. Also, their proposed FPGA accelerator for SBS achieves higher throughput and energy efficiency than state-of-art FPGA accelerators [40,70].

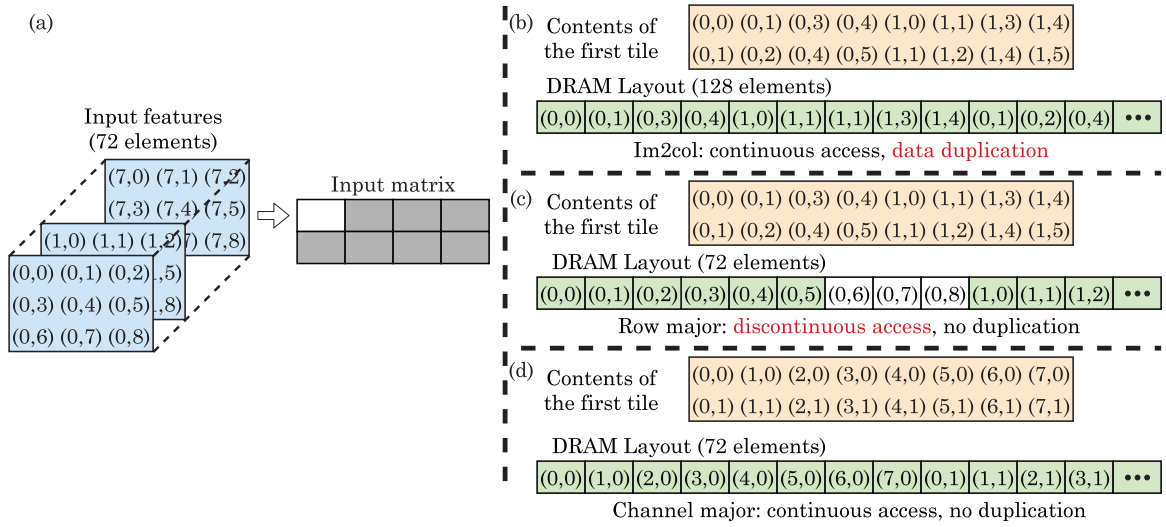


Fig. 31. Layouts evaluated by Guan et al. [39].

Yao et al. [62] (2019) present sub-row balanced pruning scheme for GPUs, which is similar to that of Cao et al. [43](2019), shown in Fig. 32(d). Balancing of workload in each sub-row matches the architecture of GPU and allows exploiting parallelism. For multi-layer CNNs, they divide the whole CNN into layers and prune both the CONV and FC layers one by one. As for GPU implementation, they assign MAC operations of one sub-row to a single thread. Due to the balanced sparsity, the workloads of different threads are balanced. To avoid losing performance due to random accesses from concurrent threads, they use ShM. Different banks of ShM can be accessed parallelly, and with K banks, the bandwidth achieved is K times that of a single bank. As shown in Fig. 34, the dense vector is rearranged and stored in ShM. Then, multiple values can be simultaneously accessed by different threads, achieving high bandwidth. For MMM kernel, their technique achieves near-ideal performance improvement for sparsity ratio between 50% to 97%. For VGG16, LSTM and CTC models, their technique achieves large speedup.

Zhu et al. [65] (2019) propose a SIMD-aware encoding technique for sparse weight matrices. Their technique splits a sparse weight matrix into L-dimensional sub-rows. Let K denote the highest number of NZ elements in any sub-row. Then, each sub-row is compressed to have exactly K elements. For sub-rows having fewer than K NZ elements, padding is done with 0. This approach is shown in Fig. 35. This scheme leads to better load-balancing than a top-K pruning scheme. They also propose architectural changes to tensor cores in Volta GPUs to execute their technique efficiently. Their technique improves the GPU performance significantly and the hardware optimizations bring even larger improvement in performance.

Kung et al. [35] (2019) note that CSR/CSC formats are not effective when the sparsity levels of a network are low. This is because the bit-width required for representing the row-index increases with rising matrix size. For example, in a 1024 × 1024 matrix, 10 bits are required for storing each column index. As such, for density values above 45%, CSC

Table 7
A classification of pruning and quantization techniques.

Category	Reference
Pruning	[7,19,22,31,32,35,40,42,43,47,54,60,62,64,65,76,77,86,88]
Novel format for sparse data	[31,43,54]
Optimizations	Using different frequency at different levels of sparsity [35], Huffman encoding [35], using diagonalRNN, which have no connections to their neighbors, to enable parallelization [77]
Use of block-circulant matrix	[67,70,71,73]
Structured pruning at level of	Column [55], columns or rows [22], columns and rows [22,23,54], blocks [64], sub-rows [43,62,65]
Criterion for pruning	
Only magnitude-based	[7,19,31,47,54,60,65,76,86,88]
Pruning to achieve a certain sparsity ratio	[22,32,35,40,43,62]
Achieving certain connectivity pattern after pruning	[35,42,64,77]
Load-balancing strategies	
pruning so as to	Leave equal number of elements in every sub-row [43,62] or sub-column [32], leave equal number of total elements in rows assigned to different PEs [40], reach hardware-favored dimensions [23]
static load-balancing	Assigning matrix rows to PEs depending on the number of NZ elements [31], matrix reordering for grouping together the rows with similar compute-pattern [54], not skipping computations on zero-data to perform same number of computations in each sub-row [65], by allocating equal number of cells in each group [61], load-balancing happens in helper core model when input-state vector is larger than hidden-state vector [79]
dynamic load-balancing	Redistributing the work across lanes/PEs [7,35]
Adaptive techniques	sparsifying only selected matrices [22], using circulant matrices in input, forget and output gates and not in memory cell [67], using higher size of block-circulant matrices for unimportant gates (e.g., input/output) [73]

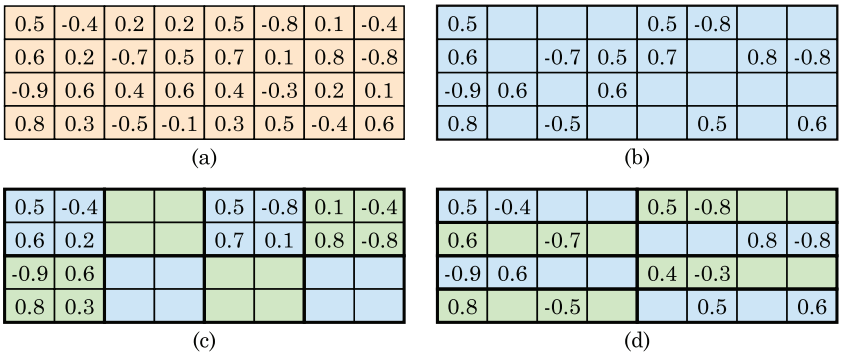


Fig. 32. (a) Original matrix (b) unstructured (fine-grain) pruning (c) coarse-grain pruning (d) SBS technique [43].

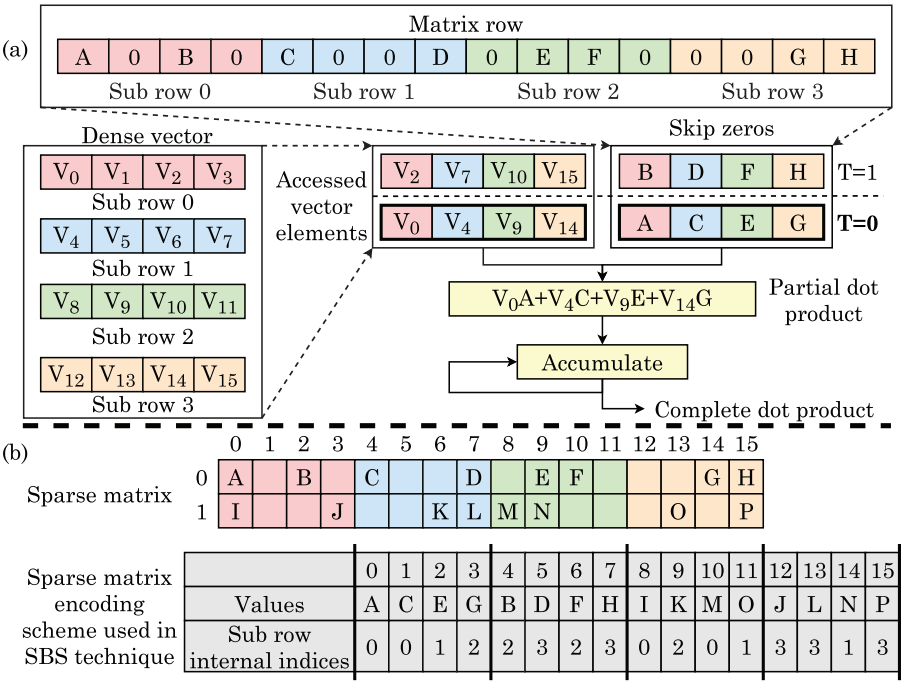


Fig. 33. (a) Dot product engine exploiting inter sub-row parallelism (b) Storage of the sparse matrix using SBS technique [43].

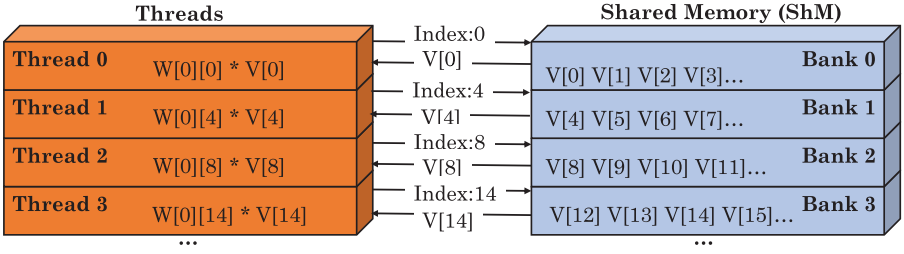


Fig. 34. Following the example of sub-row balanced pruning shown in Figure 32(d), the index of first NZ element in the first four sub-rows is 0, 4, 8 and 14. Hence, the values V[0], V[4], V[8], V[14] are fetched from ShM and supplied to different threads in parallel for enabling sparse MMM.

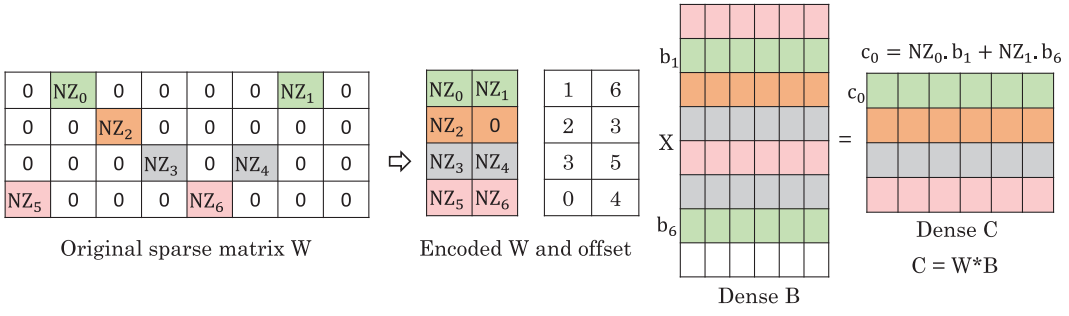


Fig. 35. Illustration of sub-row-wise encoding of a sparse matrix W, with $L = 8$ and $K = 2$ [65]. The encoded W matrix is multiplied with the dense matrix B to obtain the matrix C.

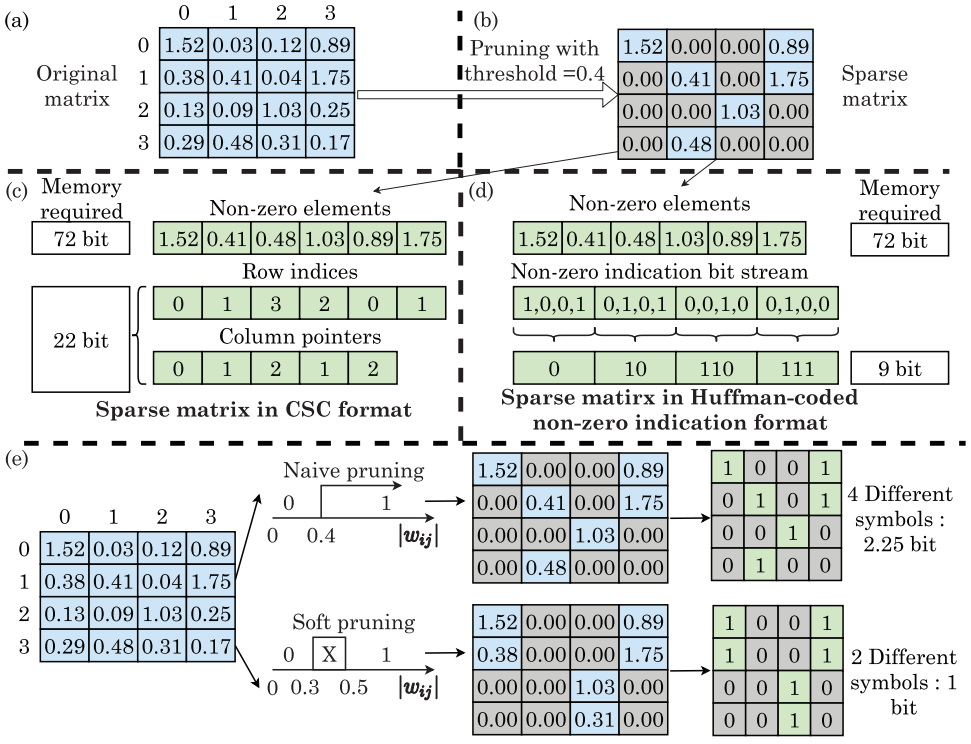


Fig. 36. (a) Original matrix (b) Sparse matrix obtained by pruning using a threshold of 0.4 (c) sparse matrix in CSC format (needs 72 + 22 bits) (d) sparse matrix in "Huffman-coded non-zero indication" format (needs 72 + 9 bits) (e) The matrix obtained from "hard pruning" requires 4 different symbols, whereas that obtained from "Soft pruning" requires only 2 different symbols [35].

format provides no advantage over the dense format. Kung et al. propose a sparse format where, instead of column indices, "non-zero indication" bit-streams are sent. The positions of NZ elements are identified at runtime using counters. To further compress the data, they encode the stream using Huffman encoding, thus leading to a "Huffman-coded non-zero indication" format. Fig. 36(a) shows a sample matrix which when pruned with a threshold of 0.4 gives the sparse matrix shown in Fig. 36(b). Figs. 36(c) and (d) contrast CSR format with the proposed format and show the Huffman encoding of symbols with a length of 4.

They further propose a "soft pruning" scheme. Previous pruning schemes apply "hard thresholding" such that weights below a threshold (say T_0) are set to zero. In their pruning scheme, two thresholds, say T_L and T_H are chosen such that $T_L < T_0 < T_H$. Values below T_L are necessarily pruned and those above T_H are necessarily kept. The values above T_L and below T_H may be kept or pruned depending on two conditions: a target sparsity level is achieved and a frequently-occurring connection pattern is obtained. Their soft-pruning scheme leads to a hardware-efficient pruned matrix. For example, assuming T_0=0.4, T_L=0.3 and T_H=0.5, in the second row, the value 0.38 is kept whereas the value 0.41 is pruned. This is because the bit-pattern 01101100, which has a symbolLength of 8 occurs with a much higher frequency than any other bit-pattern. Fig. 36(e) contrasts "soft pruning" with pruning schemes using "hard thresholding". To recover the accuracy loss due to soft-pruning, retraining is performed.

They evaluate their pruning technique on a network with one input, 2 LSTM and one output layers. Each LSTM layer has 4 wt matrices each for $x(t)$ and $h(t)$, and one weight matrix for the output layer. For sharing the same Huffman tree while applying the soft pruning, they group 4 matrices together. They note that Huffman encoding with a soft pruning technique (with symbolLength of 8bits) provides a lower memory footprint than both Huffman encoding and CSR format. Further, by virtue of reducing the code-length, soft pruning (with symbolLength of 8bits) reduces the decoding latency. Also, a decrease in the number of symbols saves the memory required for storing the Huffman tree.

They further propose an accelerator for handling the pruning format. It has an array of PEs for performing MVM. One PE has four MAC units, each of which handles one gate of LSTM layer. They divide the Huffman tree into multiple levels by splitting the Huffman LUT at a specific depth, e.g., a depth of 4. With rising length of the code, the chances of accessing higher-level tree is very low. Based on this, higher-level LUT is shared between multiple (e.g., 4) PEs. This reduces the memory overhead of Huffman encoding.

The number of MAC operations to be performed for a symbol depends on the number of NZ weights in the symbol. For example, if 7 out of 8 weights are NZ, then 7 MAC operations are performed, which needs 7 cycles. Hence, the decoding circuit stays idle for the next 6 cycles. The number of NZ weights, on average, depends on the sparsity levels of a matrix. Hence, they adapt the clock frequency based on sparsity levels to avoid hardware idling. For example, for sparsity levels of { > 0.75, 0.5 < *sparsity* ≤ 0.75, 0.25 < *sparsity* ≤ 0.5, ≤ 0.25}, the frequency is set to {F,F/2,F/4,F/8}. To achieve load-balancing among the PEs, if the number of NZ weights pending in the queue of any MAC exceeds a threshold, then these weights are distributed to multiple MACs. On a speech recognition benchmark, they perform pruning with the constraint that the accuracy loss remains zero. Their technique provides higher throughput and energy efficiency than an implementation based on the CSC-format.

Wang et al. [32] (2018) present a hardware-friendly pruning scheme. In the first step, during FWP phase, the outcomes of the output gate that are below a threshold are set to zero. This is referred to as "clipped gating" and its enables sparse activations. Also, a regularizer is used in the loss function for striking a balance between sparsity and accuracy. In the second step, top-k pruning is performed as follows. All the weight matrices are stacked, as shown in Fig. 37(a). Then, every column of the matrix is split into groups of size c. Then, each group is pruned such that only top k elements are retained. Fig. 37(b) illustrates (8,3) pruning.

In every group, the weights need to access only one element of x_t or h_t. Hence, the computations of a group are assigned to a single PE. This avoids the load-imbalance problem. For every group, only the NZ values and their indices are stored. They further use two quantization schemes, viz., fixed-point and log-domain quantization. In log-domain quantiza-

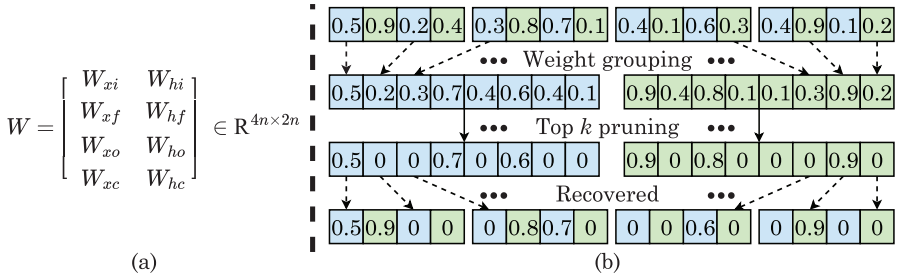


Fig. 37. (a) Stacking of weight matrices to form the big matrix W (b) Illustration of (8,3) pruning i.e., removing 3 least contributing weights out of 8 with the proposed scheme [32].

(a)

Sparse matrix

	0	1	2	3	4	5	6	7
0		A						
1				B		C		
2								D
3		E	F			G		
4		H					I	J
5						K		
6			L	M				
7		N			O			P

CSR

PE0						
Clk	1	2	3	4		
Value	A	H	I	J		
PE1						
Clk	1	2	3			
Value	B	C	K			
PE2						
Clk	1	2	3			
Value	D	L	M			
PE3						
Clk	1	2	3	4	5	6
Value	E	F	G	N	O	P

Unbalanced load

Technique of Park et al.

PE0						
Clk	1	2	3	4		
Value	E	F	G	A		
PE1						
Clk	1	2	3	4		
Value	H	I	J	D		
PE2						
Clk	1	2	3	4		
Value	N	O	P	K		
PE3						
Clk	1	2	3	4		
Value	B	C	L	M		

Balanced load

(b)

Original load

R_id	R_len
0	1
1	2
2	1
3	3
4	3
5	1
6	2
7	3

After sorting

R_id	R_len
3	3
4	3
7	3
1	2
6	2
0	1
2	1
5	1

Original and switched R_id

Original	Switched
0	3
1	4
2	7
3	1
4	6
5	0
6	2
7	5

Fig. 38. (a) Illustration of CSR and the technique of Park et al. [31] for a sample sparse matrix (b) sorting of rows according to the number of NZ entries and the change in ordering of rows by their technique.

tion a value is quantized to an integer power of two. Log-domain quantization requires fewer bits to store a range of values and it allows replacing multiplications with shift operations. They use log-domain quantization for weight matrices that are responsible for the majority of the parameters and fixed-point quantization for remaining parameters and interim results. Their technique achieves sparsity in both weights and activations, whereas some other techniques (e.g., [40]) have sparsity in weights only. On a "word-level language modeling" task, their technique reduces the parameters and computations with only minor loss of accuracy.

Park et al. [31] (2018) present a format termed "compressed and balanced sparse row" (CBSR) for storing sparse matrix, which provides load-balancing in sparse-MVM operations. The baseline CSR scheme assigns computations to the PEs at row-granularity. In their technique, different rows of the matrix are sorted in descending order according to the number of NZ elements in the row. This ordering is used for processing the rows. When the processing of a row is done, the next row with the least load is assigned to a PE. Fig. 38(a) contrasts baseline CSR scheme and the proposed technique for a given sparse matrix. Fig. 38(b) illustrates how the proposed technique sorts the rows (R_id) based on the number of NZ elements in them (R_len). It also shows the original and final ordering of rows. This approach increases the probability of assigning a similar number of NZ elements to every PE.

However, sorting the rows changes the order of hidden-neurons, and hence, that of dot-products. To avoid asynchrony in the update of every gate, they ensure that all weight matrices in a layer use the same final row ordering. For this, they transform the matrix design while generating the CBSR format, as shown in Fig. 39(a). After this, CBSR format is produced for every weight matrix pair of every gate that is accessed in parallel.

For the data-format, in the beginning, the first NZ element of a row mapped to every PE is stored. Then, the column index of every entry is stored, as also done in the CSR format. After storing all NZ elements of a row, subsequent row mapped to that PE is stored. The number of NZ elements in a row are also stored, which allows deciding the number of computations needed for completing a dot-product. The memory-access costs of CBSR format can be avoided by performing a pre-processing step, as shown in Fig. 39(b). In this step, multiplication with an elementary matrix is performed for changing the order of rows or columns of a "weight matrix". In effect, the order of columns in the subsequent matrices is changed for handling the change in the order of the hidden neurons. This transformation step allows the sequential computation

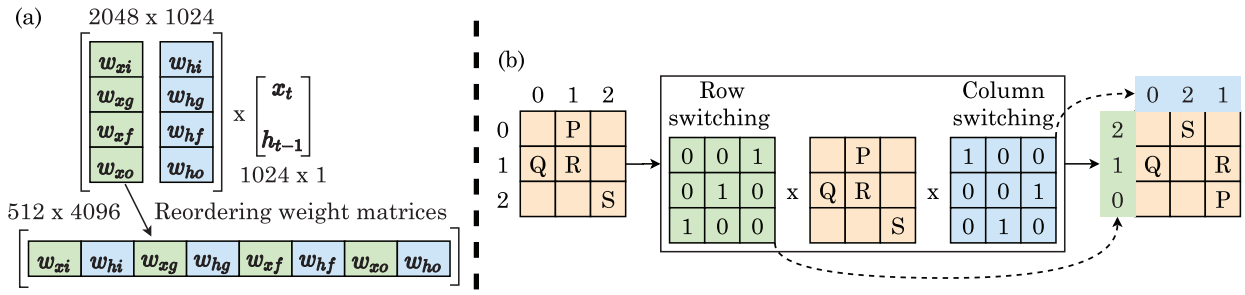


Fig. 39. (a) Matrix transformation while generating the proposed format for storing sparse matrices (b) Matrix multiplication to alter the order of rows and columns [31].

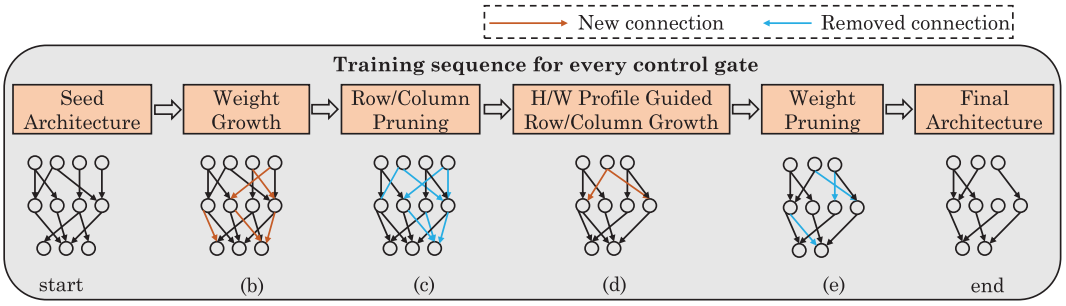


Fig. 40. Training strategy of Yin et al. [23].

of dot-products without the need to know the original ordering of the rows. Since this step is done only once, its overhead is easily amortized. Their proposed format improves the performance and energy efficiency of LSTM accelerators over the use of CSR format.

Rizakis et al. [42] (2018) present a technique for improving the hardware implementation efficiency of LSTM by using approximate computing. For each of the four gates, they concatenate the current input matrix and the previous output vector matrix. On each of these four augmented matrices, they apply a full SVD and utilize a rank-1 approximation. They keep the singular vectors corresponding to the largest singular value. Further, they perform structured pruning of LSTM such that every output neuron must have the same number of inputs. For this, in each row of the augmented matrix, only K elements are kept, which have the highest absolute value. No retraining is performed. Their technique iteratively applies SVD and pruning scheme for updating the weight matrix.

To achieve inter-gate parallelism, a separate hardware unit is used is for each gate. Similarly, to achieve intra-gate parallelism, a DPE is used for performing dot-products in parallel. For each gate, the vectors are stored in off-chip memory and they are streamed to on-chip memory in a tiled manner. After experimenting with different levels of sparsity (decided by K), the number of pruning iterations and tiling factors, a suitable level is chosen such that the impact on application-level accuracy is acceptable and performance is maximized. Experiments on an image captioning workload show that their technique lowers the time required for achieving the desired level of accuracy.

Yin et al. [23] (2019) record the latency of MMM on a GPU. They note that although in general, a smaller MMM is faster than a larger MMM, within a local range of matrix-size, a smaller MMM may be slower than a larger MMM. This effect is termed "latency hysteresis" and the point where such effect starts is termed "hysteresis point". This effect arises because due to vectorization operations, some input dimensions can fully leverage the memory bandwidth and SIMD hardware. This effect is also observed on the DeepSpeech2 model and is seen on both CPU and GPU platforms. They propose an LSTM training scheme to leverage this observation for jointly optimizing latency, model size and accuracy. Fig. 40 illustrates their scheme.

The initial network has a few connections for allowing the initial BWP of gradients. The training proceeds in four phases. In the "weight growth" phase, only the most important connections are iteratively inserted for reducing the loss function. This helps in achieving high accuracy without using a FC network design. After this, "structured row/column pruning" scheme is used for reducing the model dimensions and the inference latency. For importance ranking, they use the sum of magnitudes of the weights in a row/column and pruning is performed iteratively along with retraining. Then, based on the latency values on the given processor, the hysteresis effect is taken into account and "row/column growth" algorithm is used for achieving a network with locally optimal dimensions. This improves both accuracy and latency. Finally, a few weights are pruned for further reducing model size. They evaluate their technique on both CPU and GPU. For LSTM models, their technique reduces model size and inference latency without losing accuracy.

Van Keirsbilck et al. [77] (2019) note that neurons in the human brain have only locally-connected and not FC architecture. They study locally-connected RNNs having structured-sparsity since they have a higher scope of parallelization than RNNs with unstructured-sparsity. FullRNN and various sparse RNNs are shown in Fig. 41. The full-RNN is a FC RNN where all neurons possess recurrent connections with all other neurons. Here, whole MMM is required for recurrent transformation and the computation cannot be parallelized across neurons. In GroupRNN, the neurons are partitioned into multiple groups, with no inter-group connectivity but full intra-group connectivity. Different groups in a layer operate independently and hence, they can be parallelized.

In BandRNN, neurons are connected to themselves and their next neighbors. The resultant "recurrent weight matrix" is sparse with band-diagonal shape. The number of recurrent weights is linearly proportional to the number of neurons. A corner case of BandRNN is DiagonalRNN, where a neuron has no connection to its neighbors. Hence, all neurons of a layer are independent and can be computed in parallel. Also, the recurrent transformation of DiagonalRNN requires only EW operations and not MMMs. Assume a layer with N neurons where C shows the number of connections per neuron. Let G show the number of groups in a group-RNN. Then, the number of recurrent weights in full-RNN, group-RNN,

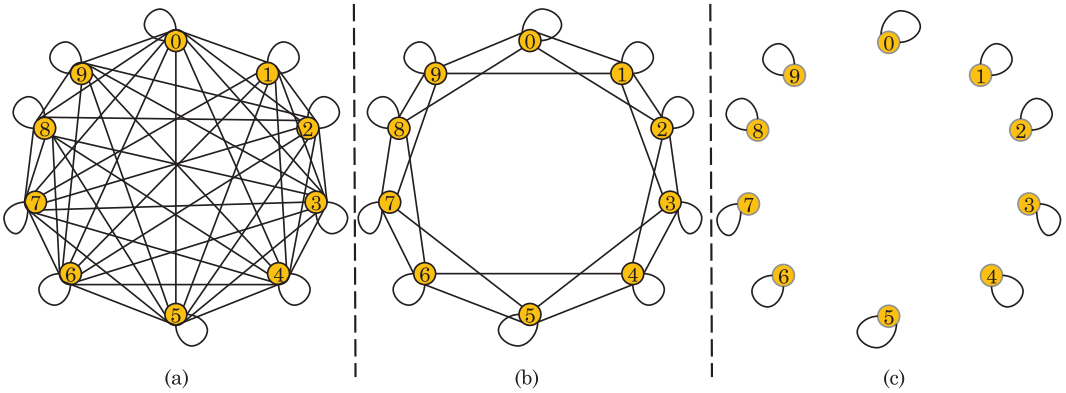


Fig. 41. (a) Full RNN (b) Band RNN (c) Diagonal RNN [77].

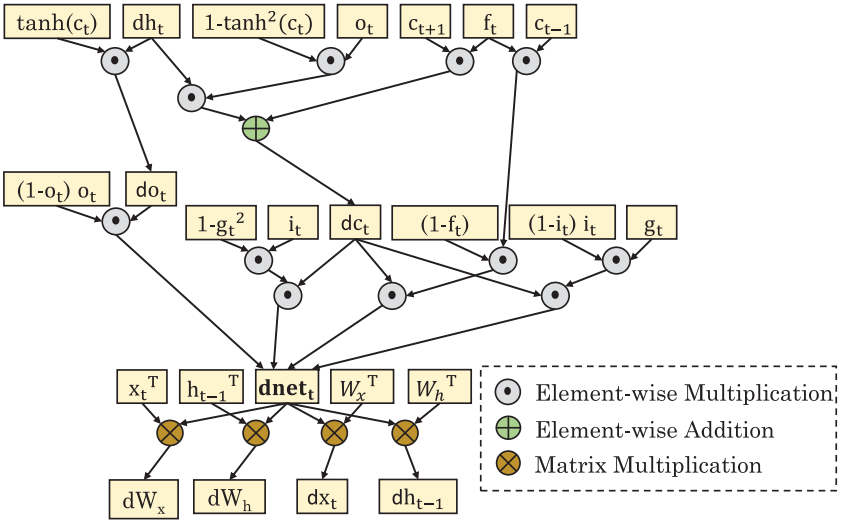


Fig. 42. Working of BWP phase in LSTM [22].

band-RNN and diagonal-RNN are N^2, N^2/G, N^*C and N, respectively. They note that for $N = 512$, the inference latency of DiagonalRNN is 1/9th of that of FullRNN. DiagonalRNN reduces the memory footprint, which allows storing the weights in on-chip memory. They also note that DiagonalRNN is complementary to PreRNN [106](2018) since DiagonalRNN lowers the number of recurrent weights and PreRNN reduces the complexity of input-to-hidden transformations. Hence, by combining them, a powerful video-action recognition model can be achieved. They evaluate these RNN designs for various tasks, such as language modeling and phoneme/speech/video recognition. The predictive performance (e.g., accuracy or test perplexity) of DiagonalRNN is close to that of FullRNN and for some cases, BandRNN has better predictive performance than FullRNN. Also, the integration of PreRNN with DiagonalRNN or BandRNN further improves the effectiveness of video action recognition.

Zhu et al. [22] (2018) present a structured sparsity technique for accelerating BWP phase of LSTM. The working of BWP phase is shown in Fig. 42. Conventional techniques remove insignificant values in the output gradient (dh_t in Fig. 42). They note that the "gate gradient" ($dnet_t$) has higher resilience towards sparsification than dht. Also, it is used in four MMMs and hence, sparsifying it is very advantageous. Hence, they sparsify *dnet* and not *dh*.

They present two approaches for structural sparsification, termed coarse-grain and fine-grain, for a layer of H LSTM cells. N shows the mini-batch size. The coarse-grain scheme divides the dnet matrix evenly into multiple slices and computes the L2 norm of every slice. A slice has P successive rows or columns. Thus, the matrix is not divided at the granularity of individual rows/columns, which reduces the overhead of regrouping. Then, neighboring slices are collected into a region. For instance, in Fig. 42, the dnet matrix of size N × 4 × H is split into H slices, each having P elements. Then, a group of R slices are combined together to form a region. Based on the L2 norm of the slices in a region, the least significant slices are removed, as shown in Fig. 43(a) to achieve the desired level of sparsity, e.g., when S=R/2, 50% sparsity is achieved. Finally, the sparse dnet matrix is stored in a dense matrix format and an extra offset index array is saved for recording the index of unpruned slices. This allows efficient computation of dnet matrix on the GPU using tiled MMM.

The limitation of coarse-grain scheme is that it requires a region to have at least one whole row or column of dnet matrix. This constraint affects the training quality. In order to have higher flexibility, the fine-grain scheme can be used, which breaks the dnet matrix into several $M_y \times M_x$ sub-matrices, as shown in Fig. 43(b). The region size is reduced to a slice of N × M_x, which helps in reducing the size of the index array. The limitation of fine-grain schemes is that its memory accesses do not coalesce, which leads to higher memory latency.

Since the TensorCores of Volta GPUs can perform efficient MMM on 4 × 4 matrices, they set $P = 4$ for coarse-grain scheme and $M_x = 4$ and $M_y = 8$ for fine-grain scheme. Their experiments show that with 50% sparsity, their schemes provide nearly 1.4 × speedup with only a small loss in the result quality. The coarse-grain scheme provides higher speedup than the fine-grain scheme at the cost of higher loss in quality. To compensate for the loss in result quality due to sparse training, they combine sparse training with dense training. For example, the ratio of sparse and dense training can be 75% and 25%, respectively. Experiments confirm that their sparse/dense-mixed training approach leads to

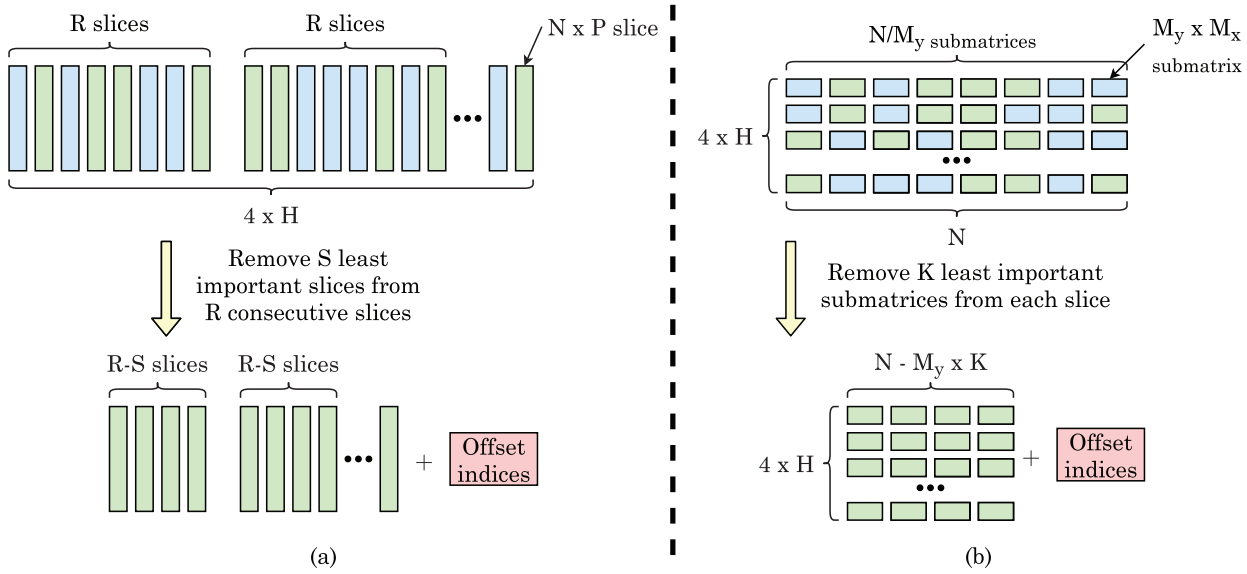


Fig. 43. (a) Coarse grain sparsification (b) Fine grain sparsification proposed by Zhu et al. [22].

lower training time while providing comparable predictive performance (measured as perplexity or "bilingual evaluation understudy" score) as the dense-only training.

5.2. Accelerators for pruned RNNs

Ardakani et al. [19] (2019) present a technique to avoid ineffective recurrent computations in RNNs. They note that the state h_{t-1} has the largest contribution in the recurrent computations. The input vector x_t is generally encoded as one-hot, except for problems such as "word-language modeling" that have very high dimension vocabulary. Thus, in general, the dimension of h_{t-1} is higher than that of x_t. Their technique trains LSTMs so that it retains only useful connections in the hidden states h_{t-1}. Specifically, in the FWP phase of LSTM training, any value lower than a threshold is pruned. But during the BWP phase, the unpruned (dense) h_{t-1} vector is used for updating the LSTM parameters. They note that for "character-level language-modeling", "word-level language-modeling" and image classification, 97%, 90% and 80% (respectively) hidden-state can be pruned with negligible impact on the accuracy.

They further design an architecture for accelerating the pruned network. They note that the computations on zero-valued elements of the hidden-state can be avoided. However, in the case of limited memory bandwidth found in edge devices, the PEs are underutilized, and throughput is reduced. To mitigate this issue, they propose batching. This requires more memory elements for storing the interim results of each batch. The challenge in batching is that in any cycle, the computations can be skipped only if all the input elements of all the batches are zero. Fig. 44(a) illustrates such "skipping" for a vector-matrix multiplication.

Their architecture of the proposed accelerator is shown in Fig. 44(b). It has four tiles, each of which handles the computation of one gate. Every tile has 48 PEs for doing MAC operations and a unit for performing sigmoid or tanh AF. A scratchpad is used for storing interim results at a batch size of 16.

A PE can take input from either off-chip DRAM or scratchpad of the same tile or other tiles via local or global routers. Based on the data dependencies, the results from one tile are forwarded to the other tile. Once the computation of one timestep is over, the results are sent to an encoder, which counts up if the current input value is zero for all the batches. The offset thus computed is stored along with h_t in the off-chip DRAM. For the computation of the subsequent timestep, the offset is used for reading only the non-zero weights. For the batches, different stages of the pipeline are fed using the input/weight register. On their accelerator, the use of sparse format achieves over 5 × higher efficiency than the use of the dense format.

Han et al. [40] (2017) present a pruning scheme that prunes the weight matrix in a manner that sparsity ratio is almost the same for all the PEs. This leads to load-balanced allocation with a negligible impact on accuracy. Fig. 45 contrasts load-unaware and load-aware pruning schemes and the reduction in execution cycles achieved by load-aware pruning. Then, 32b floating-point weights are quantized to 12b integers. Both weights and activations are quantized using a linear quantization approach. To achieve byte-alignment of weights and "relative row indices", they combine 4b indices with 12b weights to achieve 16b data. Thus, the dynamic quantization scheme used by them leads to complex implementation. Further, in pruned LSTM, multiple PEs consume a single element of the voice vector. This mandates synchronization between the PEs, which incurs high latency due to the load-imbalance.

They further present an FPGA architecture that directly operates on the pruned/quantized data. It is composed of PEs, each of which handles computations between a piece of voice vector with the partial weight matrix. Every channel applies LSTM on one voice vector sequence. Sparse-MVM and EW multiplication are the most costly operations. They pipeline the computations of LSTM while taking the constraints into account, for example, the operations that are mutually dependent or consume the same resources cannot be executed in parallel. To mitigate the load imbalance issue, they share a queue between all PEs in a channel. This queue has multiple FIFOs, each having a workload for a PE. Since different PEs get their workload from their FIFOs, they do not have to wait for other PEs and thus, the stall due to imbalance is avoided. Each channel also has engines for performing sparse-MVM, AF, EW multiplication, and for handling encoded weights. Their implementation obtains a large boost in performance and energy efficiency.

Gupta et al. [7] (2019) present an accelerator for RNNs with weight and activation sparsity. They use a baseline that incorporates many optimizations: (A) language modeling for post-processing the RNN output based on language semantics and structure, (B) pruning of low-magnitude parameters and (C) linear quantization. (D) Also, they use "knowledge distillation" for training RNNs with ReLU, which achieves same accuracy as the GRU baselines with tanh. They note that activations consume a much larger fraction of memory in RNNs than they do in CNNs. They first propose techniques to make activations sparse: (1) Un-

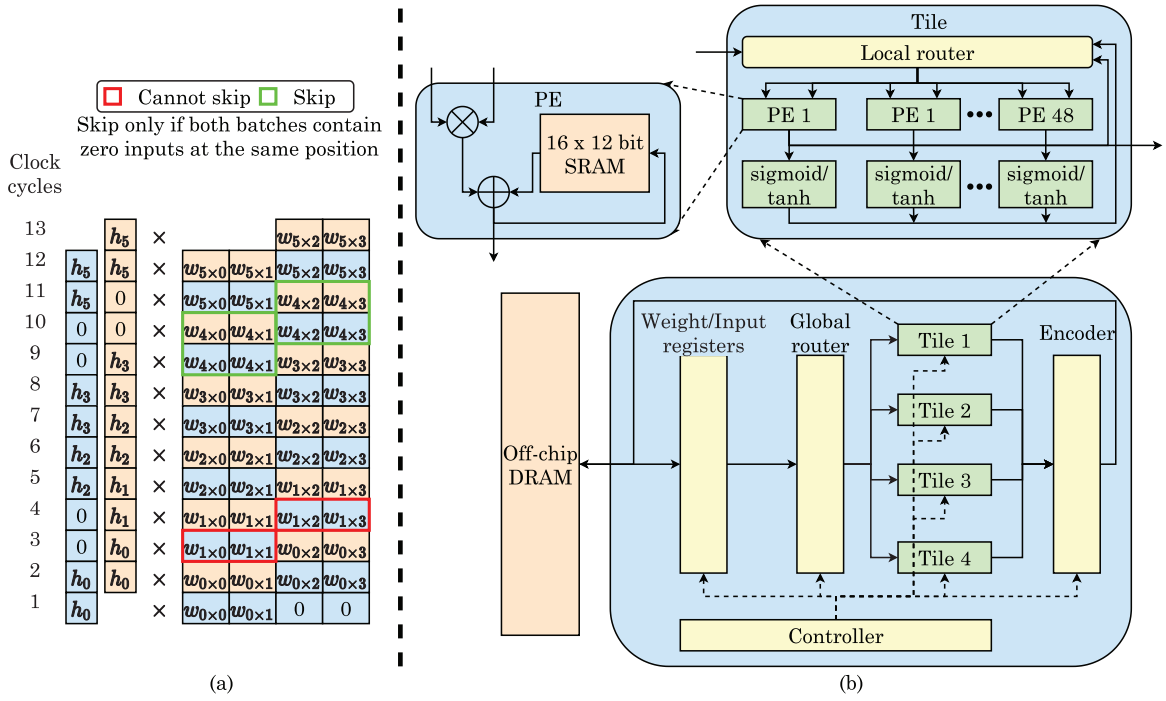


Fig. 44. (a) Skipping computations while batching in vector-matrix multiplications (b) Accelerator architecture proposed by Ardakani et al. [19].

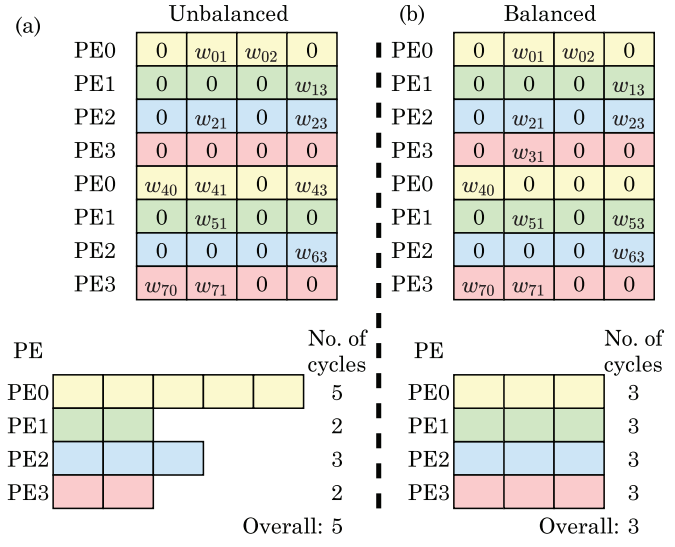


Fig. 45. (a) Load unaware pruning (b) Load aware pruning [40].

like the technique of Silfa et al. [33](2018), they process every timestep sequentially. This halves the storage requirement of interim values. Also, since they use ReLU activation, sequential computing makes the interim values sparse. (2) ReLU sets 80% of hidden-states to zero. (3) Batch normalization scheme performs linear transformation that maps sparse inputs into dense ones and removes input sparsity. To address this issue, they incorporate this transformation into the next layer's weights. After this optimization, 60% of inputs become zero.

They further propose an encoding scheme for storing sparse weights and activations. This scheme uses separate bitmasks for weights and activations. For a NZ weight/activation, the corresponding location in the weight/activation bitmask stores a value of 1. For decoding, they perform AND operation between these bitmasks and obtain the work mask. This bitmask stores 1 at the locations where both the weight and the activation are NZ. Then, by performing leading NZ detection and population-count, the addresses of weights and activations to be fetched from memory are identified. These steps are pipelined.

The CSR scheme encodes weights and activations separately. It skips zero-valued activations in computation, not storage. Also, every PE needs to store its own set of row pointers. CSR scheme is good when sparsity levels are high. By contrast, their encoding scheme avoids row pointers and finds the address of NZ activations and weights through logic operations. Fig. 46(a) illustrates the CSR encoding scheme, whereas Fig. 46(b) shows their proposed encoding scheme.

Their technique can also exploit sparsity in output neurons. Specifically, the distributions of $W_x x_t$ and $W_h h_{t-1}$ are different. Since batch-normalization is applied only on inputs (x_t), $W_h h_{t-1}$ tends to be more negative than $W_x x_t$. In a timestep t, first $W_h h_{t-1}$ is computed and then $W_x x_t$ is computed. Since the ReLU function sets negative values to zero, they propose skipping the computation of $W_x x_t$ whenever $W_h h_{t-1}$ is highly negative.

Their proposed accelerator, shown in Fig. 47, has a 2D array of PEs, each having multiple lanes. Lanes in a PE share activation RF. Every lane has its dedicated weight and weight mask SRAMs and the matrix is split equally across the SRAMs of all the lanes. Vertical and horizontal lanes evenly divide the input and output dimensions of the matrix to balance the parallelism across both dimensions. A lane computes partial product for a few rows and columns in the MVM. These partial products are accumulated as they are computed.

For processing a bidirectional RNN, they perform layer-wise execution, and in every layer, the first forward pass is done for all timesteps and then, a backward pass is done. In every timestep, the following computation is done: $h_t = ReLU(W_x x_t + W_h h_{t-1})$. To execute a layer, all weights for the forward pass (W_x and W_h) are loaded from DRAM to weight SRAMs in the lanes. Also, all inputs (x_t) are fetched into activation SRAMs. While the forward pass is being executed, weights for the backward pass are fetched into different SRAMs, achieving double buffering. After parallelization of $W_x x_t$ and $W_h h_{t-1}$ computations, two bottlenecks remain: (1) vector addition between them due to the limited bandwidth of activation SRAMs. To improve bandwidth, they partition the activation SRAM into smaller banks. This increases lane utilization

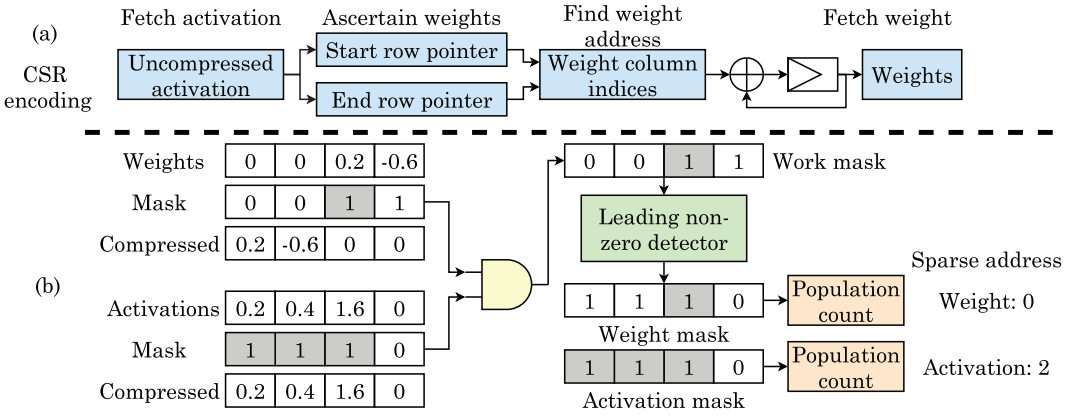


Fig. 46. (a) CSR encoding (b) encoding scheme proposed by Gupta et al. [7] (shown with an example).

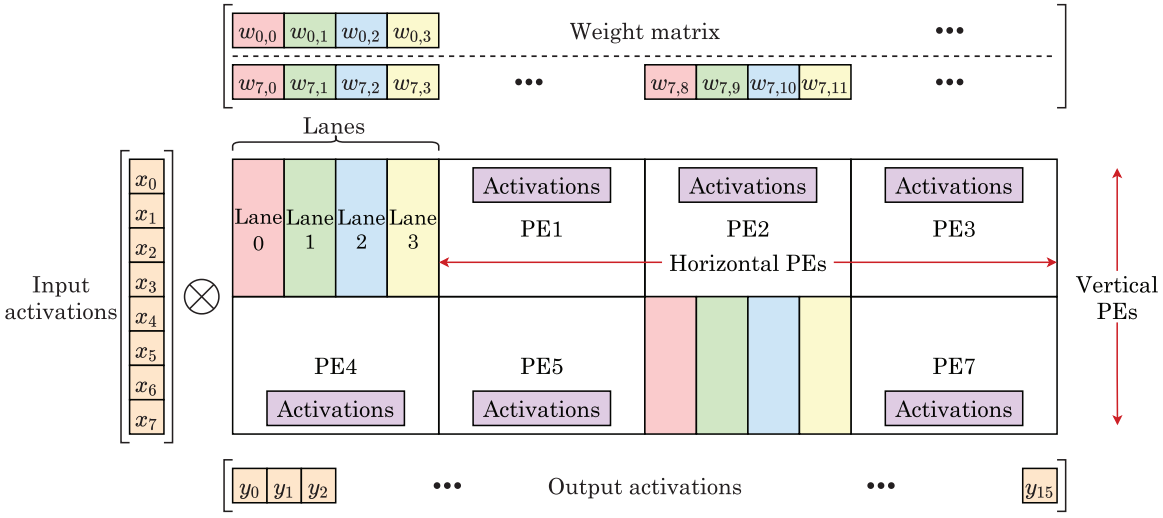


Fig. 47. Proposed accelerator along with input actiivations, weight matrices and output activations split across the PEs [7].

and reduces the latency of vector addition. (2) imbalanced load arising due to the non-uniform pattern of NZ activations across PEs. To mitigate this issue, they redistribute the work of early-finishing lanes in intra-PE and inter-PE manner. In the intra-PE redistribution, the work is reassigned to lanes in the same PE. Since these lanes work on the same activations, only a few weights need to be copied in nearby lanes. In inter-PE redistribution, the work is reassigned to lanes of other PEs. This requires duplication of both weights and activations but avoids the need for complex interconnects. Compared to a contemporary sparse RNN accelerator, their technique provides higher performance/area/energy efficiency.

5.3. Techniques based on circulant matrices

"Block-circulant matrix": A circulant matrix is a square matrix, such that by circularly rotating its first rows (or columns), all the remaining row (or column) can be obtained. Any matrix can be transformed into a set of "circulant submatrices". For RNNs, "block-circulant matrices" achieve similar performance as the full matrix [73](2019). Fig. 48 illustrates reformatting of a 8x4 matrix into a "block-circulant matrix" having two 4x4 circulant matrices. A circulant matrix can be represented by a single row vector only, which reduces the number of parameters. For instance, in Fig. 48, the number of parameters in the original and block-circulant matrices are 32 and 8, respectively. The use of a block-circulant matrix also reduces the computational-complexity of CONV since the MVM operation is replaced by FFT/IFFT.

0.36	1.29	0.03	-2.69
-0.89	2.34	-0.57	-0.45
1.51	2.91	4.19	2.15
0.19	-2.76	-0.93	1.59
0.81	1.17	0.64	-2.18
-0.21	0.53	1.18	0.97
-0.08	-1.37	1.92	2.87
1.33	-0.87	0.01	-2.37

(a) Unstructured weight matrix (32 parameters)

0.36	1.29	0.03	-2.69
-2.69	0.36	1.29	0.03
0.03	-2.96	0.36	1.29
1.29	0.03	-2.69	0.36
0.81	1.17	0.64	-2.18
-2.18	0.81	1.17	0.64
0.64	-2.18	0.81	1.17
1.17	0.64	-2.18	0.81

(b) Block-circulant weight matrix (8 parameters)

Fig. 48. An example of block-circulant matrix [70].

If the original weight matrix W has a dimension of $M \times N$, then, it is split into square blocks of size k*k, each of which is a circulant matrix. The number of blocks is (M/k)*(N/k). This technique reduces the number of weights from M*N to M*N/k and thus, reduces both computations and storage overhead. With the increasing value of k (block dimension), there is an increasing reduction in the number of computations at the cost of higher loss of accuracy. If the first-row vector of weight matrix is w_{ij} then, the computation of $w_{ij}x_j$ can be done using FFT-based CONV. FFT CONV further reduces the number of computations [104].

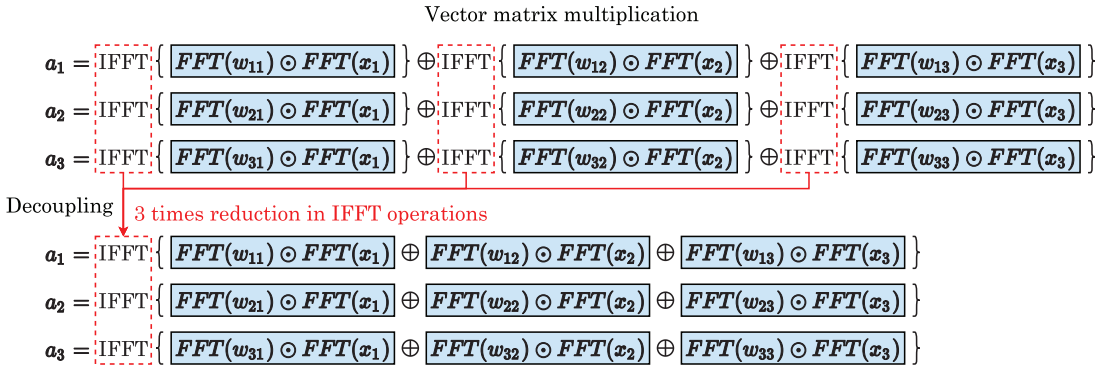


Fig. 49. Vector-matrix multiplication using pre-computed FFT of weights and decoupling of FFT/IFFT [70,71,73].

Wang et al. [67] (2017) present techniques for improving memory efficiency of LSTMs. They note that while the weights in the memory cell show sensitivity to numerical errors, the weights in the input, forget and output gates can tolerate errors. Hence, they propose using circulant matrices only in the latter gates. This reduces the number of weights by nearly 4×. Three quantization schemes are used: (1) exponential, which quantizes the value to an integer power of two. (2) Ternary: the value is limited to be either -1, 0, or +1. (3) $Q_{m,f}$ quantizes a value to its nearest "fixed-point value", where the number of integer and fractional bits is m and f, respectively. In decreasing order of accuracy, the quantization schemes are $Q_{m,f}$, exponential and ternary. In decreasing order of memory saving, the schemes are ternary, $Q_{m,f}$ and exponential. Exponential and ternary quantization schemes allow replacing multiplication with shift and add operations, and their hardware implementation requires less area. Retraining is performed to recover the accuracy loss.

LSTM weights are divided into multiple groups and different quantization schemes may be used for each group. For example, they propose storing all six weight matrices in the input, output and forget gates as a circulant matrix with exponential quantization. All the remaining weights and interim values are quantized using $Q_{m,f}$. They also propose a hardware accelerator for implementing the proposed optimizations. Their optimizations reduce memory usage by more than 95% with insignificant accuracy loss.

Wang et al. [70,71] (2018) present block-circulant matrix-based hardware accelerators for optimizing DNNs. In the FPGA implementation, for a $M \times N$ weight matrix, each PE has N-fold parallelization and a depth of $\log N$. The IFFT is also implemented by the same PE except for that division and conjugation operations are also required. They propose three optimization schemes. (1) Based on the fact that both FFT and IFFT operations are linear, they decouple them and move the IFFT operator outside the accumulation operation, as shown in Fig. 49. (2) Since the weights (w_{ij}) do not change during inference, they precompute the $F(w_{ij})$ values and store them in BRAM buffers. This halves the number of calls to FFT operator for each CONV. In general, for a weight matrix with MxN elements, pre-computation of FFT brings-down the number of FFT computations from MxN to N times and the decoupling brings-down the number of IFFT from MxN to M times. (3) Although $F(w_{ij})$ has both real and imaginary portions, half of the complex conjugate numbers need not be stored based on the symmetry property of FFT output values. The same property also allows reducing the number of multiplications and additions in the EW multiplication between the vectors $F(w_{ij})$ and $F(x_j)$.

Further, they quantize the weights into 16b fixed-point numbers. They note that in the first stage of the IFFT pipeline, the output of IFFT is divided by the block size k. The division is implemented as the right-shift operation. Since "right shifting" one bit at a time leads to higher accuracy than "right shifting" multiple bits in one go, they uniformly spread the shift operations inside the stages of IFFT pipeline. In the accumulation stage, data-overflow may occur. To avoid it, they move the uniformly-spread right-shifting operations from IFFT pipeline stages to the FFT pipeline stages. As FFT is computed before accumulation and "right-shifting" reduces the magnitude of numbers, the chances of overflow in the accumulation phase are reduced.

They disconnect the "feedback edges" from cell output c_t to the LSTM output. This allows representing the LSTM algorithm in the form of a "directed acyclic graph". The challenge in pipelining this algorithm is that there is a huge difference in the computation complexity of circulant CONV and EW multiplication. To address this issue, they divide the pipeline into multiple smaller coarse-grain pipelines. For overlapping their latency, double-buffers are inserted for every pipeline pair. They split the LSTM operator graph into three stages and realize each stage as a coarse-grain pipeline. They finally use an algorithm to perform efficient scheduling of operators to different stages. Based on the FPGA resources, this algorithm finds optimal replication factors for different stages of the pipeline for optimizing the throughput. Their technique can implement a range of LSTM variants on FPGA and achieves high efficiency.

Li et al. [73] (2019) propose techniques for efficient implementation of RNNs using block-circulant matrices. They use "alternating direction method of multipliers" (ADMM) training to directly train the weight matrices in the "block-circulant format" by training a single vector for every block. The use of ADMM also improves training speed and accuracy. Also, there is no need to change a trained weight matrix to block-circulant representation. For the `speech recognition` benchmark, the use of a block-size of 4 or 8 gives near-zero accuracy loss and even with a block-size of 16, the accuracy loss remains negligible. In fact, on using a block-size of 8 or 16, the entire `speech recognition` model can fit in the on-chip buffer of FPGAs. They apply the three optimizations incorporated by Wang et al. [70,71], as mentioned above. In addition, they note that in the first two levels of FFT, the "W twiddle factors" are -1, 1, i or -i. Hence, no multiplications are required in these levels of the butterfly diagram. Further, the number of units that need to perform multiplication is only half in the third level, 1/4 in the fourth level and so on.

For overall optimization, first, the minimum and maximum block size are found so as to fit the RNN model in the on-chip parameter, without incurring a large loss of accuracy. Also, block-size is constrained to be a power of two. For example, if the minimum and maximum block size are 8 and 64 (respectively), then the choices of block-size are limited to 8, 16, 32 and 64. Then, the model size (LSTM or GRU) is selected. Since GRU has smaller memory and compute overhead, it is preferred, as long as the accuracy constraint is met. Then, the block-size is increased for the less-important weight matrices such as input/output matrices that do not propagate to the next time-step.

They further propose a hardware-accelerator design for LSTMs and GRUs. For example, for LSTM, they propose a coarse-grained pipelining design, which has the following three stages: (1) multiplication of weight matrices with the input vector. (2) Non-MVM operations, e.g.,

Table 8
A classification of low-precision techniques.

Category	Reference
Low-precision strategies	
Quantization	[7,13,19,20,30,34,39–41,43–45,47,48,50,56,66,67,70,71,73–76,79,86,88,89]
Variable-precision inference	[13,34]
Adaptive techniques	Layer-specific quantization [76], applying quantization in only selected layers [12], using different quantization schemes for weight matrices of different gates [32,67], different bitwidth for different (weight/input/hidden-state/intermediate-results) data values [20,34,38] and different cells [37],
Precision/bitwidth used by different works	
Floating-point	32-bit [10,12,14,33,39,44,55,74,80], 16-bit [33,34,54,57,69,85]
Fixed-point	16-bit [37,39,40,43,48,49,55,68,70,78,79,90], 12-bit [29,68,73,76], 10-bit [7], 8-bit [12,13,30,33,36,37,43,52,67,68,86], 6-bit [75], 5-bit [38], 4-bit [43]
Integer	12-bit [40,47], 8-bit [33,34,44,47,80], 4-bit [33]

diagonal MVM, adding bias and AFs. (3) MVM for "projection matrix" and projected output. Double-buffering is used between different stages for storing inputs and interim results since their dimension is small. Double-buffering is not used for storing weights. For performing the operations in each stage of coarse-grained pipelining, fine-grained pipelining is used. They also use quantization for inputs and weights. They implement their technique on FPGA and show that it achieves much higher energy efficiency than the contemporary FPGA-based techniques.

5.4. Techniques for using variable-precision

Table 8 provides an overview of strategies of using low-precision.

Silfa et al. [34] (2019) present a technique for dynamically adapting the precision of LSTM cell-state. Fig. 50 shows the variation of cell-state with timestamps for different precisions. They note that the precision required depends on the cell-state. When the cell-state changes at a slow pace, the absolute difference between the cell-state in "full-precision" and "low-precision" is small (19% in their experiments). However, in the regions shown as "peak" in Fig. 50, the cell-state changes fast and hence, the above difference is very large (78% in their experiments). This leads to accuracy loss. They use 8b as full-precision and 4b as low-precision because the use of 8b provides the same accuracy as the use of 32b floating-point.

Unlike previous works that change precision at each layer's level, they change the precision of different neurons in different timesteps. Specifically, they individually adapt the precision of input vectors x_t and h_{t-1} and their weights for every element of the cell, in each timestep. Their technique operates in three phases. In the profiling phase, low precision is used and the maximum (C_{max}) and minimum (C_{min}) values of cell-state (C_i) are found. Here, C_i denotes the cell-state of i^{th} neuron.

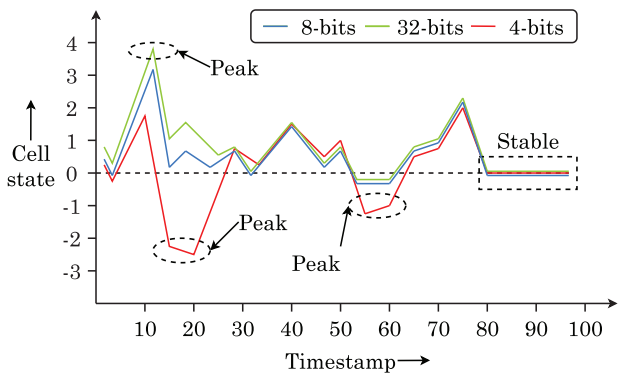


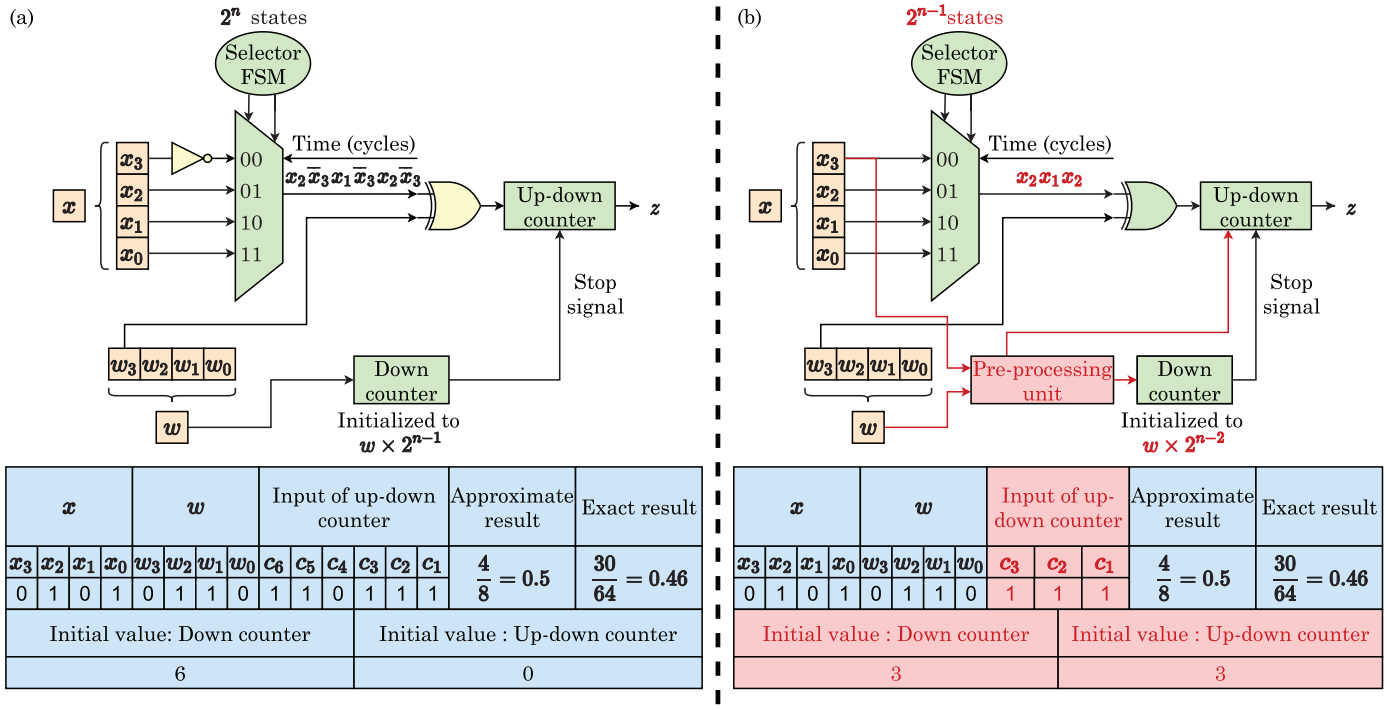
Fig. 50. Variation of cell-state with timestamps for 4-bit, 8-bit and 32-bit precision [34].

After a fixed timestep, execution enters into the stable phase. Here, if C_i is found to exceed C_{max} or C_{min} by a threshold, a peak is ascertained; the execution enters peak-phase and the precision is changed to full. When C_i value again comes in the normal range, execution changes to a stable phase and the precision is changed to low.

For supporting variable precision, they use "serial inner product" engine, which computes an inner product over multiple cycles. In each cycle, one bit of the first operand is multiplied with the entire (i.e., all the bits of) second operand. Then, the partial products are accumulated. The total latency of multiplication equals the bitwidth of the first operand. By having multiple such engines, the throughput can be increased. Also, for large-size matrices encountered in LSTMs, these engines can be easily kept busy. Since, in any timestep, an index may be accessed in high or low precision, they also propose an energy-efficient approach for storing and fetching both indices. Their technique is able to use low precision for 2/3rd of the total execution time. Their technique provides large speedup and energy saving with no accuracy loss and only minor area overhead.

Azari et al. [36] (2020) present an architecture for accelerating LSTMs, which uses inexact multipliers (IMs). These IMs are used in both MVM and EW multiplication modules. Fig. 51(a) shows an IM proposed by Sim et al. [107] and also illustrates its working. The inputs to the IM are signed fractions. The input of the LSTM layer 'x' $\{x_{n-1}, x_{n-2}, \ldots, x_0\}$ and the corresponding weight 'w' $\{w_{n-1}, w_{n-2}, \ldots, w_0\}$ serve as inputs of the IM. The finite state machine generates the bitstream S according to the following rule: x_{n-i} appears at cycle 2^{n-i} and subsequently repeats after every 2^i cycles. The length of this bitstream is equal to the absolute value of the weight. The difference between the number of '1's and '0's is approximately equal to the approximate result. The up-down counter counts this difference while the down counter keeps track of the size of S. Azari et al. [36](2020) modify the IM as shown in Fig. 51(b) to halve the execution time while incurring minimal logic overhead. The pre-processing unit extracts the MSB of x and initializes the up-down counter. The bitstream for the IM of Sim et al. [107](2017) in the illustration is $\{x_2\bar{x}_3x_1\bar{x}_3x_2\bar{x}_3\}$. Since the MSB x_3 is accessed once every two cycles, the pre-processing unit eliminates this need and reduces the execution time. The modified bitstream of the IM of Azari et al. [36] is $\{x_2x_1x_2\}$ which is half as long as that of the original IM [107]. With increasing bitwidth of IM, its accuracy increases, e.g., an 8b IM has an accuracy of 97.8%, whereas a 16b IMB has an accuracy of 99.9%.

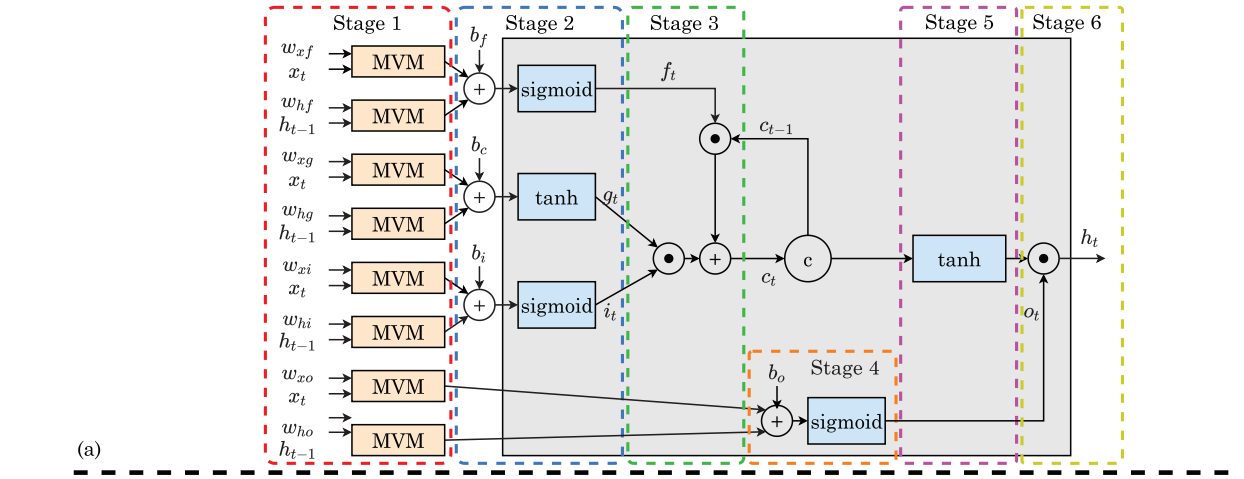
Their technique performs pipelining across various IMs, compute-units in the LSTM layer and across timesteps. The pipeline is shown in Fig. 52(a) and the control flow and computation of each stage is shown in Fig. 52(b). Since some stages have variable latency, traditional strategies are not sufficient for sustaining the pipeline. Hence, they propose a two-level controller design that intelligently chooses the computations that can be parallelized based on the data-dependencies. In the "controller-state D", the computations of stages 2, 4 and 5 at time t hap-



x				w				Input of up-down counter						Approximate result	Exact result
x_3	x_2	x_1	x_0	w_3	w_2	w_1	w_0	c_6	c_5	c_4	c_3	c_2	c_1	$\frac{4}{8} = 0.5$	$\frac{30}{64} = 0.46$
0	1	0	1	0	1	1	0	1	1	0	1	1	1		
Initial value: Down counter								Initial value : Up-down counter							
6								0							

x				w				Input of up-down counter			Approximate result	Exact result
x_3	x_2	x_1	x_0	w_3	w_2	w_1	w_0	c_3	c_2	c_1	$\frac{4}{8} = 0.5$	$\frac{30}{64} = 0.46$
0	1	0	1	0	1	1	0	1	1	1		
Initial value : Down counter								Initial value : Up-down counter				
3								3				

Fig. 51. (a) IM proposed by Sim et al. [107] (b) Modified IM proposed by Azari et al. [36].



(b)

>1 cycles	1 cycle	>1 cycles	1 cycle	>1 cycles	1 cycle	>1 cycles
Stage 1(t) full	Stage 2(t) $f_j(t), g_j(t), i_j(t)$	Stage 3(t) $c_j(t)$	Stage 2(t): $f_{j+1}(t), g_{j+1}(t), i_{j+1}(t)$ *Stage* 4(t) : $o_j(t)$ *Stage* 5(t) : $tanh_j(t)$	Stage 6(t) ⟶ Stage 1(t+1) $h_j(t), partial$ Stage 3(t) $c_{j+1}(t)$	Stage 5(t) $tanh_n(t)$	Stage 6(t) $h_n(t)$
Controller states: State A	State B	State C	State D	State E	State F	State G

Fig. 52. (a) Stages of the proposed LSTM pipeline (b) Working of the proposed pipeline [36].

pen parallelly. In the "controller-state E", the computations in stage-6 (time t) and stage-1 (time $t+1$) are overlapped with those of stage-3 (time t), because they are mutually independent. These optimizations overcome the variable latency limitation of IM and also hide the latency of MVMs and memory access.

They further develop a model, which estimates the performance of their design as a function of the hidden nodes and timesteps. They compare their design with two designs. (1) An implementation using floating-point computations. Here, mean squared error is computed between the hidden-states and memory-states with their design and floating-point computations. This helps in studying error propagation in the RNN. (2) An implementation with exact fixed-point computations. Its output accuracy is compared with that of the proposed design to ascertain the optimal bit-width for balancing various optimization metrics. To save power, they transition SRAMs to sleep-mode and clock-gate the secondary-controllers and compute-

Table 9
A classification of similarity/significance based techniques.

Category	Reference
Leveraging similarity between	Input-vectors [11,28], quantized input-vectors [12] and input-elements [12,37]
Measuring the similarity between the encoder state before and after processing a symbol	Using cosine distance [18] and L2-norm distance [18,28]
Predicting input similarity using	A BNN [11], random forest, logistic regression and a shallow NN [18]
Optimizations based on similarity/significance	Reusing the results of similar previous symbols [11,12,37], skipping unimportant symbols [18,28], reducing RNN state dimension for relati, not loading unimportant rows [61]
Bounding accuracy loss due to skipping of too many consecutive timesteps	Placing an upper limit on the number of consecutive timesteps skipped [37], bounding sum of error in consecutively skipped timesteps to below a threshold [11]

units. Their technique provides high performance, area and energy efficiency.

5.5. Techniques for leveraging similarity and significance information

We now discuss works that exploit similarity and significance information. Table 9 provides an overview of these techniques.

Silfa et al. [11] (2019) present an approximate memoization technique for reducing the number of computations in RNN. In an RNN, neurons are executed repeatedly for processing the elements of an input. They note that since the inputs to a neuron in successive timesteps have high similarity, the corresponding outputs also show high similarity. Also, relative errors of 30% to 50% in outputs of neurons have an insignificant impact on RNN accuracy. Based on this, they propose dynamically memoizing the neuron outputs in a buffer. When the subsequent output is likely to be highly similar to a previous output, the neuron's output is fetched from the "memoization buffer". This avoids the need for recomputation and associated memory accesses.

They note that similar inputs need not generate similar outputs because the inputs are multiplied with large weights. Hence, to predict whether the subsequent neuron output may be almost the same as a previous output, they use a BNN that is equivalent to the original FC NN. The weights of BNN are obtained by binarizing the weights of RNN. Thus, by using BNN, the impact of both weights and inputs can be taken into account. The reason for using BNNs is that its output is highly correlated with that of the original RNN. In other words, if the output of a binarized neuron differs from a previous output by a small amount, then the full-precision neuron is also highly likely to show small changes. Based on this, memoization can be applied. Further, the compute and memory costs of BNN are extremely low.

Their technique works as follows. The recent output of full-precision neuron R and its corresponding binary neuron R_b are stored and they are denoted as Y^m and Y_b^m, respectively. At each timestep t, first R_b is evaluated for obtaining Y_b^t. Then, the magnitude of relative difference ϵ_b^t is computed as $|(Y_b^m - Y_b^t)/Y_b^t|$. If this is lower than a threshold, BNN outputs are similar and hence, the outputs of "full-precision neuron" are likely to be similar as well. Hence, the memoized output Y^m is reused as the output of neuron R at timestep t. However, if ϵ_b^t exceeds the threshold, then "full-precision" output Y^t is computed and "memoization buffer" is updated. Thus, although the BNN is always computed, the large RNN is computed selectively. Since consecutively skipping many timesteps leads to significant error, they record the sum of the relative difference in consecutive timesteps when skipping is done. If this sum exceeds a threshold, no skipping is done in the next timestep. Their technique significantly reduces energy consumption and the number of computations with negligible loss of accuracy.

Riera et al. [12] (2018) present a technique for exploiting input similarity to accelerate RNNs. They note that in successive executions of RNNs, the relative difference in inputs is small. However, due to the uniqueness of IEEE-754 floating-point format, the bit representations of two similar values may not be similar. To increase the similarity between executions, they perform linear quantization of inputs of recurrent layers. For a layer L and input x_i, quantized value is computed as $Q_C = round(x_i/step_L) \times step_L$, where $step_L = range/numberOfClusters$. The *range* of inputs of a layer is computed by profiling the training dataset. Q_C is the "cluster centroid" that is nearest to the original input value.

For exploiting the input similarity, they cache the last output of neuron. Then, no computation is required for unmodified inputs. Assume that the inputs to a RNN in the first execution are p_1, q_1 and r_1 and that in the second execution are p_2, q_2 and r_2. The corresponding weights are w_p, w_q and w_r, and the outputs in two executions are z_1 and z_2, respectively. Then, $z_1 = w_p * p_1 + w_q * q_1 + w_r * r_1$ and $z_2 = w_p * p_2 + w_q * q_2 + w_r * r_2$. But if the first two inputs are same in both the runs, then, we have $z_2 = z_1 + w_r * (r_2 - r_1)$. For this example, just one weight (w_r) is accessed from the memory and the value $(r_2 - r_1)$ is reused for each neuron in the layer. Thus, input similarity brings large reduction in the number of computations and memory accesses.

As for the choice of number of clusters, use of a small number of clusters leads to high similarity by virtue of constraining the inputs to a small range. However, it also increases the accuracy loss. They find that the use of 16 clusters achieves a balance between accuracy-drop and computation-saving. For the EESEN RNN benchmark, they apply quantization in all the layers except the last FC layer, since applying quantization in this layer leads to large accuracy loss. Also, this layer has very few neurons and hence, the scope of reducing computations is also very small. In EESEN, for most of the recurrent layers, over 50% computations are avoided for just 0.18% accuracy loss.

They further discuss an accelerator architecture that exploits reuse effectively. For an FC layer, the first execution proceeds without exploiting reuse. In this execution, the index of input quantization is saved in a buffer to be used in later executions. In the later executions, the first input and its index in the last execution are fetched from the buffer. Then, this input is quantized and its centroid is fetched using the index. If this "quantized input" equals the centroid, then the input is skipped. Otherwise, the input is deemed to have changed and hence, correction of all the neurons is performed using the weights corresponding to that input. This process is performed repeatedly for correcting all the neurons for every input that has changed.

Since the four gates of an LSTM cell work independently as FC layers, a recurrent layer works similar to an FC layer. However, RNNs also have important differences from CNNs or MLPs. Every recurrent layer is executed successively for every symbol in the input sequence before the subsequent layer executes. Hence, in RNNs, redundant computations show higher temporal locality. Also, all the four gates of an LSTM cell work with the same inputs. As such, for an unchanged input, computation reuse can be leveraged in all the four gates. Their technique brings large speedup and energy-saving.

Jo et al. [37] (2019) present an approximate computing technique for improving the efficiency of LSTM-based speech recognition system. They note that in speech recognition systems, even a single character input is split into multiple "consecutive sections" with high-similarity values. Hence, successive LSTM units handle similar features. They define the similarity score between consecutive input vectors (x_{t-1} and x_t) for computation of NZ elements in x_t. Since high similarity in inputs leads to high similarity in outputs if the similarity score exceeds a threshold Θ_1, the LSTM computation at time t is deactivated. Also, previous output is transferred as the current output, that is, $h_k = h_{k-1}$ and

$c_k = c_{k-1}$. This simple strategy can reduce 10% operations at the cost of increasing WER from 9.3% to 10.3%.

Let $Skip_{\max}$ denote the maximum number of successively skipped cells. The above scheme has $Skip_{\max} = \infty$. A challenge in the above scheme is that the use of $Skip_{\max} = \inf$ leads to the accumulation of errors from the consecutive skipping, which harms the output quality greatly. To mitigate this issue, they propose limiting $Skip_{\max}$ to a small value, such as two or three. For example, if $Skip_{\max} = 3$, then after skipping 3 cells, the next cell is always computed, irrespective of its similarity score. This strategy bounds the accumulation of errors and for a fixed loss in final output quality, it allows skipping more cells. For example, with $Skip_{\max} = 2$, 45% LSTM operations can be skipped while increasing the WER to 11.3%.

To further improve the benefits, they propose a technique for skipping operations at the element level. When $\Theta_1 > Similarity > \Theta_2$, where Θ_2 is slightly lower than Θ_1, they use delta-RNN algorithm [108](2017) for performing the LSTM operation. Here, the previous input is subtracted from the current input to create the input of the cell. The input sparsity is further enhanced by masking the small differences. Since this skipping is performed only for the cells with similarity above Θ_2, EW differences between successive input vectors are generally much smaller than the values in the original input itself. This allows reducing the bitwidth of these cells to 8bits, whereas the bitwidth of remaining cells is 16bits. This further lowers the computation complexity and allows dynamic trade-off with accuracy. For a bidirectional LSTM network, their technique achieves a large improvement in energy efficiency with only a small increase in WER.

Sen et al. [28] (2018) present two approximate computing techniques for LSTMs: "timestep skipping technique" (TST) and "state reduction technique" (SRT). TST is founded on the fact that some symbols of a sequence have negligible impact on the LSTM state. For instance, an LSTM can produce a video caption without seeing all the frames of a video. Similarly, not all words of a sentence need to be processed to find its summary or the sentiment. Based on this, unimportant words/symbols/timesteps can be bypassed and only significant symbols are passed to the LSTM. For determining the significance of a symbol, two metrics are used: InputDifference and StateImpact. The InputDifference metric skips an input symbol that is akin to the preceding symbol since such a symbol does not provide additional information to the encoder. The StateImpact metric computes L2 norm of the change in "encoder state" before and after consuming a symbol. Based on the quality requirement, only inputs with a high magnitude of L2 norm are passed on.

SRT is founded on the observation that every input sequence does not have high complexity that needs the entire "dimensionality" of the LSTM state. Based on this, SRT reduces the state's dimension by keeping only the important state elements for a specific input sequence. This effectively reduces the size of weight, which lowers MMM's overhead in the computation of different gates of the LSTM. This lowers the latency of evaluating every timestep of LSTM. For "sequence-to-sequence models", they use SRT in the decoder. SRT employs a state-clipper at the interface between the encoder and decoder for identifying the group of important indices. Thus, the hidden state size in the encoder remains at its peak value during the entire encoding process, which generates a "complete context vector". The state-clipper slices this vector to contain only important elements. The state-clipper also slices the decoder weight matrices depending on the indices of the important elements of the "context vector".

In order to prune unimportant indices, two thresholding schemes are used. In "relative thresholding", elements that are much smaller than the highest value in context vector are pruned. In "absolute thresholding", elements below a threshold are pruned. For sequence-to-sequence models, state-clipper forms a union of salient elements in all input sequences in a batch. Due to this, the overall state size could be larger than the needed for a single sequence. Yet, batching provides the benefit of amortizing the slicing-overhead over multiple sentences in a batch. For models having several layers in the encoder and decoder, SRT finds a different group of salient indices for each layer. Their techniques bring large improvement in performance with minor loss of accuracy.

Tao et al. [18] (2019) present a timestep-skipping technique for RNNs. They first discuss an ideal predictor that forecasts whether the present input element will cause a substantial change in the hidden state. To measure the change between the hidden states at successive time instances h_{t-1} and h_t, they use two distance metrics: cosine distance and L2 norm distance, which measure the orientation and magnitude, respectively. Based on these metrics, inputs that cause a change larger than a threshold are bypassed and the remaining are sent to the bigger RNN model. The threshold value is found based on the analysis of statistical distributions of these distances for training datasets. They evaluate this ideal predictor on two RNN models, trained for IMDB and SST datasets, respectively. They find that cosine distance is a better metric than L2 norm distance since the former enables aggressive skipping with negligible accuracy-loss. This predictor allows skipping 80% and 45% time-steps in IMDB and SST datasets, respectively.

To develop a realistic predictor, they first create a training dataset for this predictor. For this, the dataset used for training the original RNN is used and each sequence in it is run through the RNN model. Then, the hidden states after each timestep are collected. They test three different predictors: random forest, logistic regression and a shallow NN (having two hidden layers, each with 10 units). They find that the shallow NN with cosine distance achieves minimal accuracy loss and largest reduction in computation. The memory footprint of this NN predictor is only 1/40 times that of the baseline RNN model with one hidden layer of 100 units, yet it reduces computations by 25% with negligible loss of accuracy.

6. Future research challenges

Nearly all the works have evaluated the proposed techniques based on performance and energy efficiency metrics only. In many mission-critical environments such as defense, medical, finance, space and self-driving cars, reliability and security are of paramount importance. In order to enable the adoption of RNNs in these scenarios, a thorough investigation of reliability and security of RNN accelerators is required [109]. For example, research into the explainability of RNNs will allow effectively mitigating fault-propagation across the RNN layers. It will also allow closing the side-channels that may leak information about RNN hyperparameters and topology.

Data-movement between the processor and off-chip memory dissipates a hundred times higher energy than a floating-point operation [110]. Due to this, von-Neumann style processors are becoming bottlenecked by the data-movement overheads. Processing-in-memory is a promising solution to overcome this challenge. Researchers have recently shown processing-in-memory capabilities in both conventional memories such as SRAM/DRAM and emerging memories such as memristor/spintronic memories [111,112]. The design of processing-in-memory based RNN accelerators will be highly interesting since they have the potential to bring substantial improvement in performance and energy efficiency.

In recent years, several improvements to LSTMs have been proposed. For example, LSTMs with peephole connections are able to count and measure the time between events. The peephole connections enable all the gates to see the current cell state even when the output gate is closed [113]. The downside of adding peephole connections is that they increase the memory and compute overhead. Another important trend in the design of LSTM networks is increasing depth and size of LSTM models, e.g., the DeepSpeech2 model has more than twice the depth and ten times the size of the original DeepSpeech model [114,115]. This, however, leads to high memory and compute overheads and also increases the inference latency. Further, large LSTMs are likely to overfit and difficult to regularize. To address these challenges, Dai et al. [115] propose a new approach for increasing the depth inside LSTM cells. They present

a hidden-layer LSTM whose control gates are enhanced by addition of hidden layers. They use a "grow-and-prune" training approach which consists of a growth phase and a prune phase. In the growth-phase, the network is allowed to increase the number of neurons, connections and feature maps, whereas in the pruning-phase, low-magnitude weights are iteratively pruned. Compared to traditional stacked LSTM cells, higher accuracy is achieved by stacking fewer hidden-layer LSTM cells. This design also has less number of parameters and floating-point operations.

Several works [33,34,47,76–79] have evaluated CNN + RNN hybrid models such as "long-term recurrent convolutional networks" (LRCN, used for visual recognition and description) or SHOWTELL (used for image captioning). In fact, real-world tasks may require not only different types of neural networks, but even other machine learning models. Different types of NN have distinct characteristics. For example, the dominant compute-pattern of CONV is MMM, whereas that of FC and RNN is MVM. Due to these and other differences, hardware accelerators designed for a single class of neural networks may not be optimal for CNN + RNN hybrid models. Of the papers reviewed in this survey, only few papers have separately evaluated their techniques on both CNN and RNN models [1,3,12,21,39,71]. Evidently, next-generation architectures must be examined based on not just one deep-learning model, but a range of models.

While CNN has a limited receptive field, RNN has an infinite receptive field, which means that RNN can look at an infinite number of symbols in the input sequence. The crucial limitation of RNN, however, is its sequentiality, which means that RNN processes an input step-by-step. Recent research ideas such as attention mechanism seeks to overcome these limitations. Since attention offers constant path length, it overcomes both the sequentiality problem of RNN and the limited receptive field problem of CNN. Researchers have recently proposed using attention mechanism to focus on the specific data elements, viz., the hidden state outputs of LSTMs. For example, Google's "neural machine translation" technique [4] places an attention network between encoding and decoding LSTM layers and achieves high accuracy. Similarly, LSTM with attention model has been successfully used for solving complex problems, such as hide-and-seek reinforcement learning. Deep-learning is, after all, a very rapidly moving research field and as algorithms evolve, hardware architectures must also evolve. Computer architects, no doubt, have "miles to go" to extract the last bit of performance from the hardware for emerging deep-learning models.

7. Conclusion

In this paper, we presented a survey of hardware accelerators for RNNs, techniques for optimizing them and techniques for simplifying RNN algorithms/architectures. We provided a bird eye's view of the field. We organized the works on several categories to show their similarities and differences. We concluded this paper with a brief discussion of future research challenges.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] N.P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, et al., In-datacenter performance analysis of a tensor processing unit, in: International Symposium on Computer Architecture, 2017, pp. 1–12.

[2] N.P. Jouppi, D.H. Yoon, G. Kurian, S. Li, N. Patil, J. Laudon, C. Young, D. Patterson, A domain-specific supercomputer for training deep neural networks, Commun. ACM 63 (7) (2020) 67–78.

[3] J. Fowers, K. Ovtcharov, M. Papamichael, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, L. Adams, M. Ghandi, et al., A configurable cloud-scale DNN processor for real-time AI, in: International Symposium on Computer Architecture (ISCA), 2018, pp. 1–14.

[4] Y. Wu, M. Schuster, Z. Chen, Q.V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, et al., Google'S neural machine translation system: bridging the gap between human and machine translation, arXiv preprint arXiv:1609.08144(2016).

[5] Y. Sun, A. Ben Ahmed, H. Amano, Acceleration of deep recurrent neural networks with an FPGA cluster, in: International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies, 2019, p. 18.

[6] J. Puigcerver, Are multidimensional recurrent layers really necessary for handwritten text recognition? in: 2017 14th IAPR International Conference on Document Analysis and Recognition (ICDAR), 1, 2017, pp. 67–72.

[7] U. Gupta, B. Reagen, L. Pentecost, M. Donato, T. Tambe, A.M. Rush, G.-Y. Wei, D. Brooks, MASR: A Modular Accelerator for Sparse RNNs, in: 2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT), 2019, pp. 1–14.

[8] S. Mittal, A survey of FPGA-based accelerators for convolutional neural networks, Neur. Comput. Appl. 32 (4) (2020) 1109–1139.

[9] S. Mittal, P. Rajput, S. Subramoney, A Survey of Deep Learning on CPUs: Opportunities and Co-optimizations, Technical Report, IIT Roorkee, 2020.

[10] Y. Zhang, C. Wang, L. Gong, Y. Lu, F. Sun, C. Xu, X. Li, X. Zhou, Implementation and Optimization of the Accelerator Based on FPGA Hardware for LSTM Network, in: IEEE International Symposium on Parallel and Distributed Processing with Applications and 2017 IEEE International Conference on Ubiquitous Computing and Communications (ISPA/IUCC), 2017, pp. 614–621.

[11] F. Silfa, G. Dot, J.-M. Arnau, A. Gonzàlez, Neuron-Level Fuzzy Memoization in RNNs, in: International Symposium on Microarchitecture, 2019, pp. 782–793.

[12] M. Riera, J.-M. Arnau, A. González, Computation reuse in DNNs by exploiting input similarity, in: International Symposium on Computer Architecture (ISCA), 2018, pp. 57–68.

[13] V. Rybalkin, A. Pappalardo, M.M. Ghaffar, G. Gambardella, N. Wehn, M. Blott, FINN-L: Library extensions and design trade-off analysis for variable precision LSTM networks on FPGAs, in: 2018 28th International Conference on Field Programmable Logic and Applications (FPL), IEEE, 2018, pp. 89–897.

[14] Q. Li, X. Zhang, J. Xiong, W.-m. Hwu, D. Chen, Implementing neural machine translation with bi-directional GRU and attention mechanism on FPGAs using HLS, in: Asia and South Pacific Design Automation Conference, 2019, pp. 693–698.

[15] J. Appleyard, T. Kocisky, P. Blunsom, Optimizing performance of recurrent neural networks on GPUs, arXiv preprint arXiv:1604.01946(2016).

[16] E. Nurvitadhi, J. Sim, D. Sheffield, A. Mishra, S. Krishnan, D. Marr, Accelerating recurrent neural networks in analytics servers: Comparison of FPGA, CPU, GPU, and ASIC, in: 2016 26th International Conference on Field Programmable Logic and Applications (FPL), 2016, pp. 1–4.

[17] K. Cho, B. Van Merriënboer, D. Bahdanau, Y. Bengio, On the properties of neural machine translation: encoder-decoder approaches, arXiv preprint arXiv:1409.1259(2014).

[18] J. Tao, U. Thakker, G. Dasika, J. Beu, Skipping RNN State Updates without Retraining the Original Model, in: 1st Workshop on Machine Learning on Edge in Sensor Systems, 2019, pp. 31–36.

[19] A. Ardakani, Z. Ji, W.J. Gross, Learning to skip ineffectual recurrent computations in LSTMs, in: Design, Automation & Test in Europe Conference & Exhibition (DATE), 2019, pp. 1427–1432.

[20] S. Li, C. Wu, H. Li, B. Li, Y. Wang, Q. Qiu, FPGA acceleration of recurrent neural network based language model, in: Annual International Symposium on Field-Programmable Custom Computing Machines, 2015, pp. 111–118.

[21] C. Meng, M. Sun, J. Yang, M. Qiu, Y. Gu, Training deeper models by GPU memory optimization on TensorFlow, in: Proc. of ML Systems Workshop in NIPS, 2017.

[22] M. Zhu, J. Clemons, J. Pool, M. Rhu, S.W. Keckler, Y. Xie, Structurally sparsified backward propagation for faster long short-term memory training, arXiv preprint arXiv:1806.00512(2018).

[23] H. Yin, G. Chen, Y. Li, S. Che, W. Zhang, N.K. Jha, Hardware-Guided symbiotic training for compact, accurate, yet execution-Efficient LSTM, arXiv preprint arXiv:1901.10997(2019).

[24] B. Zheng, A. Tiwari, N. Vijaykumar, G. Pekhimenko, EcoRNN: efficient computing of LSTM RNN training on GPUs, arXiv preprint arXiv:1805.08899(2018).

[25] F. Khorasani, H.A. Esfeden, N. Abu-Ghazaleh, V. Sarkar, In-register parameter caching for dynamic neural nets with virtual persistent processor specialization, in: Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2018, pp. 377–389.

[26] K. Hwang, W. Sung, Single stream parallelization of generalized LSTM-like RNNs on a GPU, in: IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), 2015, pp. 1047–1051.

[27] Z. Jia, B. Tillman, M. Maggioni, D.P. Scarpazza, Dissecting the graphcore ipu architecture via microbenchmarking, arXiv preprint arXiv:1912.03413(2019).

[28] S. Sen, A. Raghunathan, Approximate computing for long short term memory (LSTM) neural networks, IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. 37 (11) (2018) 2266–2276.

[29] P. Ouyang, S. Yin, S. Wei, A fast and power efficient architecture to parallelize LSTM based RNN for cognitive intelligence applications, in: Design Automation Conference, 2017, pp. 1–6.

[30] F. Conti, L. Cavigelli, G. Paulin, I. Susmelj, L. Benini, Chipmunk: A systolically scalable 0.9 mm 2, 3.08 Gop/s/mW@ 1.2 mW accelerator for near-sensor recurrent neural network inference, in: IEEE Custom Integrated Circuits Conference (CICC), 2018, pp. 1–4.

[31] J. Park, J. Kung, W. Yi, J.-J. Kim, Maximizing system performance by balancing computation loads in LSTM accelerators, in: Design, Automation & Test in Europe Conference & Exhibition (DATE), 2018, pp. 7–12.

[32] Z. Wang, J. Lin, Z. Wang, Hardware-oriented compression of long short-term memory for efficient inference, IEEE Signal Process. Lett. 25 (7) (2018) 984–988.

[33] F. Silfa, G. Dot, J.-M. Arnau, A. Gonzàlez, E-PUR: an energy-efficient processing unit for recurrent neural networks, in: International Conference on Parallel Architectures and Compilation Techniques, 2018, pp. 1–12.

[34] F. Silfa, J.-M. Arnau, A. Gonzàlez, Boosting LSTM performance through dynamic precision selection, arXiv preprint arXiv:1911.04244(2019).

[35] J. Kung, J. Park, S. Park, J.-J. Kim, Peregrine: A flexible hardware accelerator for LSTM with limited synaptic connection patterns, in: Design Automation Conference, 2019, pp. 1–6.

[36] E. Azari, S. Vrudhula, ELSA: A Throughput-Optimized Design of an LSTM accelerator for energy-Constrained devices, ACM Trans. Embedd. Comput. Syst. (TECS) 19 (1) (2020) 1–21.

[37] J. Jo, J. Kung, S. Lee, Y. Lee, Similarity-based LSTM architecture for energy-efficient edge-level speech recognition, in: IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED), 2019, pp. 1–6.

[38] V. Rybalkin, N. Wehn, M.R. Yousefi, D. Stricker, Hardware architecture of bidirectional long short-term memory neural network for optical character recognition, in: Design, Automation & Test in Europe, 2017, pp. 1394–1399.

[39] Y. Guan, H. Liang, N. Xu, W. Wang, S. Shi, X. Chen, G. Sun, W. Zhang, J. Cong, FP-DNN: an automated framework for mapping deep neural networks onto FPGAs with RTL-HLS hybrid templates, in: Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2017, pp. 152–159.

[40] S. Han, J. Kang, H. Mao, Y. Hu, X. Li, Y. Li, D. Xie, H. Luo, S. Yao, Y. Wang, et al., ESE: Efficient speech recognition engine with sparse LSTM on FPGA, in: Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, ACM, 2017, pp. 75–84.

[41] A.X.M. Chang, E. Culurciello, Hardware accelerators for recurrent neural networks on FPGA, in: IEEE International Symposium on Circuits and Systems (ISCAS), 2017, pp. 1–4.

[42] M. Rizakis, S.I. Venieris, A. Kouris, C.-S. Bouganis, Approximate FPGA-based LSTMs under computation time constraints, in: International Symposium on Applied Reconfigurable Computing, 2018, pp. 3–15.

[43] S. Cao, C. Zhang, Z. Yao, W. Xiao, L. Nie, D. Zhan, Y. Liu, M. Wu, L. Zhang, Efficient and effective sparse LSTM on FPGA with Bank-Balanced Sparsity, in: International Symposium on Field-Programmable Gate Arrays, 2019, pp. 63–72.

[44] E. Nurvitadhi, A. Boutros, P. Budhkar, A. Jafari, D. Kwon, D. Sheffield, A. Prabhakaran, K. Gururaj, P. Appana, M. Naik, Scalable low-latency persistent neural machine translation on CPU server with multiple FPGAs, in: International Conference on Field-Programmable Technology (ICFPT), 2019, pp. 307–310.

[45] A.X.M. Chang, B. Martini, E. Culurciello, Recurrent neural networks hardware implementation on FPGA, arXiv preprint arXiv:1511.05552(2015).

[46] Y. Guan, Z. Yuan, G. Sun, J. Cong, FPGA-based accelerator for long short-term memory recurrent neural networks, in: Asia and South Pacific Design Automation Conference (ASP-DAC), 2017, pp. 629–634.

[47] S. Zeng, K. Guo, S. Fang, J. Kang, D. Xie, Y. Shan, Y. Wang, H. Yang, An efficient reconfigurable framework for general purpose CNN-RNN models on FPGAs, in: 2018 IEEE 23rd International Conference on Digital Signal Processing (DSP), 2018, pp. 1–5.

[48] C. Gao, F. Zhang, FPGA-based accelerator for independently recurrent neural network, in: 2018 IEEE 4th International Conference on Computer and Communications (ICCC), 2018, pp. 2075–2080.

[49] Z. Chen, A. Howe, H.T. Blair, J. Cong, CLINK: compact LSTM inference kernel for energy efficient neurofeedback devices, in: International Symposium on Low Power Electronics and Design, 2018, p. 2.

[50] T. Mealey, T.M. Taha, Accelerating inference in long short-term memory neural networks, in: IEEE National Aerospace and Electronics Conference (NAECON), 2018, pp. 382–390.

[51] Z. Sun, Y. Zhu, Y. Zheng, H. Wu, Z. Cao, P. Xiong, J. Hou, T. Huang, Z. Que, FPGA acceleration of LSTM based on data for test flight, in: 2018 IEEE International Conference on Smart Cloud (SmartCloud), 2018, pp. 1–6.

[52] M. Wang, Z. Wang, J. Lu, J. Lin, Z. Wang, E-LSTM: an efficient hardware architecture for long short-Term memory, IEEE J. Emerg. Sel. Top. Circuits Syst. (2019).

[53] E. Bank-Tavakoli, S.A. Ghasemzadeh, M. Kamal, A. Afzali-Kusha, M. Pedram, POLAR: A Pipelined/Overlapped FPGA-Based LSTM accelerator, IEEE Trans. Very Large Scale Integr. VLSI Syst. (2019).

[54] P. Dong, S. Wang, W. Niu, C. Zhang, S. Lin, Z. Li, Y. Gong, B. Ren, X. Lin, Y. Wang, et al., RTMobile: beyond real-Time mobile acceleration of RNNs for speech recognition, arXiv preprint arXiv:2002.11474(2020).

[55] S. Wang, P. Lin, R. Hu, H. Wang, J. He, Q. Huang, S. Chang, Acceleration of LSTM with structured pruning method on FPGA, IEEE Access 7 (2019) 62930–62937.

[56] D. Diamantopoulos, C. Hagleitner, A system-level transprecision FPGA accelerator for blstm using on-chip memory reshaping, in: 2018 International Conference on Field-Programmable Technology (FPT), 2018, pp. 338–341.

[57] G. Diamos, S. Sengupta, B. Catanzaro, M. Chrzanowski, A. Coates, E. Elsen, J. Engel, A. Hannun, S. Satheesh, Persistent RNNs: stashing recurrent weights on-chip, in: International Conference on Machine Learning, 2016, pp. 2024–2033.

[58] Q. Cao, N. Balasubramanian, A. Balasubramanian, MobiRNN: efficient recurrent neural network execution on mobile GPU, in: International Workshop on Deep Learning for Mobile Systems and Applications, 2017, pp. 1–6.

[59] P. Gao, L. Yu, Y. Wu, J. Li, Low latency RNN inference with cellular batching, in: Proceedings of the Thirteenth EuroSys Conference, 2018, p. 31.

[60] F. Zhu, J. Pool, M. Andersch, J. Appleyard, F. Xie, Sparse persistent RNNs: squeezing large recurrent networks on-chip, arXiv preprint arXiv:1804.10223(2018).

[61] X. Zhang, C. Xie, J. Wang, W. Zhang, X. Fu, Towards Memory Friendly Long-Short Term Memory Networks (LSTMs) on Mobile GPUs, in: International Symposium on Microarchitecture (MICRO), 2018, pp. 162–174.

[62] Z. Yao, S. Cao, W. Xiao, C. Zhang, L. Nie, Balanced sparsity for efficient DNN inference on GPU, in: AAAI Conference on Artificial Intelligence, 33, 2019, pp. 5676–5683.

[63] B. Zheng, A. Nair, Q. Wu, N. Vijaykumar, G. Pekhimenko, Ecornn: fused LSTM RNN implementation with data layout optimization, arXiv preprint arXiv:1805.08899(2018).

[64] S. Gray, A. Radford, D.P. Kingma, GPU Kernels for block-sparse weights, arXiv preprint arXiv:1711.09224(2017).

[65] M. Zhu, T. Zhang, Z. Gu, Y. Xie, Sparse tensor core: algorithm and hardware co-design for vector-wise sparse neural networks on modern GPUs, in: IEEE/ACM International Symposium on Microarchitecture, 2019, pp. 359–371.

[66] C. Gao, A. Rios-Navarro, X. Chen, T. Delbruck, S.-C. Liu, EdgeDRNN: enabling low-latency recurrent neural network edge inference, arXiv preprint arXiv:1912.12193(2019).

[67] Z. Wang, J. Lin, Z. Wang, Accelerating recurrent neural networks: amemory-efficient approach, IEEE Trans. Very Large Scale Integr. VLSI Syst. 25 (10) (2017) 2763–2775.

[68] B. Liu, H. Qin, Y. Gong, W. Ge, M. Xia, L. Shi, EERA-ASR: An energy-Efficient reconfigurable architecture for automatic speech recognition with hybrid DNN and approximate computing, IEEE Access 6 (2018) 52227–52237.

[69] S. Dey, P.D. Franzon, An Application Specific Processor Architecture with 3D Integration for Recurrent Neural Networks, in: 20th International Symposium on Quality Electronic Design (ISQED), 2019, pp. 183–190.

[70] S. Wang, Z. Li, C. Ding, B. Yuan, Q. Qiu, Y. Wang, Y. Liang, C-LSTM: Enabling efficient LSTM using structured compression techniques on FPGAs, in: International Symposium on Field-Programmable Gate Arrays, 2018, pp. 11–20.

[71] C. Ding, A. Ren, G. Yuan, X. Ma, J. Li, N. Liu, B. Yuan, Y. Wang, Structured weight matrices-based hardware accelerators in deep neural networks: FPGAs and ASICs, in: Great Lakes Symposium on VLSI, 2018, pp. 353–358.

[72] J. Liu, J. Wang, Y. Zhou, F. Liu, A cloud server oriented FPGA accelerator for LSTM recurrent neural network, IEEE Access 7 (2019) 122408–122418.

[73] Z. Li, C. Ding, S. Wang, W. Wen, Y. Zhuo, C. Liu, Q. Qiu, W. Xu, X. Lin, X. Qian, et al., E-RNN: Design optimization for efficient recurrent neural networks in FPGAs, in: 2019 IEEE International Symposium on High Performance Computer Architecture (HPCA), 2019, pp. 69–80.

[74] V. Rybalkin, N. Wehn, When massive GPU parallelism ain't enough: a novel hardware architecture of 2D-LSTM neural network, in: International Symposium on Field-Programmable Gate Arrays, 2020, pp. 111–121.

[75] M. Lee, K. Hwang, J. Park, S. Choi, S. Shin, W. Sung, FPGA-based low-power speech recognition with recurrent neural networks, in: IEEE International Workshop on Signal Processing Systems (SiPS), 2016, pp. 230–235.

[76] X. Zhang, X. Liu, A. Ramachandran, C. Zhuge, S. Tang, P. Ouyang, Z. Cheng, K. Rupnow, D. Chen, High-performance video content recognition with long-term recurrent convolutional network for FPGA, in: International Conference on Field Programmable Logic and Applications (FPL), 2017, pp. 1–4.

[77] M. Van Keirsbilck, A. Keller, X. Yang, Rethinking full connectivity in recurrent neural networks, arXiv preprint arXiv:1905.12340(2019).

[78] Z. Que, T. Nugent, S. Liu, L. Tian, X. Niu, Y. Zhu, W. Luk, Efficient weight reuse for large LSTMs, in: IEEE 30th International Conference on Application-specific Systems, Architectures and Processors (ASAP), 2160, 2019, pp. 17–24.

[79] L. Peng, W. Shi, J. Zhang, S. Irving, Exploiting model-level parallelism in recurrent neural network accelerators, in: International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC), 2019, pp. 241–248.

[80] E. Nurvitadhi, D. Kwon, A. Jafari, A. Boutros, J. Sim, P. Tomson, H. Sumbul, G. Chen, P. Knag, R. Kumar, et al., Why compete when you can work together: FPGA-ASIC integration for persistent RNNs, in: 2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2019, pp. 199–207.

[81] S. Park, J. Jang, S. Kim, S. Yoon, Energy-efficient inference accelerator for memory-augmented neural networks on an FPGA, in: 2019 Design, Automation & Test in Europe Conference & Exhibition (DATE), 2019, pp. 1587–1590.

[82] M.R. Yousefi, M.R. Soheili, T.M. Breuel, E. Kabir, D. Stricker, Binarization-free OCR for historical documents using LSTM networks, in: 2015 13th international conference on document analysis and recognition (ICDAR), IEEE, 2015, pp. 1121–1125.

[83] K. Khalil, O. Eldash, A. Kumar, M. Bayoumi, Economic LSTM approach for recurrent neural networks, IEEE Trans. Circuits Syst. II Express Briefs 66 (11) (2019) 1885–1889.

[84] I. Kouretas, V. Paliouras, Logarithmic number system for deep learning, in: International Conference on Modern Circuits and Systems Technologies (MOCAST), 2018, pp. 1–4.

[85] R. Yazdani, O. Ruwase, M. Zhang, Y. He, J.-M. Arnau, A. Gonzalez, LSTM-sharp: an adaptable, energy-Efficient hardware accelerator for long short-Term memory, arXiv preprint arXiv:1911.01258(2019).

[86] J. Wu, F. Li, Z. Chen, X. Xiang, A 3.89-GOPS/mw scalable recurrent neural network processor with improved efficiency on memory and computation, IEEE Trans. Very Large Scale Integr. VLSI Syst. 27 (12) (2019) 2939–2943.

[87] C. Gao, S. Braun, I. Kiselev, J. Anumula, T. Delbruck, S.-C. Liu, Real-time speech recognition for IoT purpose using a delta recurrent neural network accelerator, in: 2019 IEEE International Symposium on Circuits and Systems (ISCAS), IEEE, 2019, pp. 1–5.

[88] D. Liu, N. Sepulveda, M. Zheng, Artificial neural networks condensation: a strategy to facilitate adaption of machine learning in medical settings by reducing computational burden, arXiv preprint arXiv:1812.09659(2018).

[89] J.C. Ferreira, J. Fonseca, An FPGA implementation of a long short-term memory neural network, in: International Conference on ReConFigurable Computing and FPGAs (ReConFig), 2016, pp. 1–8.
[90] K. Chen, L. Huang, M. Li, X. Zeng, Y. Fan, A compact and configurable long short--term memory neural network hardware architecture, in: IEEE International Conference on Image Processing (ICIP), 2018, pp. 4168–4172.
[91] BLAS (Basic Linear Algebra Subprograms), (http://www.netlib.org/blas/).
[92] S. Mittal, J.S. Vetter, D. Li, Improving energy efficiency of Embedded DRAM Caches for High-end Computing Systems, in: International ACM Symposium on High Performance Parallel and Distributing Computing (HPDC), 2014, pp. 99–110.
[93] S.S. Manohar, H.K. Kapoor, Dynamic reconfiguration of embedded-DRAM caches employing zero data detection based refresh optimisation, J. Syst. Archit. 100 (2019) 101648.
[94] S. Mittal, A survey of techniques for managing and leveraging caches in GPUs, J. Circu. Syst. Comput. (JCSC) 23 (8) (2014).
[95] Nate Oh, The NVIDIA titan V deep learning deep dive: it's all about the tensor cores, (https://www.anandtech.com/show/12673/titan-v-deep-learning-deep-dive/8).
[96] S. Mittal, J. Vetter, A survey of architectural approaches for data compression in cache and main memory systems, IEEE Transactions on Parallel and Distributed Systems (TPDS) 27 (5) (2016) 1524–1536.
[97] GraphCore Benchmarks, (https://cdn2.hubspot.net/hubfs/729091/NIPS2017/NIPS201720-20benchmarks20final.pdf).
[98] J. Volder, The CORDIC computing technique, in: Papers presented at the March 3–5, 1959, western joint computer conference, 1959, pp. 257–261.
[99] B. Li, E. Zhou, B. Huang, J. Duan, Y. Wang, N. Xu, J. Zhang, H. Yang, Large scale recurrent neural network on GPU, in: International Joint Conference on Neural Networks (IJCNN), 2014, pp. 4062–4069.
[100] S. Mittal, A survey of techniques for dynamic branch prediction, Concurrency and Computation: Practice and Experience 31 (1) (2018) e4666.
[101] S. Mittal, A survey on optimized implementation of deep learning models on the NVIDIA jetson platform, J. Syst. Archit. 97 (2019) 428–442.
[102] RenderScript Overview, 2020, (https://developer.android.com/guide/topics/renderscript/compute).
[103] S. Mittal, A survey of techniques for architecting and managing GPU register file, IEEE Transactions on Parallel and Distributed Systems (TPDS) (2016).
[104] S. Mittal, S. Vaishay, A survey of techniques for optimizing deep learning on GPUs, J. Syst. Archit. 99 (2019) 101635.
[105] S. Mittal, A survey of techniques for approximate computing, ACM Comput Surv 48 (4) (2016) 62:1–62:33.
[106] X. Yang, P. Molchanov, J. Kautz, Making convolutional networks recurrent for visual sequence learning, in: IEEE Conference on Computer Vision and Pattern Recognition, 2018, pp. 6469–6478.
[107] H. Sim, J. Lee, A new stochastic computing multiplier with application to deep convolutional neural networks, in: Proceedings of the 54th Annual Design Automation Conference 2017, 2017, pp. 1–6.
[108] D. Neil, J.H. Lee, T. Delbruck, S.-C. Liu, Delta networks for optimized recurrent network computation, in: International Conference on Machine Learning-Volume 70, JMLR. org, 2017, pp. 2584–2593.
[109] S. Mittal, A survey on modeling and improving reliability of DNN algorithms and accelerators, J. Syst. Archit. 104 (2020) 101689.
[110] S. Mittal, S. Nag, A survey of encoding techniques for reducing data-movement energy, J. Syst. Archit. 97 (2019) 373–396.
[111] S. Umesh, S. Mittal, A survey of spintronic architectures for processing-in-Memory and neural networks, J. Syst. Archit. 97 (2019) 349–372.
[112] S. Mittal, A survey on applications and architectural-optimizations of Micron's automata processor, J. Syst. Archit. 98 (2019) 135–164.
[113] 5 Types of LSTM Recurrent Neural Networks and What to Do With Them, (https://blog.exxactcorp.com/5-types-lstm-recurrent-neural-network/).
[114] (https://deepspeech.readthedocs.io/en/v0.7.4/).
[115] X. Dai, H. Yin, N.K. Jha, Grow and prune compact, fast, and accurate LSTMs, IEEE Trans. Comput. 69 (3) (2019) 441–452.

Sparsh Mittal received the B.Tech. degree in electronics and communications engineering from IIT, Roorkee, India and the Ph.D. degree in computer engineering from Iowa State University (ISU), USA. He worked as a post-doctoral research associate at Oak Ridge National Lab (ORNL), USA for 3 years and as an assistant professor at IIT Hyderabad, India. He is currently working as an assistant professor at IIT Roorkee, India. He was the graduating topper of his batch in B.Tech. and has received fellowship from ISU and performance award from ORNL. Sparsh has published more than 85 papers in top conferences and journals. His research has been covered by several technical news websites, e.g. Phys.org, InsideHPC, Primeur Magazine, StorageSearch, Data- Compression.info, TechEnablement, ScientificComputing, SemiEngineering, ReRAM forum and HPCWire. His research interests include accelerators for neural networks, architectures for machine learning, non-volatile memory, and low-power computing.

Sumanth Umesh is presently pursuing B.Tech. degree in the Department of Electrical Engineering at IIT Jodhpur, India. His research interests include Spintronic Memories, low-power computing and Hardware Architecture for Machine Learning.