RNNFast: An Accelerator for Recurrent Neural Networks Using Domain-Wall Memory

MOHAMMAD HOSSEIN SAMAVATIAN, The Ohio State University, USA
ANYS BACHA, University of Michigan, USA
LI ZHOU and RADU TEODORESCU, The Ohio State University, USA

Recurrent Neural Networks (RNNs) are an important class of neural networks designed to retain and incorporate context into current decisions. RNNs are particularly well suited for machine learning problems in which context is important, such as speech recognition and language translation.

This work presents RNNFast, a hardware accelerator for RNNs that leverages an emerging class of nonvolatile memory called domain-wall memory (DWM). We show that DWM is very well suited for RNN acceleration due to its very high density and low read/write energy. At the same time, the sequential nature of input/weight processing of RNNs mitigates one of the downsides of DWM, which is the linear (rather than constant) data access time.

RNNFast is very efficient and highly scalable, with flexible mapping of logical neurons to RNN hardware blocks. The basic hardware primitive, the RNN processing element (PE), includes custom DWM-based multiplication, sigmoid and tanh units for high density and low energy. The accelerator is designed to minimize data movement by closely interleaving DWM storage and computation. We compare our design with a state-of-the-art GPGPU and find 21.8× higher performance with 70× lower energy.

CCS Concepts: • **Hardware → Memory and dense storage**; • **Computer systems organization → Neural networks**;

Additional Key Words and Phrases: Recurrent neural networks, domain-wall memory, LSTM, accelerator

ACM Reference format:
Mohammad Hossein Samavatian, Anys Bacha, Li Zhou, and Radu Teodorescu. 2020. RNNFast: An Accelerator for Recurrent Neural Networks Using Domain-Wall Memory. *J. Emerg. Technol. Comput. Syst.* 16, 4, Article 38 (September 2020), 27 pages.
https://doi.org/10.1145/3399670

1 INTRODUCTION

Deep learning is transforming the way we approach everyday computing. From speech recognition that empowers today's digital assistants to business intelligence applications fueled by the analysis of social media postings, processing information in a way that preserves the correct context is

This work was funded in part by the National Science Foundation under XPS Award 60053525.
Authors' addresses: M. H. Samavatian, L. Zhou, and R. Teodorescu, The Ohio State University, 2015 Neil Ave., Columbus, OH 43210; emails: {samavatian.1, zhou.785, teodorescu.1}@osu.edu; A. Bacha, University of Michigan, 4901 Evergreen Rd, Dearborn, MI 48128; email: bacha@umich.edu.

1550-4832/2020/09-ART38 $15.00
https://doi.org/10.1145/3399670

crucial. For instance, the sentences "white blood cells destroying an infection" and "an infection destroying white blood cells" have very different meanings even though they contain the same words. Traditional machine learning designs such as Convolutional Neural Networks (CNNs) do not consider context and are therefore not well suited for solving such problems. Recurrent Neural Networks (RNNs) are a powerful class of networks designed to consider context by retaining and using information from previously processed inputs. RNNs are used across a wide range of applications that include speech recognition for digital assistants such as Siri and Google Now, sentiment analysis for classifying social media postings, and language translation. The popularity of RNN networks in production applications was highlighted by Google in a recent paper [31], which reports that RNN workloads represent almost 30% of the workloads on Google's TPU datacenters. This is in contrast to only 5% for CNN workloads.

However, RNN workloads are data-intensive because they store a partial history of the output sequence and perform computations on that history along with the current input. As a result, RNNs require both vast amounts of storage and increased processing power. For example, the RNN neuron requires 8× the number of weights and multiply-accumulate (MAC) operations of a typical CNN cell. RNN networks are also generally quite large. For instance, Amodei et al. [5] developed a network for performing speech recognition that utilized seven recurrent layers and a total of 35 million parameters. At this scale, RNNs with large input sets are susceptible to memory bottlenecks when running on existing accelerators such as GPUs [23] or FPGAs [9, 18, 23, 35, 36, 41, 63, 64]. In addition, the fundamentally different design of the RNN cell makes previously proposed custom CNN accelerators [4, 10–14, 17, 25, 30, 32, 33, 37–39, 46, 50–52, 59, 65] not directly applicable to RNN workloads.

This article presents RNNFast, a hardware accelerator for RNN networks. RNNFast leverages domain-wall memory (DWM), an emerging non-volatile memory technology, to provide high-density on-chip storage as well as energy efficient computation. DWM [16, 28, 47, 66, 67, 70, 75] is a magnetic spin-based memory technology, which stores information by setting the spin orientation of so-called magnetic domains in a ferromagnetic wire. Multiple magnetic domains can occupy a single wire (referred to as "racetrack") allowing up to 64 bits to be represented.

DWM has many attractive characteristics. It has read/write latencies that are close to SRAM and write performance and energy that are substantially lower than STT-RAM and other non-volatile memories [60]. Perhaps more importantly, DWM is expected to have 30× higher density than SRAM and 10× higher than DRAM or STT-RAM. The technology would therefore allow dramatically higher storage capacity in the same chip area. While the technology is still in the early stages of development, prototypes have yielded encouraging results [8]. We show that DWM is very well suited for RNN acceleration due to its very high-density, linear access pattern, and low read/write energy.

The RNNFast architecture is modular and highly scalable forgoing the need for long communication buses despite the high output fanout of typical RNN networks. RNNFast allows flexible mapping of logic neurons to RNN hardware blocks. The accelerator is designed to minimize data movement by closely interleaving DWM storage and computation. The basic hardware primitive, the RNN processing element (PE), includes custom DWM-based multiplication and custom nonlinear functional units for high performance and low energy. RNNFast also includes an error mitigation mechanism for position errors, expected to be relatively common in DWM. The error mitigation is tailored to the RNNFast data access pattern to minimize overhead. We compare RNNFast with a state-of-the art NVIDIA P100 GPGPU and find RNNFast improves performance by 21.8× while reducing energy 70×.

We also compare with two alternative RNNFast designs. (1) a CMOS-based RNNFast design in which both memories and logic use traditional CMOS. We find the RNNFast design to be up to two

times more energy efficient than the CMOS version, in a much smaller chip area. (2) A memristor-based implementation that uses an analog dot-product engine, a state-of-the-art design that has been shown to be very efficient for CNNs [7, 14]. RNNFast shows better performance, energy, and area than the memristor-based design. Qualitative comparisons with FPGA-based RNN accelerators, Google's TPU and Microsoft's Brainwave [19], also indicate RNNFast has better performance and lower energy for similar workloads.

This article makes the following main contributions:

- —Presents RNNFast, the first DWM-based custom accelerator for LSTMs and other RNN variants.
- —Introduces novel DWM-based designs for efficient neural network hardware including sigmoid, and tanh units.
- —Implements an efficient error mitigation solution for DWM overshift errors.
- —Presents a new efficient and scalable interconnection mechanism based on racetrack chains.
- —Demonstrates that DWM is very well suited for efficient acceleration of recurrent neural networks.

The rest of this article is organized as follows: Section 2 provides background information. Section 3 details the RNNFast architecture. Section 4 presents the error mitigation aspects of the design. Sections 5 and 6 describe the evaluation. Section 7 discusses related work and Section 8 concludes.

2 BACKGROUND

RNNs are a powerful class of networks that have the ability to learn sequences. They are applicable to anything with a sense of order that needs to be remembered. RNNs are used across a wide range of applications that includes speech recognition for enabling today's digital assistants, sentiment analysis for analyzing posts (text and video) and classifying them as positive or negative, and machine translation for sequence to sequence translation between languages.

2.1 The Long Short-Term Memory Cell

Most RNNs make use of special "neurons" called Long Short-Term Memory (LSTM) cells [22, 27]. LSTMs are designed to process and remember prior inputs and factor them into their outputs over time. Figure 1 shows an example of a very simple three-layer RNN with three LSTM cells/layer. The output of each layer is a vector that is supplied as the input to the following layer. In addition to those inputs, a feedback loop takes the output vector of each layer and feeds it back as an additional input to each LSTM neuron. An illustration of the inputs and outputs of a single LSTM cell C unrolled over time is shown in Figure 1(c). An input x_0 into neuron C at timestep $t = 0$, will generate an output h_0 that is propagated downstream to the next layer. In addition, h_0 is saved within the neuron's memory cell for use in the next timestep. At timestep $t = 1$, the same neuron C will process input x_1, but also use the previously stored output h_0 to generate the new output h_1.

A detailed look inside the LSTM neuron (Figure 1(b)) reveals a significantly more complex operation compared to CNN neurons. The strength of the LSTM lies in the way it regulates the fraction of information it recalls from its embedded memory and the fraction of input it processes for generating outputs over time. In other words, the LSTM cell progressively memorizes and forgets contextual information as it processes more inputs. This is achieved through special gates that are controlled through a set of mathematical functions [21] governed by Equations (1)–(5).

$$i_t = \sigma(W_{xi}x_t + W_{hi}h_{t-1} + b_i), \tag{1}$$

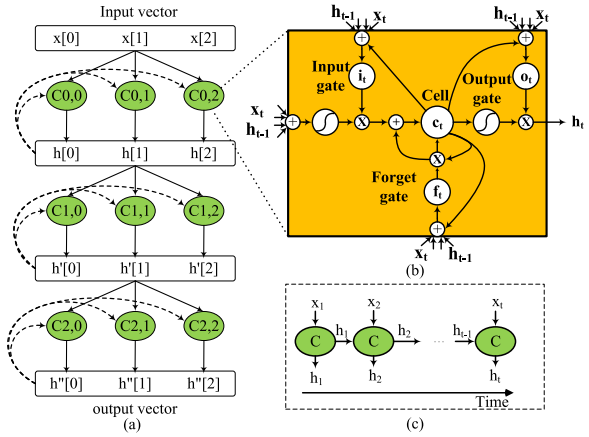

Fig. 1. (a) Three-layer RNN with three LSTM cells/layer; (b) LSTM cell; (c) an LSTM cell unrolled over time.

$$f_t = \sigma(W_{xf}x_t + W_{hf}h_{t-1} + b_f), \tag{2}$$

$$o_t = \sigma(W_{xo}x_t + W_{ho}h_{t-1} + b_o), \tag{3}$$

$$c_t = f_t \odot c_{t-1} + i_t \odot tanh(W_{xc}x_t + W_{hc}h_{t-1} + b_c), \tag{4}$$

$$h_t = o_t \odot tanh(c_t). \tag{5}$$

The input gate i_t receives the input to be written into a neuron's memory cell at timestep t. The forget gate f_t controls what information should be erased from a neuron's memory cell at timestep t. The cell c_t represents the content of the neuron's memory cell. The output gate o_t controls the amount of information read from the neuron's cell and how much of it contributes to the output. The output h_t represents the output of the cell to the next layer at timestep t. This output is also fed back into the input gate i_{t+1} of the same LSTM cell at timestep $t + 1$. The W's and b's represent the weights and biases, respectively.

Note that $\odot$ used in Equations (4) and (5) represents the dot product operator. In addition, Equations (1)–(5) represent neurons for an entire layer within a network. Therefore, i_t, f_t , o_t, c_t, h_t, h_{t-1}, and x_t are vectors and all W's are matrices. As such, if we augment a given matrix W to include the weights for both x and h such that its dimensions are $n \times m$, then each row in W^l for hidden layer l would be mapped to neuron j where $j \in [1, n]$. The value m is the size of input vector.

$$W^l = \begin{bmatrix} W^l_{11} & \cdots & W^l_{1m} \\ \vdots & \ddots & \vdots \\ W^l_{n1} & \cdots & W^l_{nm} \end{bmatrix}. \tag{6}$$

The $tanh$ and σ activation functions are also outlined in Equations (7) and (8) for clarity. These functions are applied as elementwise operations on the resulting vectors.

$$\sigma(z) = \frac{1}{1 + e^{-z}}, \tag{7}$$

$$tanh(z) = 2\sigma(2z) - 1. \tag{8}$$

Because of the complex design, LSTM cells require substantially more storage and computation relative to their CNN counterparts. Moreover, RNN networks are also generally fully connected, further increasing the data movement overhead.

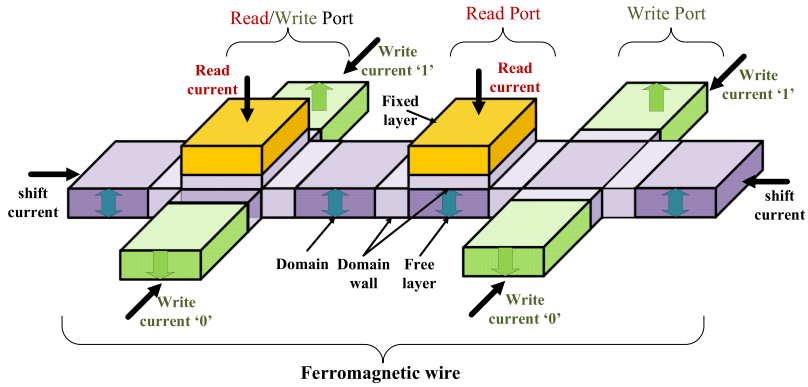

Fig. 2. DWM device structure.

2.2 Domain-Wall Memory

Domain-wall (a.k.a. racetrack) memory was first proposed by Parkin et al. [47] from IBM in 2008. In 2011, Annunziata et al. [8] demonstrated the first 200-mm DWM wafer, fabricated with IBM 90-nm CMOS technology. Each die contained 256 racetrack cells, proving the feasibility of DWM fabrication. A large body of research has since sought to improve and optimize the technology at device and circuit levels [20, 44, 54, 55, 61, 71, 74] and find solutions to improve its reliability [72].

Domain-wall (racetrack) memory represents information using the spin orientation of magnetic domains in a ferromagnetic wire, as shown in Figure 2. Each of these domains can be independently set to an up-spin or down-spin to represent the value of a single bit. Since multiple magnetic domains can reside on a single wire, multiple bits (32–64) of data can be packed in a single DWM device, resulting in a very high density. Three basic operations can be performed on a DWM device: read, write, and shift. A magnetic tunnel junction (MTJ) [53, 69] structure is used to read data from the DWM cell (read port in Figure 2).

In a DWM device, all the magnetic domains share a single read MTJ (generally referred to as a read head or port). The bit to be read needs to be aligned with the MTJ before it can be accessed. This is accomplished using a property that is unique to DWM, called domain-wall motion, which refers to the shifting of magnetic domains down the ferromagnetic wire. When a current pulse of a suitable magnitude is applied through the ferromagnetic wire, the magnetic spins of all domains "move" across the wire in a direction opposite to the direction of the current. The number of bit positions in a shift motion is controlled by the duration of the shift current. Additional blank domains are included at the ends of each racetrack to allow all data domains to be shifted to the read head without data loss at the ends of the wire [49].

Writing into DWM is also fast and energy efficient due to recently developed [71] "shift-based writes" as demonstrated in Figure 2 (write port). The design of the write head consists of a ferromagnetic wire with two fixed domains that straddle a free domain at an arbitrary location on the racetrack. One of the fixed domains is hardwired to up-spin and the other to down-spin at fabrication. The spin of either of the fixed domains can be shifted into the free domain through the domain motion process by applying a current pulse in the appropriate direction. The latency and energy of shift-based writes are equivalent to those of simple shifts.

The main challenge of racetrack memory is the access latency to data stored in a DWM tape, which is variable depending upon the number of shifts required to align the accessed bit with the read or write heads. RNNFast mitigates this disadvantage by optimizing data placement for sequential access such that most accesses only require a single shift.

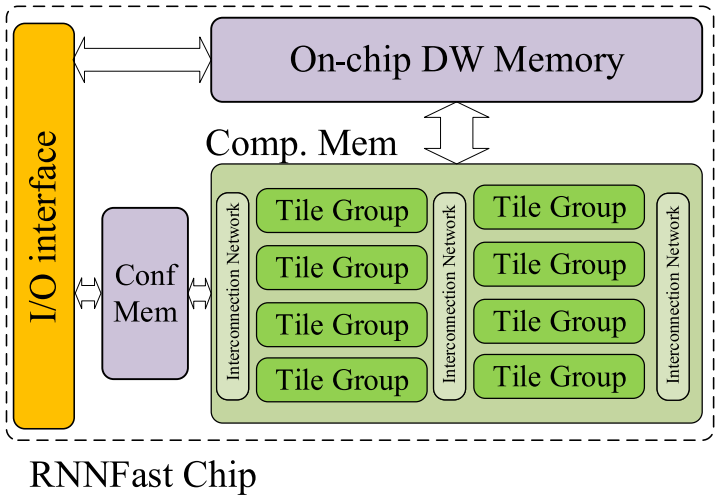

Fig. 3. RNNFast architecture overview at chip level.

2.2.1 Reliability. DWM technology also presents reliability challenges including possible misalignment of the data domains leading to erroneous reads and/or writes [29, 72]. Prior work [72] has classified DWM errors into two main types: "stop-in-the-middle" and "out-of-step" errors. The first class of errors is caused when data domains are not aligned with the read/write heads, leading to invalid accesses. The second class of errors is caused when the incorrect domain is aligned with the read/write head, which causes the wrong bit in the track to be accessed. The errors are generally caused by variability in the magnitude or duration of the current pulse applied during the domain shift operation. Zhang et al. [72] has developed a technique for eliminating "stop-in-the-middle" errors that relies on the application of a short subthreshold shift current to nudge the misaligned domain back into alignment. They also demonstrate that the subthreshold pulse is small enough that it cannot misalign a correctly aligned domain. As a result, subthreshold shifts can virtually eliminate "stop-in-the-middle" errors, at the cost of increasing the number of "out-of-step" errors.

While subthreshold shifts can be applied in both directions, we choose to apply them in the shift direction. As a result, all "out-of-step" errors will be converted into overshift errors by one or more positions in the shift direction. For a single-position shift, which represents virtually all shifts in RNNFast, the probability of single-bit overshift is on the order of 10^{-5} [72], which is quite high. However, the probability of multibit overshift is about 10^{-21}, which is negligible. As a result, RNNFast implements mitigation for single-bit overshift errors.

3 RNNFAST ARCHITECTURE

At a high level the RNNFast chip consists of Global Memory, a Computational Memory array, Configuration Memory, and I/O interface as shown in Figure 3. The Global Memory is a dense memory block implemented using DWM. This is the main memory of the accelerator and is used to store inputs and results. The Computational Memory is the compute engine and is implemented primarily using DWM elements augmented with CMOS logic where appropriate. The compute array is organized as a pool of highly reconfigurable and tightly interconnected tile groups.

One or more multi-layer RNN networks can be mapped to multiple tile groups, in a weight-stationary design (weights are stored locally in the Computational Memory). The Configuration Memory holds the runtime configuration settings for the chip.

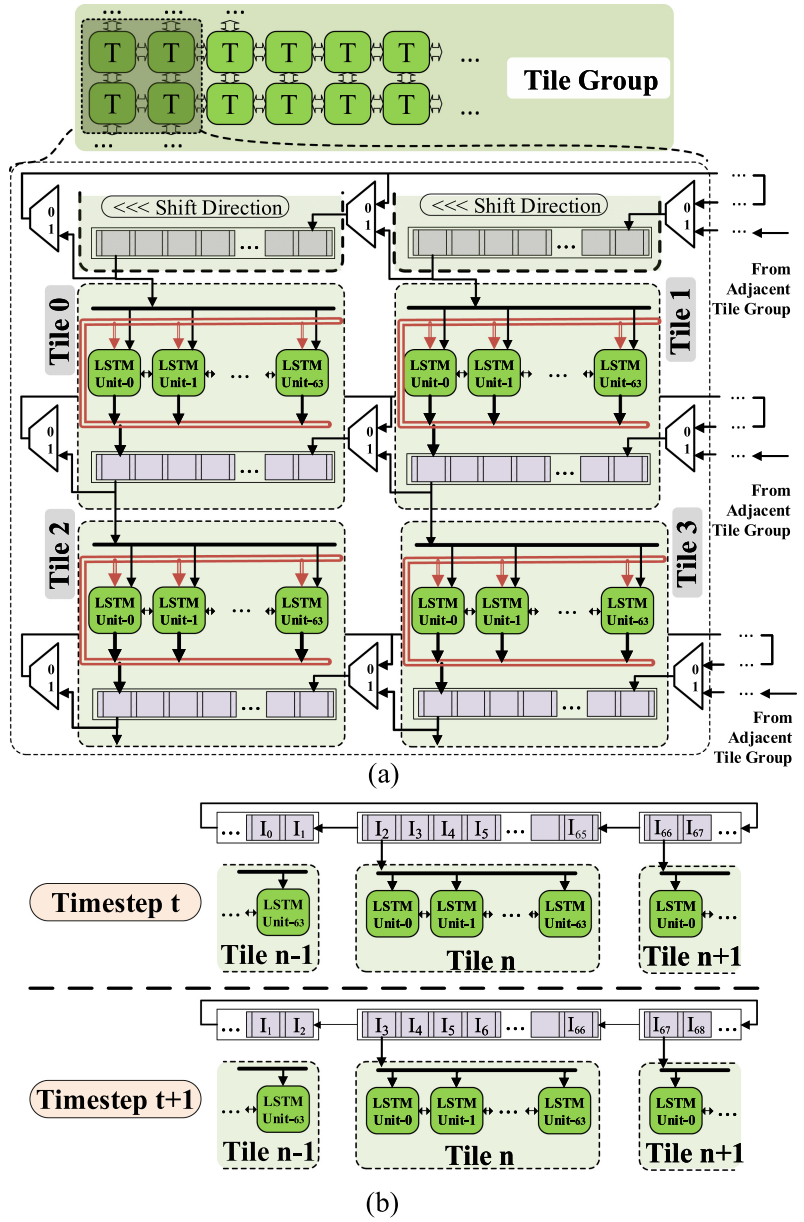

Fig. 4. (a) Compute tile layout, internal design, and interconnection through racetrack chains. (b) Reading inputs into tiles in two consecutive timesteps.

3.1 Compute Tiles

A compute tile consists of multiple LSTM hardware units that share a single input and a single output racetrack. They are interconnected with their nearest horizontal and vertical neighbors through racetrack memories. Figure 4 shows the tile design and layout. The results of the computation within each tile are written directly onto the input track of the tile belonging to the next

layer in the network. Tiles are organized in tile groups, which are connected to each other through traditional wired interconnection networks.

3.1.1 Inter-Tile Communication. RNNs are typically fully connected networks requiring all inputs to be delivered to all the neurons in a given layer. The high degree of connectivity that has to be supported by the hardware can lead to substantial energy and area overheads when traditional wired interconnects are used. To address this challenge, we leverage the shifting mechanism of DWM racetracks for communication both within and across tiles.

Within a tile, inputs are read sequentially from the tile's input racetrack and broadcast to all LSTM units across a locally shared bus. Each read is followed by a shift of the input track to align the next input element with the read head. Figure 4(b) illustrates two timesteps in this process. In addition to the tile-local broadcast, each input is also sent to the neighboring tile on the left for addition to its input track. We call this process "chaining." Chains are essentially circular buffers that circulate all inputs to all tiles that are mapped to the same layer of the NN. Chains of different lengths can be configured depending on the number of neurons in each layer of the network. Racetracks are connected through MUXs (Figure 4(a)) that enable different chain lengths. A variable number of tracks can be included in a chain by simply setting the rightmost track MUX to 0 and the rest to 1.

3.2 LSTM Units

Each tile consists of multiple LSTM compute units (64 in our design). RNNFast is a weight-stationary design, with fixed capacity for weight storage in each LSTM unit. A logical neuron can be mapped to one or more LSTM compute units depending on the number of weights it requires. We expect a one-to-one mapping between logical neurons and hardware LSTM units for most networks. However, when a logical neuron requires more weights than a single LSTM unit can store, it is mapped to multiple LSTM units. Figure 5(a) shows three mapping examples for a single logical LSTM cell: one LSTM unit (top), two LSTM units (middle), and four LSTM units (bottom).

3.2.1 PEs. The architecture of an LSTM cell is shown in Figure 5(b). Each cell is subdivided into four PEs ①. Per Equations (1)–(5), each input X_t is multiplied with four different sets of weights. A single PE can be assigned to any one of the weight sets (known as gates), e.g., I_G, F_G, O_G, or C_G. However, an LSTM cell gate can be mapped to one or more PEs across LSTM units depending on its storage requirements and input/output fanout. Allocating four hardware PEs to each LSTM unit allows RNNFast to accommodate different RNN variants (see Section 3.4).

PEs have racetrack-based storage for weights and racetrack-based compute units, including MAC engines for matrix multiplication. The MAC engine is composed of 256+16 DWM-based full adders. The MAC unit is deeply pipelined into 48 stages. In order to increase parallelism, each PE uses two MAC engines; one for the main input X_t and one for the feedback input h_{t-1}.

Each PE unit holds a set of weights and performs the dot product on the corresponding subset of inputs. Each PE only consumes inputs corresponding to the weights it stores. Each input to a PE is multiplied by its weight and accumulated with the result of the previous multiplication ②. Each PE stores the result of the accumulation in its own output racetrack.

3.2.2 Input and Weight Mapping. The input and weight assignment to racetracks is a tradeoff between access latency and hardware overhead. In RNNFast, inputs are spread across multiple racetracks with 1 bit per track. This allows an entire input word to be read in a single cycle, as the top half of Figure 6 illustrates. Error detection bits are also included in the tracks and their role will

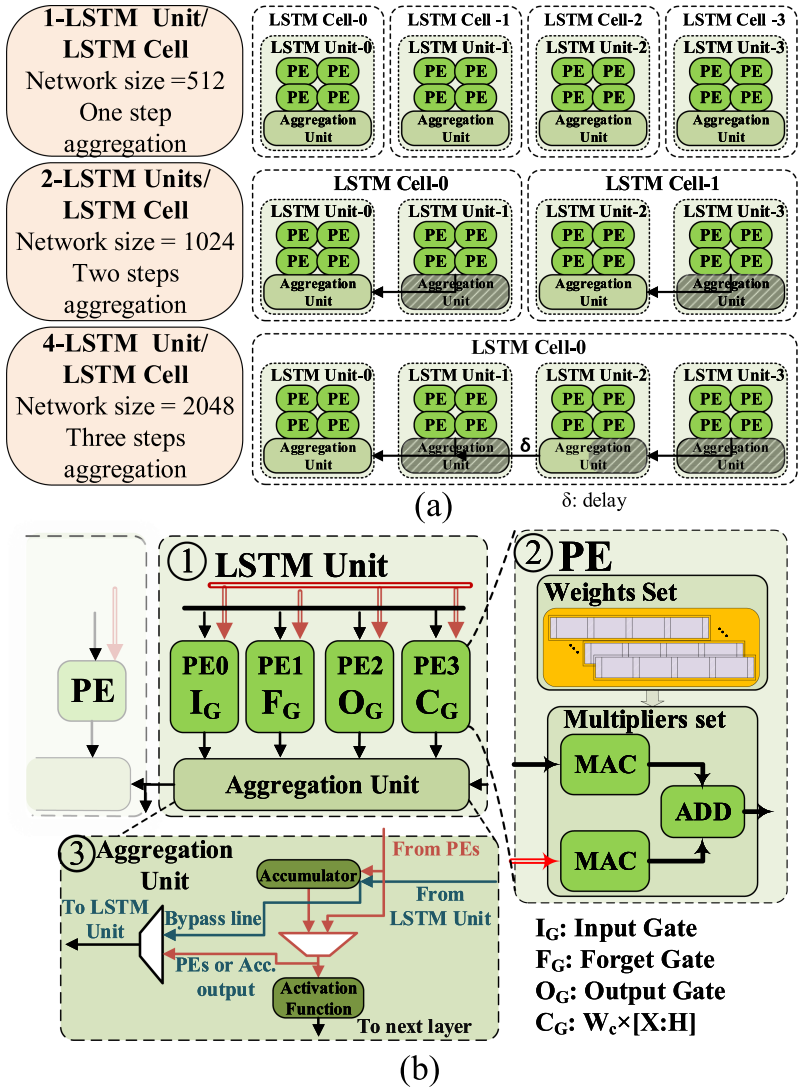

Fig. 5. (a) Three mapping examples of logical LSTM cells to LSTM units. (b) LSTM unit design.

be detailed in Section 4. Note that the input tracks do not require dummy domains (Figure 4(b)). Values at the end of the track are read and sent to the neighboring track.

Unlike inputs, which move from track to track along the chain, weights are stationary at the PE level and are reused multiple times. This means that after scanning all weights, the tracks need to be returned to the initial weight. To minimize the number of shifts, weight values are distributed both within and across multiple racetracks. Weight racetracks are provisioned with multiple read/write heads (five in our design) which divide the racetrack into six 10-bit segments. The leftmost segment domains are used as dummy domains and the rest of the segments are used to store weight values. Data layout is such that all read heads across all tracks can access all the bits of a single weight simultaneously. Racetracks are grouped in sets of four, with each set storing 10 weights. The bottom of Figure 6 illustrates this layout. Weight W_0 (red) is currently aligned

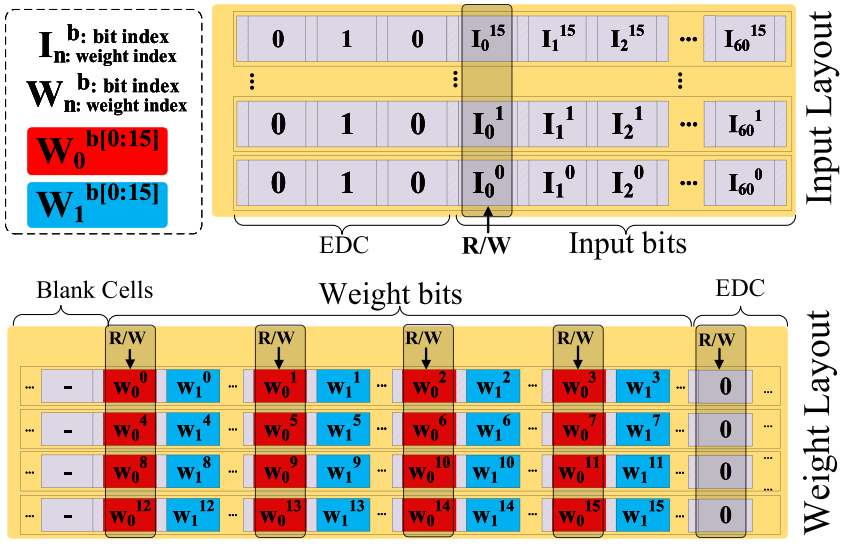

Fig. 6. Mapping of inputs and weights to racetracks.

with the read heads. A single-position shift to the left will align the next weight W_1 (blue) with all the read heads. Access to each set of weight racetracks is pipelined. When all 10 weights are read from the current set of racetracks, the next set of weights will be read from the next set. While the new weights are accessed, the weights in the previous set are shifted back to their initial positions. This takes place when the racetrack set is not being accessed and is therefore off the timing critical path.

3.2.3 Result Aggregation. If more than one LSTM unit is mapped to a neuron, the partial results of the individual LSTMs have to be combined to form the neuron's output. Aggregation units ③ in each LSTM are used to sum up partial results in that LSTM block. In addition, the aggregation units apply the sigmoid and tanh functions and perform the multiplication and accumulation operations in order to generate the final output of the cell.

For cases in which neurons span multiple LSTM blocks, aggregation units in those blocks are linked to produce the final result. This is achieved by collecting all the partial results computed by each LSTM unit (mapped to the same neuron) to a single aggregation unit. Aggregation units are also chained through adjacent LSTM units. Each aggregation unit sends out its final result to the adjacent aggregation unit to its left. The adjacent unit will use the incoming result to either accumulate or bypass it to the next unit (Figure 5- ③). Even-indexed aggregation units consume and odd-indexed aggregation units forward the incoming result. The leftmost LSTM in a neuron will be responsible for the final aggregation and will apply the sigmoid and tanh. Aggregation time is a logarithmic function in the number of LSTM cells mapped to a single neuron. This is also done by setting multiplexers in the aggregation unit and power gating the inactive units in output generators at odd indexed LSTM units.

The design tradeoff for LSTM units is driven by the need to support networks that are both large and small. If LSTM units and PEs are too large, storage space will be wasted when small networks are mapped. If they are too small, large networks will require several LSTM units per neuron, increasing the aggregation time.

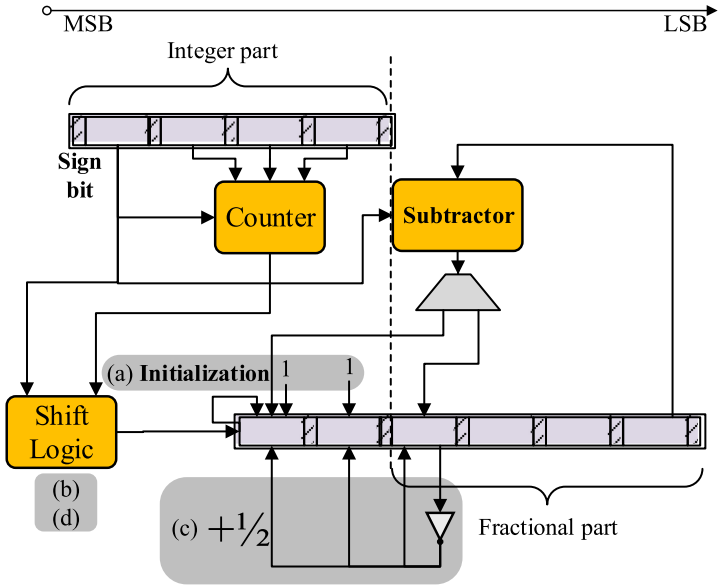

Fig. 7. DW-based implementation of sigmoid/tanh.

3.3 Nonlinear Functions

The nonlinear functions are an important component of the RNN cells and are used for output activation. RNNFast uses hardware acceleration for the sigmoid and tanh nonlinear functions. The hardware is included in each Aggregation Unit (Figure 5). We propose an area efficient approximate logic function-based unit implemented using DWM for the nonlinear functions.

The approximation has been proposed by prior work [57] as an alternative to the standard sigmoid and follows Equation (9):

$$\sigma(z) = \begin{cases} \frac{\frac{1}{2} + \frac{\hat{z}}{4}}{2^{|(z)|}} & if z < 0 \\ 1 - \sigma(-z) & if z > 0 \end{cases}. \tag{9}$$

This approximation has the advantage of being easier to implement in hardware. As Equation (9) shows, the hardware has to support division by 2^n numbers. This can be implemented using shift operations which are a feature of racetrack memories. The tanh approximation function can be computed from the sigmoid function through two multiplications and a subtraction. Note that $\hat{z} = z + \mid (z) \mid$, where (z) is the integer part of z.

Figure 7 shows our DWM-based implementation of the sigmoid approximation. The sigmoid for a negative value will be computed as follows: (a) the output integer part is initialized with binary "1"; (b) two right shifts are performed to compute $\hat{z}/4$; (c) $+1/2$ is applied to the result; (d) the final result is shifted right $|(z)|$ times. For a positive number two subtraction steps are added in the beginning and end of the above steps. To compute the tanh approximation, a right shift ($2 \times z$) and a subtraction will be applied in the first and last steps, respectively. This design is very area and energy efficient utilizing only a 16-bit racetrack memory, along with some simple subtraction and counting logic. Section 6 evaluates the relative merits of the approximate designs regarding LUTs.

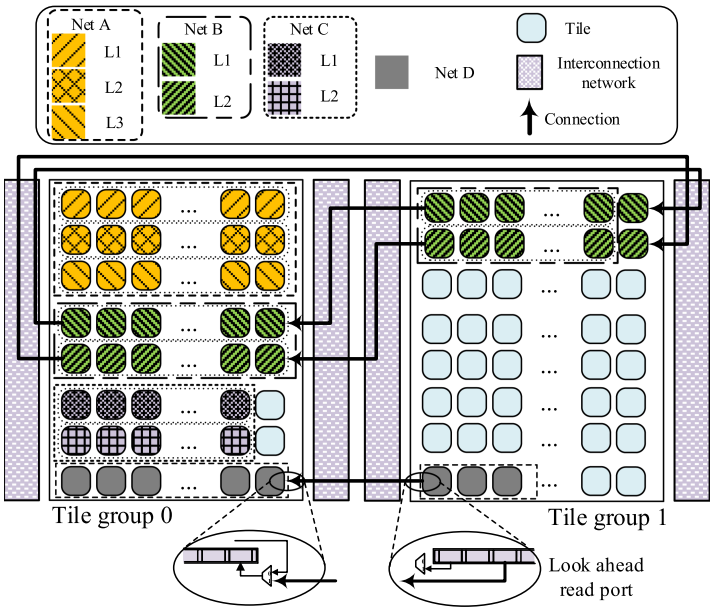

Fig. 8. Mapping multiple LSTM networks to RNNFast. Interconnection network helps extend racetrack chains beyond tile groups for large networks.

3.4 RNNFast Mapping and Configuration

The RNNFast hardware can be configured to implement different network sizes and topologies. Moreover, multiple distinct neural networks can be mapped to the same chip.

Outputs from one network can be delivered directly to the following network or stored in the on-chip memory for further processing, if needed. Figure 8 illustrates an example of four networks A, B, C, and D mapped to two tile groups. Tile groups are connected through a wired interconnect. The racetrack chains for each row of tiles have additional read/write heads to provide access to the inter-tile network.

Multilayer networks span multiple rows with different layers mapped to consecutive rows. Tile groups are designed with wide rows to accommodate most network sizes (e.g., Nets A and C). However, when a network layer cannot fit in a single row, RNNFast supports splitting it across tile groups (e.g., Nets B and D). This is achieved by extending the input/output racetrack chains to neighboring tile groups using the inter-group wire interconnect. We chose to split layers across tile groups (as opposed to within a tile group) in order to allow consecutive network layers to continue to be mapped to adjacent rows, preserving inter-layer communication.

One important design constraint was to enable the extension of the racetrack chains across tile groups without adding to the track chain shift latency. This is accomplished by implementing a look-ahead read port at the end of the track that reads inputs several cycles ahead of the end of the track, as illustrated for Net D in Figure 8. This allows the input to reach the destination row in the neighboring tile through the higher latency interconnect by the time the same input reaches the end of the source track.

3.4.1 Other LSTM Variants. RNNFast is designed for the more demanding LSTM design. However, it is also compatible with LSTM variants like Gated Recurrent Unit (GRU) and Vanilla RNN, which require fewer compute resources. Unlike LSTM, the GRU unit does not use a memory

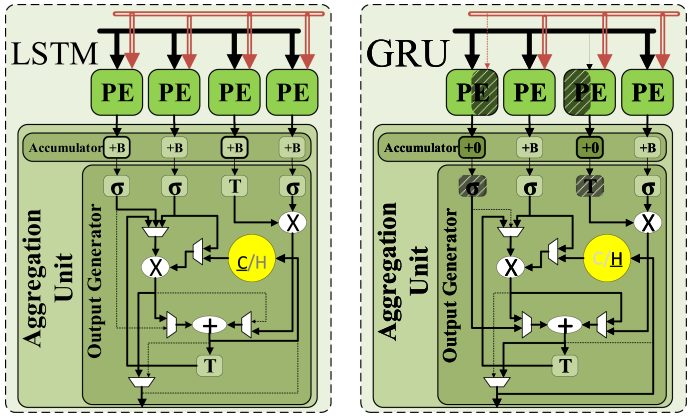

Fig. 9. LSTM vs. GRU cell configuration on RNNFast.

element to control the flow of information and is useful when input sequences are not very long. Figure 9 shows how a GRU cell can be mapped to a RNNFast LSTM unit. The shaded areas represent unutilized components. A GRU utilizes 75% of the MAC resources.

Simpler RNNs like *Vanilla RNN*, only utilize a single PE per neuron and do not need the aggregation unit. As a result, RNNFast can map four *Vanilla RNN* neurons in each LSTM unit.

Moreover, RNNFast allows the mapping of other network types such as Bidirectional RNNs (BiRNNs). A BiRNN consists essentially of two RNNs stacked on top of each other. The output is computed based on the hidden state of both networks. In our design, the two networks are mapped on the hardware in an interleaved fashion. The aggregation hardware is used to link the two networks. The input data is also duplicated and interleaved in reverse order $(x_1, x_n, x_2, x_{n-1}, x_3, x_{n-2}, \ldots, x_n, x_1)$.

3.4.2 RNNFast Configuration. The RNNFast configuration is programmed through configuration registers that control input assignment at the PE level, input track chaining, result aggregation setup, and so forth. A configuration file with the LSTM network(s) specifications is loaded into the device driver of the accelerator and propagated to the appropriate registers.

4 ERROR MITIGATION DESIGN

4.1 DWM Position Errors

Out-of-step shift errors, in which the wrong bit is aligned with the read/write heads, are a significant reliability challenge for DWM. Since RNNFast accesses data sequentially, that means virtually all accesses require only single-position bit shifts. We therefore focus only on single-bit overshift errors, which are expected to occur with a relatively high probability (10^{-5} according to [72]).

While prior work [50] has shown that neural networks are quite resilient to errors, we find that error rates on the order of DWM overshift errors can degrade output accuracy substantially. Figure 10 shows the accuracy of the output for two benchmarks, measured by the BLEU (bilingual evaluation understudy) metric [45], relative to an error-free baseline. BLEU is an algorithm for evaluating the quality of text which has been machine-translated from one natural language to another. Quality is considered to be the correspondence between a machine's output and that of a human. The models that we used have reported very close BLEU scores to the state-of-the-art models [58]. We inject single-bit overshift errors in different DWM components of RNNFast: the racetrack chains used to hold inputs and outputs for each NN layer, the weights associated with all

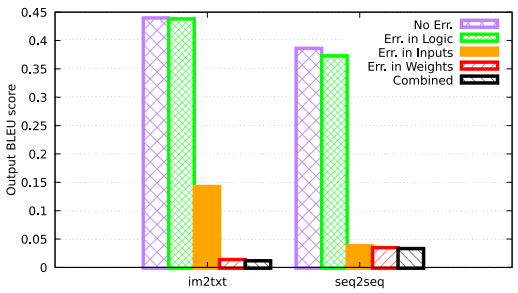

Fig. 10. Output accuracy (BLEU score) for logic, inputs, and weights components.

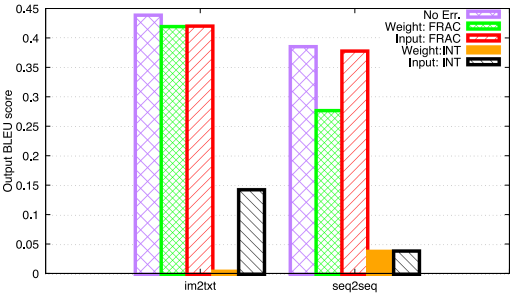

Fig. 11. Output accuracy (BLEU score) for integer and fraction components.

PEs, and the DWM components of the logic functions (MAC units and the nonlinear functions). Shift errors are modeled as a uniform distribution with an overshift probability of 4.55×10^{-5} [72].

Figure 10 shows that when errors are injected only in the logic, the drop in output accuracy is very low: <1% for *im2txt* and 3% for *seq2seq*, two of the benchmarks we run. This is because overshift off-by-one errors in the MAC and nonlinear functions tend to produce results that are relatively close to the correct value. As a result, the accuracy of the output is very high. However, when errors are injected into the input chains and the weight arrays, the output accuracy drops dramatically to between 10% and 35% of the original. When errors are injected uniformly in all DWM tracks, the output accuracy drops below 5% for *im2txt* and below 10% for *seq2seq*, meaning that the results are essentially useless. This data highlights that mitigation solutions for errors in the inputs as well as weights are essential.

To better understand which errors have the worst effect on output quality, we selectively inject errors into different bits of data words. RNNFast uses 2's complement fixed point representation for both inputs and weights. We inject errors separately into the integer and the fraction portions of the word. Figure 11 shows the results of this experiment. When errors are injected only in the fraction, the drop in accuracy is less than 3% for both inputs and weights in *im2txt*. For *seq2seq*, the accuracy degradation is worse when errors are injected in the weights compared to inputs, but the overall output quality is still reasonably high.

Injecting errors with the same probability in the integer portion of the data words has a much more dramatic effect, leading to a drop in output accuracy of between 35% and 10%. The large effect is due to the fact that in these workloads both inputs and weights are represented with small fractional numbers. A single bit flip in the integer fraction can turn a small number into a much larger value, which has a disproportionate effect on the rest of the network.

The large effect on output accuracy is due to the 2's complement representation. This is because a single shift error in a data word that stores a small value can cause that value to be interpreted as a large value with the opposite sign. For example, the binary "00000011.10000010" (3.5078125 in decimal) would flip into "00100011.10000010" (35.5078125) or "10000011.10000010" (−124.492188) when a non-sign or sign bit in the integer part is inverted, respectively. This is also true for a negative number. The value "11111111.00101010" (−0.8359375) turns into "01111111.00101010" (127.1640625) after the sign bit is flipped.

4.2 RNNFast Error Mitigation

RNNFast addresses overshift errors by implementing an efficient error mitigation mechanism that considers the sensitivity of RNN workloads to errors that result in very large values. We implement different error detection and mitigation mechanisms for input/output racetrack chains and

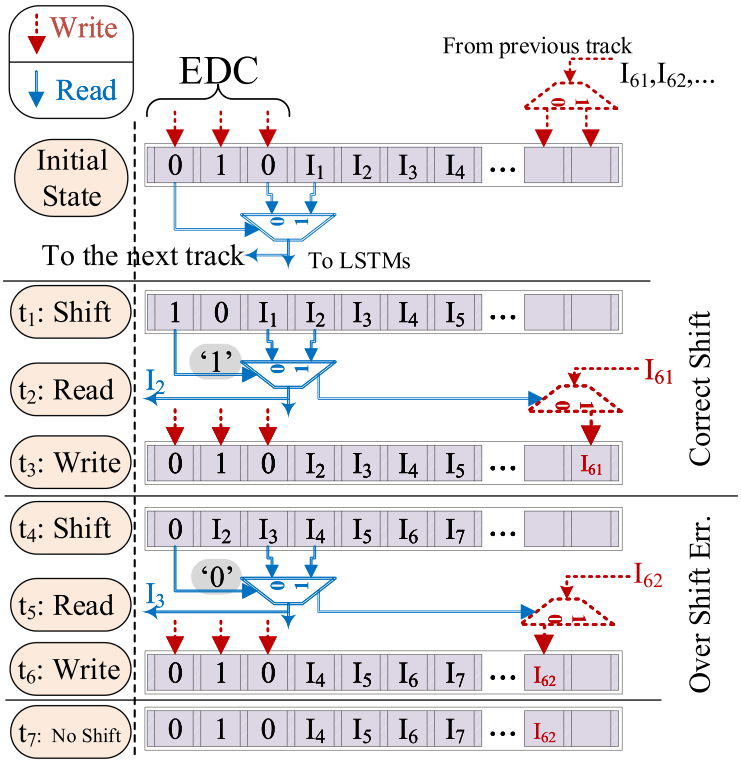

Fig. 12. Mitigation mechanism for overshift errors in the input track chains.

for weight arrays. We take advantage of their design characteristics to implement a more efficient single error detect, single error correct (SEDSEC) design that has lower area overhead and requires fewer extra domains and access ports compared to prior DWM Error Detection Code (EDC) solutions such as [72].

4.2.1 Input Errors. In order to detect overshift errors in the input tracks, we append a 3-bit pattern to the left side of each track, as shown in the example in Figure 12. The figure shows a single track that stores bit n for multiple inputs I_1–I_7. In the initial state, the EDC "101" is stored in the leftmost bits of the track. Input I_1 is read in the current cycle. At time t_1 the track is shifted left by one to access the next input. If the shift is correct, the leading (check) bit should be a "1." Input I_2 is read and sent to the LSTM units. A new EDC code is written at cycle t_3 in the first 3 bits of the track using three parallel write ports. Note that updating the EDC does not introduce any time overhead since a write cycle already exists following each read to allow data to be written into the next track in the chain.

At cycle t_4 we show an overshift error. The track has incorrectly shifted left two positions instead of one. This means that I_3 (instead of I_2) is now aligned with the read head. The check bit is now "0" indicating a shift error. To recover from this error, we use an additional read head to also read I_2. The outputs of the two read heads are connected to a multiplexer. The check bit value selects the multiplexer output (shown in blue in Figure 12). A "1" selects the error-free output and a "0" selects the overshifted output. A similar mechanism selects the correct location for writing the input coming from the previous track in the chain. If an overshift error occurs, the write location is also shifted to the left, as the right-hand side of Figure 12 shows.

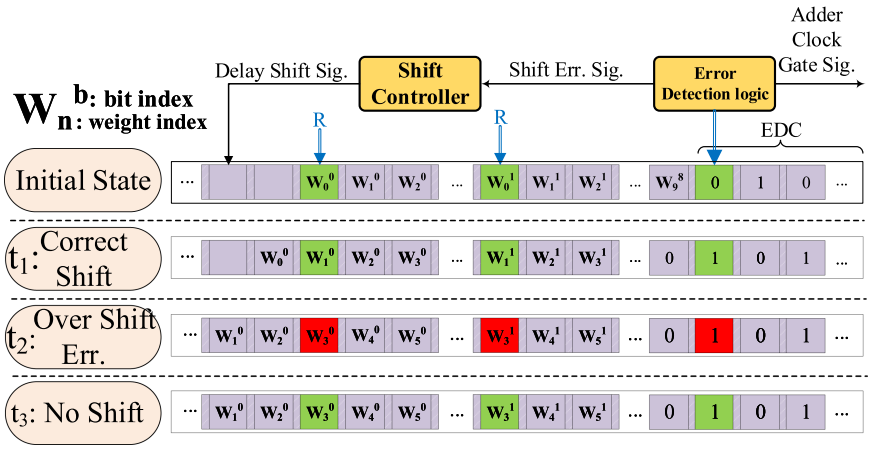

Fig. 13. Mitigation mechanism for overshift errors in the weight track chains.

At t_6 the EDC code is updated again. Following an overshift error, the shift controller will not issue a shift command for the following cycle (t_7) since the track is already properly aligned to access the next input (I_4) during that cycle. Note that, since individual words are stored across multiple tracks to enable single-cycle access, an overshift error will affect all inputs that share that track (up to 60 in our design). It is therefore important to detect and correct these errors.

4.2.2 Errors in Weight Arrays. A similar mechanism is deployed to detect and mitigate errors in weight arrays associated with each PE. However, because the access timing to the weights array is more critical and weights are stored in a more compact representation, the detection and mitigation steps are implemented differently. Unlike inputs, which move from track to track along the chain, weights are stationary at PE level and are reused multiple times. This means that after scanning all weights, the tracks need to be returned to the initial weight. To minimize the number of shifts, weight values are distributed both within and across multiple racetracks. Weight racetracks are provisioned with multiple read/write heads (five in our design). Data layout is such that all read heads across all tracks can access all the bits of a single weight simultaneously.

Similarly, unlike the input racetrack chain, access to the weight arrays does not require a write cycle, so an update to the EDC code is not feasible. We instead store a fixed EDC pattern of "01010" at the rightmost edge of the weight tracks as shown in Figure 13. Error detection logic detects an overshift error when the current EDC bit does not match the expected value. For instance, in the initial state, the read heads are aligned with bits from weight W_0 and the error detection logic expects to read "0" from the EDC.

At time t_1 a correct shift takes place and W_1 can be read. At time t_2 an overshift error occurs and weight W_3 is read instead of W_2. A recovery mechanism similar to the one for inputs could be employed. This would require doubling the number of read heads in each track and extra logic. Since weight storage in RNNFast is substantial, the overhead would be nontrivial. We can, however, avoid this extra overhead by leveraging the observation that replacing the incorrect weight with "zero" yields very little loss in output accuracy compared to error-free execution. This is in contrast with using the erroneous weight, which can be a large value. In the following cycle at t_3, the shift controller will not shift because the track is already aligned for accessing the next weight.

Table 1. Racetrack Memory and RNNFast Design Parameters with Associated Power and Area Overheads

DWM properties				
racetrack width/length/thickness		1F / 64F / 3 nm	domain length	1F
number of bits per track		64	Effective cell size	$2.56F^2$
read/shift/write latency		1 ns / 0.5 ns / 0.5 ns	Technology node	32 nm
read/shift/write energy		0.39 pJ / 0.24 pJ / 9.6 fJ		
Tile properties				
Component	Configuration	Specification	Power (mW)	area (μm^2)
Input buffer	1 track/tile with EDC	16 stripes/track 64 cell/stripe	2.59	2.68
LSTM unit	64 per tile	4 PEs/LSTM 1 Aggre./LSTM	9.74	2046
Total tile		**256 PEs 64 Aggre. Unit**	**626**	**0.130 mm^2**
PE properties				
MAC	2/PE	272 Adder	2.43	422
Weight array	2 track/PE with EDC	205 stripes/track 64 cell/stripe		
Aggregation Unit properties				
Accumulator	4/LSTM	-	0.004	356
Multiplier	2/LSTM	-		
sigmoid	3/LSTM	Approx. nonlinear func. design		
tanh	2/LSTM	Approx. nonlinear func. design		
On-chip DW Memory				
Size: 128 MB, 4 R/W ports, Area: 6.2 mm^2, Acc. Eng.: 0.89 nJ, Acc. lat.: 1.69 ns, Leakage 24.3 mW				

5 EVALUATION METHODOLOGY

5.1 RNNFast Modeling Infrastructure

We implemented a detailed behavioral model to evaluate performance, chip area, and energy consumption of the RNNFast design. A cycle-level model that accounts for the latency of each component in the design is used for the timing simulation. The simulated hardware is configured for each neural network in our benchmark set, by enabling the appropriate number of hardware tiles, LSTMs, and PEs. Since all LSTM units execute independently and in parallel, only a single LSTM per tile is simulated to speed up simulation time. For the energy evaluation, we include the number of reads, writes, and shifts as well as decoder, adder/multiplier, and LUT accesses for all the units in the design.

To understand the energy consumption, an electrical model for the shift and write latency of the DWM is necessary. To this end, a Verilog-A–based SPICE model for DWM from [42, 43, 44] was simulated on Cadence Virtuoso. The DWM model estimates the effective resistance as a function of the length of the track and uses the width and thickness of the strip to calculate current density and position shift. A Cadence component was created for the DWM model and a test-bench was set up to stimulate the device. A sensitivity analysis was conducted to study the effect of track length on the shift latency and energy.

Table 1 shows the characteristics of the DWM we model in addition to the architectural parameters for RNNFast and the power/area breakdown for the different components. Since the weight

values use 16-bit precision, each four sets of racetracks stores 10 weights. Therefore, storing 512 weights requires each PE to have 205 recetracks (Table 1). We performed an energy analysis on the number of LSTMs per tile and chose the number of LSTMs per tile to be 64. A more detailed discussion on parameter tuning is included in Section 6.4. The number of accumulator, multiplier, sigmoid, and tanh units in the Aggregation unit (Figures 1 and 9) is optimized for energy and performance. We select the smallest number of units that allows the LSTM to operate without stall cycles.

5.1.1 RNNFast Design Variations. We compare our design with two alternative RNNFast architectures that use CMOS and Memristor technologies. We call them RNNFast-CMOS and ISAAC-RNN, respectively. For RNNFast-CMOS, we used SRAM buffers for both LSTM inputs and weight storage within PEs. MAC units are also implemented with CMOS logic. We used SRAM-based LUTs for the nonlinear functions. Input SRAM buffers are also chained like racetrack memories in order to deliver all the inputs to the LSTM units.

ISAAC-RNN is an ISAAC [51]-like design for RNN that stores inputs in eDRAM and is entirely CMOS and memristor-based. ISAAC-RNN uses 128×128 2-bit memristor crossbars, similar to what was used in ISAAC, for the dot-product engine. We kept the input buffer and aggregation unit designs the same as RNNFast in order to observe the effect of memristor in the design and have a more fair comparison since eDRAM and CMOS logic have higher energy consumption than DWM. Each memristor dot-product engine is capable of 128×16 multiplications in parallel (128 inputs by 16 weights). Within an LSTM neuron, each input is multiplied by four different weight sets. Thus, each memristor dot-product engine can handle four neurons, making each crossbar in the ISAAC-RNN computationally equivalent to four LSTMs in RNNFast. Thus, there are 16 LSTM units per tile for ISAAC-RNN instead of 64 per tile in RNNFast. Inputs are delivered bit by bit to the memristor crossbars. However, a chuck of 128 inputs needs to be supplied in a single cycle. For a fair comparison, we changed the input layout to maximize the performance of ISAAC-RNN.

5.1.2 GPU Baseline. We choose as a baseline system for our evaluation a GPGPU optimized for machine learning: the NVIDIA Tesla P100 (Pascal architecture) with 16 GB of CoWoS-HBM2 memory. All of our benchmarks use the DNN-optimized cuDNN NVIDIA libraries version 7 [2], which deliver roughly 6× performance improvement relative to a standard GPU implementation for LSTM on Torch [3]. We measure the runtime of the forward passes through the LSTM layers using instrumentation in Deepbench. We measure power consumption using the NVIDIA SMI profiler. Since the SMI profiler provides the total board power, we subtract the power measured at idle in order to isolate the active power of the GPU. Since the board components are less energy proportional with activity compared to the GPU, they account for most of the idle power.

5.1.3 PUMA. We also compared our design with PUMA [6], a recently proposed DNN accelerator built with ReRAM. The authors of PUMA released a simulator and toolchain that we use to compile and run our benchmarks. We used the PUMA compiler to find the number of tiles required for each benchmark. We then set the simulator configuration file to inference mode and used the PUMA simulator to measure runtime and energy consumption.

5.2 Benchmarks

We used LSTM-based RNN workloads from the Deepbench [1] open source benchmark suite for DNNs, released by Baidu. For our experiments we used

Image Caption Generator: This benchmark is based on the "Show and Tell" Model [62], which is an encoder-decoder type neural network. The decoder is an LSTM RNN that generates captions from a fixed-length vector input.

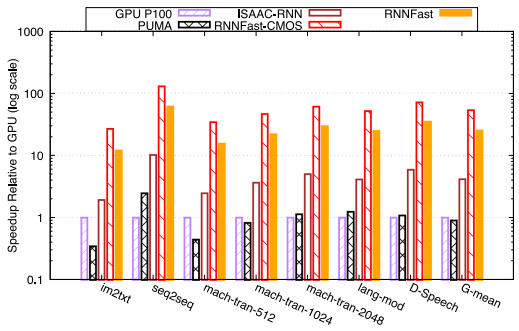

Fig. 14. RNNFast, RNNFast-CMOS, ISAAC-RNN, and PUMA runtime relative to the GPU P100 execution.

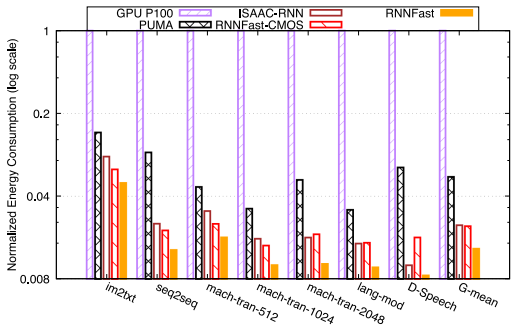

Fig. 15. Energy consumption for RNNFast, RNNFast-CMOS, ISAAC-RNN, and PUMA relative to the GPU P100.

Sequence-to-Sequence Model: This benchmark is based on the RNN encoder-decoder model by Cho et al. [15], which performs language translation. The encoder and decoder are three-layer LSTM networks.

Machine Translation: also based on the RNN encoder-decoder model by Cho et al. [15].

Language Modeling: a probability distribution over sequences of words. It is used in speech recognition, sentiment analysis, information retrieval, and other applications [48].

Deep Speech: a Speech-To-Text engine that uses a model trained by machine learning techniques, based on Baidu's Deep Speech research [26].

All benchmarks are run using 16-bit precision arithmetic.

6 EVALUATION

We evaluate the RNNFast performance and energy consumption compared to the NVIDIA GPU, and the CMOS-based and the Memristor-based RNNFast design. We evaluate the reliability of the RNNFast error mitigation. We show an area utilization estimate for different benchmarks. We also include a high-level comparison to other RNN accelerators.

6.1 Performance Improvements and Energy Savings

Figure 14 shows the execution time speedup for RNNFast, RNNFast-CMOS, and ISAAC-RNN relative to the P100 GPU for the seven benchmarks we run. RNNFast speedup relative to the GPU varies between 12× for *im2txt* and 34.5× for *D-speech*, with an average speedup of 21.8×. RNNFast speedups increase with the network size, demonstrating the excellent scalability of the design. For instance, in *mach-trans* we test three different network sizes ranging from 512 to 2,048, We observe speedups increase from 15.4× to 29.3×. This is because the large number of threads required to handle the larger network becomes a bottleneck even for the GPU, whereas RNNFast scales much better.

ISAAC-RNN also brings a substantial speedup relative to the GPU ranging between 1.88× for *im2txt* and 5.8× for *D-speech*. Although this is significant, ISAAC-RNN is more than 6.1× slower than the DWM RNNFast implementation. This is primarily due to the higher latency of the LSTM unit in ISAAC-RNN, which is 7.3× higher than a RNNFast LSTM unit. The higher latency is due to the memristor array read latency (100 ns) and overheads that stem from the ADC/DAC components. Even though a single memristor array can handle up to four neurons, which increases throughput, ISAAC-RNN is still fundamentally slower than RNNFast. RNNFast-CMOS

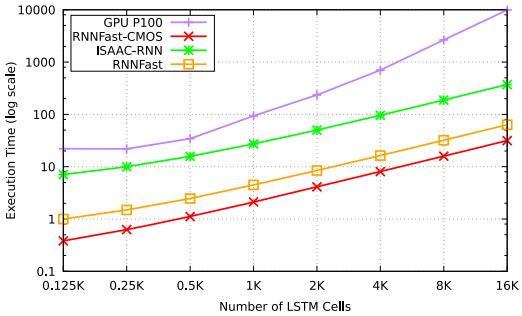

Fig. 16. RNNFast, ISAAC-RNN, and GPU execution times vs. network size for *mach-tran*, normalized to RNNFast 0.125K.

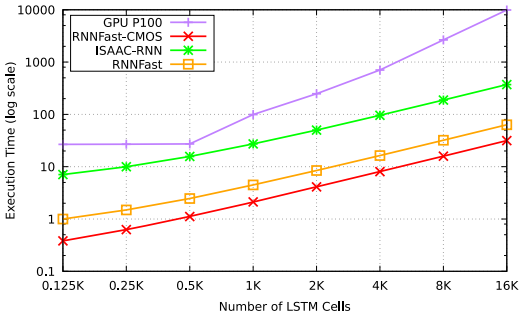

Fig. 17. RNNFast, ISAAC-RNN, and GPU execution times vs. network size for *im2txt*, normalized to RNNFast 0.125K.

shows 2.1× speedup compared to RNNFast. This is due to faster CMOS adders and random memory access instead of the shift-based access in RNNFast.

The PUMA ReRAM-based design is more general than ISSAC and RNNFast, supporting both CNNs and DNNs. However, its performance is lower than both ISAAC-RNN and RNNFast. In general, PUMA tends to have better performance than the GPU for larger networks, especially for multi-layer networks (seq2seq) where PUMA benefits from its pipelined architecture.

Figure 15 shows the energy consumption for RNNFast, RNNFast-CMOS, and ISAAC-RNN relative to the GPU, on a log scale. RNNFast reduces energy consumption on average by 70×. This is due to a much faster execution time achieved with about 1/3 the power of a GPU. The RNNFast-CMOS design has 55% higher energy compared to RNNFast. This reaches a 100% increase for *D-speech* due to higher resource demand, which increases the leakage energy for both compute and memory logic in CMOS. This causes the CMOS design to reach its maximum thermal design power (TDP) at smaller network sizes. ISAAC-RNN also has higher energy usage than RNNFast due to its ADC/DAC and CMOS logic. PUMA energy consumption is much lower than the GPU. However, as expected, it is not lower than ISAAC-RNN. RNNFast is much more energy efficient, using about 25% the energy of PUMA.

RNNFast offers a much more scalable design relative to a GPU due to its modularity and very high storage density of DWM. Figure 16 shows the log scale of execution time for the *mach-tran* benchmark as a function of problem (neural network) size ranging from 128 nodes to 16K nodes per layer in a single-layer configuration. For problem sizes larger than 16K, the GPU runs fail because the device runs out of memory. The GPU execution time exhibits a super-linear increase in execution time with problem size due to memory pressure. RNNFast is consistently faster than the GPU with an improvement that ranges from 13.9× (0.5K) to 156× (16K). RNNFast also scales better to very large problem sizes of 16K nodes and beyond. ISAAC-RNN also scales well, but it is 6.2× slower than RNNFast on average for *mach-tran*. RNNFast-CMOS shows almost 2× speedup over RNNFast. However, this speedup comes at the cost of a much higher energy.

Figure 17 shows a similar trend for *im2txt*. The GPU shows good performance up to 0.5K, but runtime increases exponentially beyond that.

6.2 Error Mitigation

We also evaluate RNNFast resilience to position errors. Figure 18 shows the accuracy of the output as evaluated by the BLEU metric [45], as a function of the probability of position errors. We can see that for a relatively low probability of errors of 4.5×10^{-7}, the output accuracy is virtually unaffected. This is primarily due to the inherent robustness of the RNN to errors. However, without

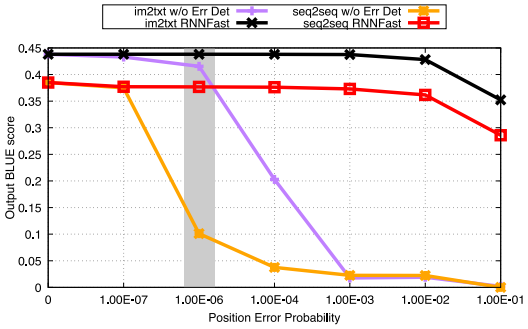

Fig. 18. Output accuracy for benchmarks *im2txt* and *seq2seq* with and without RNNFast EDC.

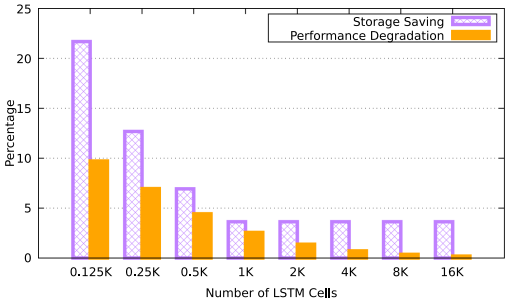

Fig. 19. Storage savings and performance degradation for different network sizes for Approx. Function-based sigmoid design relative to LUT.

error mitigation, the output accuracy degrades substantially at higher errors rates. In the region around 4.5×10^{-5} (highlighted region), which is the expected rate for single bit position errors, the output accuracy drops to 45% for *im2txt* and 10% for *seq2seq*, an unacceptable performance for most applications. When RNNFast error mitigation is enabled, the drop in output accuracy is negligible at less than 2%.

The RNNFast error mitigation produces outputs with less than 5% accuracy loss even for much higher error rates of 10^{-3} or around 20% accuracy loss for 10^{-2}. This shows that RNNFast EDC is robust to much higher error rates than what is expected for DWM technology.

It is also worth highlighting the fact that error mitigation incurs no performance penalty even when errors are detected. Correction or mitigation is performed without stalling the execution pipeline. This is an important design consideration because of the highly synchronized nature of the design. A single stall to correct an error would result in lost cycles for thousands of functional units.

6.3 Nonlinear Function Hardware

We evaluate two designs for the nonlinear function hardware: a LUT-based implementation, and an approximate logic function-based unit. The function-based implementation is area efficient since it does not require as much storage as the LUT-based design. While the function-based implementation is slower than the simple lookup of the LUT version, the activation functions are not a significant latency bottleneck. The advantage for our design is the area reduction. At this scale we have thousands of nonlinear units on chip and reducing their area adds up to real savings.

Figure 19 shows the storage savings and performance degradation of the function-based sigmoid/tanh relative to the LUT design for multiple network sizes. The storage savings diminish as the network size increases because the storage space for the weights dominates. For large networks the storage savings are about 4%, which represents >1 GB of DWM for a 16K network. As for the performance cost, it starts at about 9%, but falls below 1% for larger networks. The approximated nonlinear function does not result in loss of accuracy as measured by the BLEU score.

6.4 RNNFast Parameter Tuning

We also conduct a sensitivity analysis on the number of LSTM units per tile. Figure 20 illustrates the tile input buffer energy versus different number of LSTMs per tile for different network sizes. As the number of LSTMs per tile increases, the power/area overhead for the within tile bus increases super-linearly. The minimum energy point is different depending on the size of the network. The 64 LSTM units per tile represents a reasonable compromise for medium-to-large networks.

Table 2. Summary of the Benchmarks Evaluated

Bench.	Platform	Precision	Layers× Neurons	Time-step	Description
im2txt	DeepBench	16 bit	1 × 512	11	Image caption
seq2seq	DeepBench	16 bit	3 × 1024	15	Language translation
mach-tran	DeepBench	16 bit	1 × 512 1 × 1024 1 × 2048	25	Machine translation
lang-mod	DeepBench	16 bit	1 × 1536	50	Language modeling
D-Speech	DeepBench	16 bit	1 × 2816	1500	Deep Speech

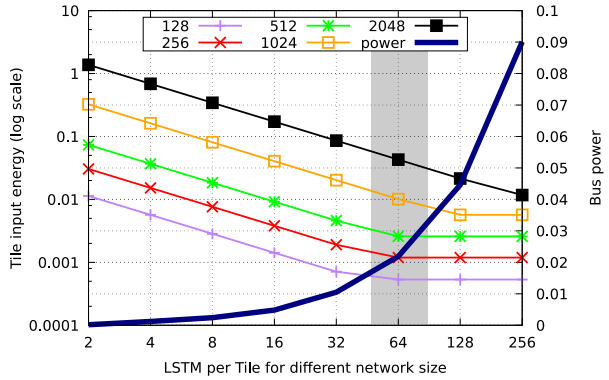

Fig. 20. Sensitivity analysis for the number of LSTMs per tile.

Table 3. Energy and Runtime for FPGA-Based RNNs

FPGA Design	Net size	Timesteps	run time(μs)	energy (μJ)	RNNFast runtime (μs)	RNNFast energy (μJ)
[18]	32	1	1.586	0.8	0.332	0.0419
[41]	256	7,735	42.48E3	NA	2.13E3	1.28E3
[24]	1,024	1	82.7	NA	1.29	12.8
[19]	256-1k-2K	150-25-25	425-74-74	Est.: 425-1,091-4,356	117-58-110.7	252-643-2,575

6.5 Comparison to Other RNN Accelerators

Several recent papers have proposed FPGA-based accelerators for RNNs [18, 19, 24, 34, 36, 41, 56, 68, 73]. We provide a qualitative comparison with some of the more recent ones, for which runtime and energy numbers were available and similar applications were evaluated. Table 3 summarizes the energy and runtime for FPGA-based designs from [18, 19, 24, 41] as well as the energy and runtime of RNNFast while running networks of equivalent size.

The networks used in [18, 24, 41] vary from very small to large. RNNFast shows from 4.7× to 64× speedup. Compared to [18] RNNFast has 19× less energy consumption.

Recently Fowers et al. [19] introduced Brainwave, an FPGA-based accelerator for RNN with no batching for real time AI. While a very efficient design, Brainwave has 50–70% higher energy than RNNFast. Brainwave also shows poorer performance for smaller networks, but slightly better

performance for large ones, compared to RNNFast. Note that this is not a quantitative apples-to-apples comparison to our design given that Brainwave uses 8-bit precision (vs. 16-bit for RNNFast) and a 14-nm technology node (vs. 32-nm for RNNFast).

The Google TPU is also capable of running RNN workloads efficiently. In [31] they report up to 8× better performance for LSTM workloads compared to NVIDIA K80. RNNFast is up to 260× faster than the newer NVIDIA P100 for workloads of similar size.

7 OTHER RELATED WORK

Many customized accelerators for machine learning algorithms and DNNs have been proposed recently [4, 11–14, 16, 17, 25, 32, 37–39, 50, 51]. The majority of this work focuses on improving the performance of CNNs, exploring the potential for resources sharing, leveraging emerging memory technologies, optimizing basic operations, and developing domain-specific methods.

Han et al. [25] used compression of the network model to reduce the memory footprint and accelerate real-time networks in which batching cannot be employed to improve data reuse. Eyeriss [12] explored local data reuse of filter weights and activations in high-dimensional convolutions in order to minimize the energy of data movement.

Emerging memory technologies and in-memory processing have been leveraged for CNN designs to address memory latency limitations and to implement custom logic. PRIME [14] combined processor-in-memory architecture and ReRAM-based neural network computation. The crossbar array structure in ReRAM can be used to perform matrix-vector multiplication as well as regular memory to increase memory space. PUMA [6] is a recently proposed general-purpose and Instruction Set Architecture (ISA)-programmable accelerator built with ReRAM. It has a spatial architecture organized in cores, tiles, and nodes. PUMA features a microarchitecture, ISA, and compiler co-designed to optimize data movement and maximize energy and area efficiency. The PUMA design is more general than ISAAC [51], and, as a result, it generally performs worse in terms of throughput and energy efficiency. ReRAM-based DNN accelerators benefit from the speed and efficiency of the memristor crossbar; however, the need for additional peripheral circuits such as ADCs and DACs, and other components, reduce the benefits of crossbar-based computation.

Neurocube [32] proposed a programmable and scalable digital neuromorphic architecture based on 3D high-density memory integrated with a logic tier for efficient neural computing. The design in [40] also used ReRAM crossbar for RNN acceleration for a case of human activity detection with a small network size of 100 and simple vanilla RNN. Cambricon [39] propose a novel domain-specific ISA for neural network accelerators. PuDianNao [38] focuses on a range of popular machine learning algorithms. However, all these optimizations are CNN/DNN specific. Chung et al. [16] used DWM for CNN computations as well. They proposed a new design that replaces the ReRAM crossbar with a DWM-based CNN layer for dot product. However, they still use costly ADC/DAC circuits and also did not address DWM shift errors in their design.

8 CONCLUSION

The unprecedented growth of available data is accelerating the adoption of deep learning across a wide range of applications including speech recognition, machine translation, and language modeling. In this study, we present RNNFast, a novel accelerator designed for recurrent neural networks. Our design demonstrates that using domain wall memory is not only feasible, but also very efficient. We compare RNNFast with a state-of-the-art P100 NVIDIA GPU and find 21.8× better performance with 70× lower energy.

ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers for their constructive feedback.

REFERENCES

[1] [n.d.]. DeepBench. Retrieved on August 12, 2020 from https://svail.github.io/DeepBench/.

[2] [n.d.]. NVIDIA CUDA Deep Neural Network library. Retrieved on August 12, 2020 from https://developer.nvidia.com/cudnn.

[3] [n.d.]. Optimizing Recurrent Neural Networks in cuDNN 5. Retrieved on August 12, 2020 from https://devblogs.nvidia.com/parallelforall/optimizing-recurrent-neural-networks-cudnn-5/.

[4] Jorge Albericio, Patrick Judd, Tayler Hetherington, Tor Aamodt, Natalie Enright Jerger, and Andreas Moshovos. 2016. Cnvlutin: Ineffectual-neuron-free deep neural network computing. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA'16)*. IEEE, 1–13.

[5] Dario Amodei, Rishita Anubhai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Jingdong Chen, Mike Chrzanowski, Adam Coates, Greg Diamos, Erich Elsen, Jesse Engel, Linxi Fan, Christopher Fougner, Awni Y. Hannun, Billy Jun, Tony Han, Patrick LeGresley, Xiangang Li, Libby Lin, Sharan Narang, Andrew Y. Ng, Sherjil Ozair, Ryan Prenger, Sheng Qian, Jonathan Raiman, Sanjeev Satheesh, David Seetapun, Shubho Sengupta, Chong Wang, Yi Wang, Zhiqian Wang, Bo Xiao, Yan Xie, Dani Yogatama, Jun Zhan, and Zhenyao Zhu. 2016. Deep speech 2: End-to-end speech recognition in English and Mandarin. In *Proceedings of the 33nd International Conference on Machine Learning (ICML'16)*. 173–182. http://jmlr.org/proceedings/papers/v48/amodei16.html

[6] Aayush Ankit, Izzat El Hajj, Sai Rahul Chalamalasetti, Geoffrey Ndu, Martin Foltin, R. Stanley Williams, Paolo Faraboschi, Wen-mei W. Hwu, John Paul Strachan, Kaushik Roy, et al. 2019. PUMA: A programmable ultra-efficient memristor-based accelerator for machine learning inference. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems*. 715–731.

[7] Aayush Ankit, Abhronil Sengupta, Priyadarshini Panda, and Kaushik Roy. 2017. RESPARC: A reconfigurable and energy-efficient architecture with memristive crossbars for deep spiking neural networks. *Arxiv Preprint Arxiv:1702.06064*.

[8] A. J. Annunziata, M. C. Gaidis, L. Thomas, C. W. Chien, C. C. Hung, P. Chevalier, E. J. O'Sullivan, J. P. Hummel, E. A. Joseph, Y. Zhu, T. Topuria, E. Delenia, P. M. Rice, S. S. P. Parkin, and W. J. Gallagher. 2011. Racetrack memory cell array with integrated magnetic tunnel junction readout. In *2011 International Electron Devices Meeting*. 24.3.1–24.3.4. DOI: https://doi.org/10.1109/IEDM.2011.6131604

[9] Elham Azari, Aykut Dengi, and Sarma Vrudhula. 2019. An energy-efficient FPGA implementation of an LSTM network using approximate computing. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'19)*. ACM, New York, 305–306. DOI: https://doi.org/10.1145/3289602.3293989

[10] K. Chang and T. Chang. 2019. VSCNN: Convolution neural network accelerator with vector sparsity. In *2019 IEEE International Symposium on Circuits and Systems (ISCAS'19)*. 1–5. DOI: https://doi.org/10.1109/ISCAS.2019.8702471

[11] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. 2014. DianNao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS'14)*. 269–284. DOI: https://doi.org/10.1145/2541940.2541967

[12] Yu-Hsin Chen, Joel S. Emer, and Vivienne Sze. 2016. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *43rd ACM/IEEE Annual International Symposium on Computer Architecture (ISCA'16)*. 367–379. DOI: https://doi.org/10.1109/ISCA.2016.40

[13] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. 2014. DaDianNao: A machine-learning supercomputer. In *47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'14)*. 609–622. DOI: https://doi.org/10.1109/MICRO.2014.58

[14] Ping Chi, Shuangchen Li, Cong Xu, Tao Zhang, Jishen Zhao, Yongpan Liu, Yu Wang, and Yuan Xie. 2016. PRIME: A novel processing-in-memory architecture for neural network computation in ReRAM-based main memory. In *43rd ACM/IEEE Annual International Symposium on Computer Architecture (ISCA'16)*. 27–39. DOI: https://doi.org/10.1109/ISCA.2016.13

[15] Kyunghyun Cho, Bart van Merrienboer, Çaglar Gülçehre, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *CoRR* abs/1406.1078 (2014). http://arxiv.org/abs/1406.1078.

[16] Jinil Chung, Jongsun Park, and Swaroop Ghosh. 2016. Domain wall memory based convolutional neural networks for bit-width extendability and energy-efficiency. In *Proceedings of the 2016 International Symposium on Low Power Electronics and Design (ISLPED'16)*. 332–337. DOI: https://doi.org/10.1145/2934583.2934602

[17] Zidong Du, Robert Fasthuber, Tianshi Chen, Paolo Ienne, Ling Li, Tao Luo, Xiaobing Feng, Yunji Chen, and Olivier Temam. 2015. ShiDianNao: Shifting vision processing closer to the sensor. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. 92–104. DOI: https://doi.org/10.1145/2749469.2750389

[18] J. C. Ferreira and J. Fonseca. 2016. An FPGA implementation of a long short-term memory neural network. In *2016 International Conference on ReConFigurable Computing and FPGAs (ReConFig'16)*. 1–8. DOI: https://doi.org/10.1109/ReConFigure2016.7857151

[19] J. Fowers, K. Ovtcharov, M. Papamichael, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, L. Adams, M. Ghandi, S. Heil, P. Patel, A. Sapek, G. Weisz, L. Woods, S. Lanka, S. K. Reinhardt, A. M. Caulfield, E. S. Chung, and D. Burger. 2018. A configurable cloud-scale DNN processor for real-time AI. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA'18)*. 1–14. DOI: https://doi.org/10.1109/ISCA.2018.00012

[20] S. Ghosh. 2013. Design methodologies for high density domain wall memory. In *2013 IEEE/ACM International Symposium on Nanoscale Architectures (NANOARCH'13)*. 30–31. DOI: https://doi.org/10.1109/NanoArch.2013.6623035

[21] Alex Graves, Abdel-rahman Mohamed, and Geoffrey E. Hinton. 2013. Speech recognition with deep recurrent neural networks. In *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP'13)*. 6645–6649. DOI: https://doi.org/10.1109/ICASSP.2013.6638947

[22] Klaus Greff, Rupesh K. Srivastava, Jan Koutník, Bas R. Steunebrink, and Jürgen Schmidhuber. 2017. LSTM: A search space odyssey. *IEEE Transactions on Neural Networks and Learning Systems* 28, 10 (Oct. 2017), 2222–2232.

[23] Y. Guan, Z. Yuan, G. Sun, and J. Cong. 2017. FPGA-based accelerator for long short-term memory recurrent neural networks. In *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC'17)*. 629–634. DOI: https://doi.org/10.1109/ASPDAC.2017.7858394

[24] Song Han, Junlong Kang, Huizi Mao, Yiming Hu, Xin Li, Yubin Li, Dongliang Xie, Hong Luo, Song Yao, Yu Wang, et al. 2017. 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, 75–84.

[25] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. 2016. EIE: Efficient inference engine on compressed deep neural network. In *43rd ACM/IEEE Annual International Symposium on Computer Architecture (ISCA'16)*. 243–254. DOI: https://doi.org/10.1109/ISCA.2016.30

[26] Awni Y. Hannun, Carl Case, Jared Casper, Bryan Catanzaro, Greg Diamos, Erich Elsen, Ryan Prenger, Sanjeev Satheesh, Shubho Sengupta, Adam Coates, and Andrew Y. Ng. 2014. Deep speech: Scaling up end-to-end speech recognition. *CoRR* abs/1412.5567 (2014). http://arxiv.org/abs/1412.5567.

[27] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural Computation* 9, 8 (1997), 1735–1780. DOI: https://doi.org/10.1162/neco.1997.9.8.1735

[28] Kejie Huang, Rong Zhao, and Yong Lian. 2016. Racetrack memory-based nonvolatile storage elements for multi-context FPGAs. *IEEE Transactions on VLSI Systems* 24, 5 (2016), 1885–1894. DOI: https://doi.org/10.1109/TVLSI.2015.2474706

[29] Anirudh Iyengar and Swaroop Ghosh. 2014. Modeling and analysis of domain wall dynamics for robust and low-power embedded memory. In *Proceedings of the 51st Annual Design Automation Conference (DAC'14)*. ACM, New York, Article 65, 6 pages. DOI: https://doi.org/10.1145/2593069.2593161

[30] Tian Jin and Seokin Hong. 2019. Split-CNN: Splitting window-based operations in convolutional neural networks for memory system optimization. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'19)*. ACM, New York, 835–847. DOI: https://doi.org/10.1145/3297858.3304038

[31] Norman P. Jouppi, et. al. 2017. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA'17)*. ACM, New York, 1–12. DOI: https://doi.org/10.1145/3079856.3080246

[32] Duckhwan Kim, Jaeha Kung, Sek M. Chai, Sudhakar Yalamanchili, and Saibal Mukhopadhyay. 2016. Neurocube: A programmable digital neuromorphic architecture with high-density 3D memory. In *43rd ACM/IEEE Annual International Symposium on Computer Architecture (ISCA'16)*. 380–392. DOI: https://doi.org/10.1109/ISCA.2016.41

[33] H. T. Kung, Bradley McDanel, and Sai Qian Zhang. 2019. Packing sparse convolutional neural networks for efficient systolic array implementations: Column combining under joint optimization. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'19)*. ACM, New York, 821–834. DOI: https://doi.org/10.1145/3297858.3304028

[34] Chen-Lu Li, Yu-Jie Huang, Yu-Jie Cai, Jun Han, and Xiao-Yang Zeng. 2018. FPGA implementation of LSTM based on automatic speech recognition. In *Proceedings of the 2018 14th IEEE International Conference on Solid-State and Integrated Circuit Technology (ICSICT'18)*. IEEE, 1–3.

[35] S. Li, C. Wu, H. Li, B. Li, Y. Wang, and Q. Qiu. 2015. FPGA acceleration of recurrent neural network based language model. In *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*. 111–118. DOI: https://doi.org/10.1109/FCCM.2015.50

[36] Z. Li, C. Ding, S. Wang, W. Wen, Y. Zhuo, C. Liu, Q. Qiu, W. Xu, X. Lin, X. Qian, and Y. Wang. 2019. E-RNN: Design optimization for efficient recurrent neural networks in FPGAs. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA'19)*. 69–80. DOI: https://doi.org/10.1109/HPCA.2019.00028

[37] Robert LiKamWa, Yunhui Hou, Yuan Gao, Mia Polansky, and Lin Zhong. 2016. RedEye: Analog convnet image sensor architecture for continuous mobile vision. In *43rd ACM/IEEE Annual International Symposium on Computer Architecture (ISCA'16)*. 255–266. DOI: https://doi.org/10.1109/ISCA.2016.31

[38] Dao-Fu Liu, Tianshi Chen, Shaoli Liu, Jinhong Zhou, Shengyuan Zhou, Olivier Temam, Xiaobing Feng, Xuehai Zhou, and Yunji Chen. 2015. PuDianNao: A polyvalent machine learning accelerator. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'15)*. 369–381. DOI: https://doi.org/10.1145/2694344.2694358

[39] Shaoli Liu, Zidong Du, Jinhua Tao, Dong Han, Tao Luo, Yuan Xie, Yunji Chen, and Tianshi Chen. 2016. Cambricon: An instruction set architecture for neural networks. In *43rd ACM/IEEE Annual International Symposium on Computer Architecture (ISCA'16)*. 393–405. DOI: https://doi.org/10.1109/ISCA.2016.42

[40] Y. Long, E. M. Jung, J. Kung, and S. Mukhopadhyay. 2016. ReRAM crossbar based recurrent neural network for human activity detection. In *Proceedings of the 2016 International Joint Conference on Neural Networks (IJCNN'16)*. 939–946. DOI: https://doi.org/10.1109/IJCNN.2016.7727299

[41] Thomas Mealey and Tarek M. Taha. 2018. Accelerating inference in long short-term memory neural networks. In *Proceedings of the NAECON 2018 IEEE National Aerospace and Electronics Conference*. IEEE, 382–390.

[42] Seyedhamidreza Motaman and Swaroop Ghosh. 2016. Adaptive write and shift current modulation for process variation tolerance in domain wall caches. *IEEE Transactions on VLSI Systems* 24, 3 (2016), 944–953. DOI: https://doi.org/10.1109/TVLSI.2015.2437283

[43] Seyedhamidreza Motaman, Anirudh Iyengar, and Swaroop Ghosh. 2014. Synergistic circuit and system design for energy-efficient and robust domain wall caches. In *International Symposium on Low Power Electronics and Design (ISLPED'14)*. 195–200. DOI: https://doi.org/10.1145/2627369.2627643

[44] Seyedhamidreza Motaman, Anirudh Srikant Iyengar, and Swaroop Ghosh. 2015. Domain wall memory-layout, circuit and synergistic systems. *IEEE Transactions on Nanotechnology* 14, 2 (March 2015), 282–291. DOI: https://doi.org/10.1109/TNANO.2015.2391185

[45] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. BLEU: A method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics (ACL'02)*. Association for Computational Linguistics, 311–318. DOI: https://doi.org/10.3115/1073083.1073135

[46] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W. Keckler, and William J. Dally. 2017. SCNN: An accelerator for compressed-sparse convolutional neural networks. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA'17)*. ACM, New York, 27–40. DOI: https://doi.org/10.1145/3079856.3080254

[47] Stuart S. P. Parkin, Masamitsu Hayashi, and Luc Thomas. 2008. Magnetic domain-wall racetrack memory. *Science* 320, 5873 (2008), 190–194. DOI: https://doi.org/10.1126/science.1145799 arXiv:http://science.sciencemag.org/content/320/5873/190. full.pdf

[48] Jay M. Ponte and W. Bruce Croft. 1998. A language modeling approach to information retrieval. In *SIGIR'98: Proceedings of the 21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. 275–281. DOI: https://doi.org/10.1145/290941.291008

[49] A. Ranjan, S. G. Ramasubramanian, R. Venkatesan, V. Pai, K. Roy, and A. Raghunathan. 2015. DyReCTape: A dynamically reconfigurable cache using domain wall memory tapes. In *Proceedings of the 2015 Design, Automation Test in Europe Conference Exhibition (DATE'15)*. 181–186. DOI: https://doi.org/10.7873/DATE.2015.0838

[50] Brandon Reagen, Paul N. Whatmough, Robert Adolf, Saketh Rama, Hyunkwang Lee, Sae Kyu Lee, José Miguel Hernández-Lobato, Gu-Yeon Wei, and David M. Brooks. 2016. Minerva: Enabling low-power, highly-accurate deep neural network accelerators. In *ISCA*.

[51] Ali Shafiee, Anirban Nag, Naveen Muralimanohar, Rajeev Balasubramonian, John Paul Strachan, Miao Hu, R. Stanley Williams, and Vivek Srikumar. 2016. ISAAC: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars. In *43rd ACM/IEEE Annual International Symposium on Computer Architecture (ISCA'16)*. 14–26. DOI: https://doi.org/10.1109/ISCA.2016.12

[52] Yongming Shen, Michael Ferdman, and Peter Milder. 2017. Maximizing CNN accelerator efficiency through resource partitioning. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA'17)*. ACM, New York, 535–547. DOI: https://doi.org/10.1145/3079856.3080221

[53] Clinton W. Smullen, Anurag Nigam, Sudhanva Gurumurthi, and Mircea R. Stan. 2011. The STeTSiMS STT-RAM simulation and modeling system. In *ICCAD*. IEEE Press, 318–325.

[54] Z. Sun, X. Bi, W. Wu, S. Yoo, and H. Li. 2016. Array organization and data management exploration in racetrack memory. *IEEE Transactions on Computers* 65, 4 (April 2016), 1041–1054. DOI: https://doi.org/10.1109/TC.2014.2360545

[55] Zhenyu Sun, Wenqing Wu, and Hai (Helen) Li. 2013. Cross-layer racetrack memory design for ultra high density and low power consumption. In *Proceedings of the 50th Annual Design Automation Conference (DAC'13)*. ACM, New York, Article 53, 6 pages. DOI: https://doi.org/10.1145/2463209.2488799

[56] Zhanrui Sun, Yongxin Zhu, Yu Zheng, Hao Wu, Zihao Cao, Peng Xiong, Junjie Hou, Tian Huang, and Zhiqiang Que. 2018. FPGA acceleration of LSTM based on data for test flight. In *Proceedings of the 2018 IEEE International Conference on Smart Cloud (SmartCloud'18)*. IEEE, 1–6.

[57] M. T. Tommiska. 2003. Efficient digital implementation of the sigmoid function for reprogrammable logic. *IEE Proceedings-Computers and Digital Techniques* 150, 6 (2003), 403–411.

[58] Antonio Toral and Andy Way. 2018. What level of quality can neural machine translation attain on literary text? In *Translation Quality Assessment*. Springer, 263–287.

[59] Swagath Venkataramani, Ashish Ranjan, Subarno Banerjee, Dipankar Das, Sasikanth Avancha, Ashok Jagannathan, Ajaya Durg, Dheemanth Nagaraj, Bharat Kaul, Pradeep Dubey, and Anand Raghunathan. 2017. ScaleDeep: A scalable compute architecture for learning and evaluating deep networks. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA'17)*. ACM, New York, 13–26. DOI: https://doi.org/10.1145/3079856.3080244

[60] Rangharajan Venkatesan, Vivek J. Kozhikkottu, Mrigank Sharad, Charles Augustine, Arijit Raychowdhury, Kaushik Roy, and Anand Raghunathan. 2016. Cache design with domain wall memory. *IEEE Transactions on Computers* 65, 4 (April 2016), 1010–1024. DOI: https://doi.org/10.1109/TC.2015.2506581

[61] R. Venkatesan, M. Sharad, K. Roy, and A. Raghunathan. 2013. DWM-TAPESTRI - An energy efficient all-spin cache using domain wall shift based writes. In *Proceedings of the 2013 Design, Automation Test in Europe Conference Exhibition (DATE'13)*. 1825–1830. DOI: https://doi.org/10.7873/DATE.2013.365

[62] Oriol Vinyals, Alexander Toshev, Samy Bengio, and Dumitru Erhan. 2016. Show and tell: Lessons learned from the 2015 MSCOCO image captioning challenge. *CoRR* abs/1609.06647 (2016). http://arxiv.org/abs/1609.06647.

[63] M. Wang, Z. Wang, J. Lu, J. Lin, and Z. Wang. 2019. E-LSTM: An efficient hardware architecture for long short-term memory. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* (2019), 1–1. DOI: https://doi.org/10.1109/JETCAS.2019.2911739

[64] Shuo Wang, Zhe Li, Caiwen Ding, Bo Yuan, Qinru Qiu, Yanzhi Wang, and Yun Liang. 2018. C-LSTM: Enabling efficient LSTM using structured compression techniques on FPGAs. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'18)*. ACM, New York, 11–20. DOI: https://doi.org/10.1145/3174243.3174253

[65] X. Wang, J. Yu, C. Augustine, R. Iyer, and R. Das. 2019. Bit prudent in-cache acceleration of deep convolutional neural networks. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA'19)*. 81–93. DOI: https://doi.org/10.1109/HPCA.2019.00029

[66] Yuhao Wang, Hao Yu, Leibin Ni, Guang-Bin Huang, Mei Yan, Chuliang Weng, Wei Yang, and Junfeng Zhao. 2015. An energy-efficient nonvolatile in-memory computing architecture for extreme learning machine by domain-wall nanowire devices. *IEEE Transactions on Nanotechnology* 14, 6 (2015), 998–1012.

[67] Yuhao Wang, Hao Yu, Dennis Sylvester, and Pingfan Kong. 2014. Energy efficient in-memory AES encryption based on nonvolatile domain-wall nanowire. In *Design, Automation & Test in Europe Conference & Exhibition (DATE'14)*. 1–4. DOI: https://doi.org/10.7873/DATE.2014.196

[68] Zhisheng Wang, Jun Lin, and Zhongfeng Wang. 2017. Accelerating recurrent neural networks: A memory-efficient approach. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 25, 10 (2017), 2763–2775.

[69] Cong Xu, Dimin Niu, Xiaochun Zhu, Seung H. Kang, Matt Nowak, and Yuan Xie. 2011. Device-architecture co-optimization of STT-RAM based memory for low power embedded systems. In *ICCAD*. IEEE Press, 463–470.

[70] Hao Yu, Yuhao Wang, Shuai Chen, Wei Fei, Chuliang Weng, Junfeng Zhao, and Zhulin Wei. 2014. Energy efficient in-memory machine learning for data intensive image-processing by non-volatile domain-wall memory. In *Proceedngs of the 19th Asia and South Pacific Design Automation Conference (ASP-DAC'14)*. 191–196. DOI: https://doi.org/10.1109/ASPDAC.2014.6742888

[71] Chao Zhang, Guangyu Sun, Weiqi Zhang, Fan Mi, Hai Li, and W. Zhao. 2015. Quantitative modeling of racetrack memory, a tradeoff among area, performance, and power. In *Proceedings of the 20th Asia and South Pacific Design Automation Conference*. 100–105. DOI: https://doi.org/10.1109/ASPDAC.2015.7058988

[72] Chao Zhang, Guangyu Sun, Xian Zhang, Weiqi Zhang, Weisheng Zhao, Tao Wang, Yun Liang, Yongpan Liu, Yu Wang, and Jiwu Shu. 2015. Hi-fi playback: Tolerating position errors in shift operations of racetrack memory. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA'15)*. ACM, New York, 694–706. DOI: https://doi.org/10.1145/2749469.2750388

[73] Yiwei Zhang, Chao Wang, Lei Gong, Yuntao Lu, Fan Sun, Chongchong Xu, Xi Li, and Xuehai Zhou. 2017. A power-efficient accelerator based on FPGAs for LSTM network. In *Proceedings of the 2017 IEEE International Conference on Cluster Computing (CLUSTER'17)*. IEEE, 629–630.

[74] Y. Zhang, C. Zhang, J. Nan, Z. Zhang, X. Zhang, J. O. Klein, D. Ravelosona, G. Sun, and W. Zhao. 2016. Perspectives of racetrack memory for large-capacity on-chip memory: From device to system. *IEEE Transactions on Circuits and Systems I: Regular Papers* 63, 5 (May 2016), 629–638. DOI: https://doi.org/10.1109/TCSI.2016.2529240

[75] Weisheng Zhao, Nesrine Ben Romdhane, Yue Zhang, Jacques-Olivier Klein, and Define Ravelosona. 2013. Racetrack memory based reconfigurable computing. In *Proceedings of the 2013 IEEE Faible Tension Faible Consommation (FTFC'13)*. IEEE, 1–4.

Received May 2019; revised February 2020; accepted May 2020