CSB-RNN: A Faster-than-Realtime RNN Acceleration Framework with Compressed Structured Blocks

Runbin Shi¹²⁺†, Peiyan Dong²†, Tong Geng³†, Yuhao Ding¹, Xiaolong Ma², Hayden K.-H. So¹, Martin Herbordt³, Ang Li⁴, Yanzhi Wang⁵
¹The University of Hong Kong, ²Northeastern University, ³Boston University, ⁴Pacific Northwest National Laboratory
{rbshi,yhdng,hs}@eee.hku.hk, {dong.pe,ma.xiao}@husky.neu.edu, {tgeng,herbordt}@bu.edu, ang.li@pnnl.gov
yanz.wang@northeastern.edu

ABSTRACT

Recurrent neural networks (RNNs) have been widely adopted in temporal sequence analysis, where realtime performance is often in demand. However, RNNs suffer from heavy computational workload as the model often comes with large weight matrices. Pruning (a model compression method) schemes have been proposed for RNNs to eliminate the redundant (close-to-zero) weight values. On one hand, the non-structured pruning methods achieve a high pruning rate but introducing computation irregularity (random sparsity), which is unfriendly to parallel hardware. On the other hand, hardware-oriented structured pruning suffers from low pruning rate due to restricted constraints on allowable pruning structure.

This paper presents CSB-RNN, an optimized full-stack RNN framework with a novel compressed structured block (CSB) pruning technique. The CSB pruned RNN model comes with both fine pruning granularity that facilitates a high pruning rate and regular structure that benefits the hardware parallelism. To address the challenges in parallelizing the CSB pruned model inference with fine-grained structural sparsity, we propose a novel hardware architecture with a dedicated compiler. Gaining from the architecture-compilation co-design, the hardware not only supports various RNN cell types, but is also able to address the challenging workload imbalance issue and therefore significantly improves the hardware efficiency (utilization). Compared to the vanilla design without optimizations, the hardware utilization has been enhanced by over 2×. With experiments on 10 RNN models from multiple application domains, CSB pruning demonstrates 3.5×-25× lossless pruning rate, which is 1.6× to 3.9× over existing designs. With several other innovations applied, the CSB-RNN inference can achieve faster-than-realtime latency of 0.79μs-6.58μs in an FPGA implementation, which contributes to 1.12×-12.57× lower latency and 3.53×-58.89× improvement on power-efficiency over the state-of-the-art.

CCS CONCEPTS

- Computer systems organization → Parallel architectures.

KEYWORDS

RNN, structured pruning, workload balancing, FPGA

ACM Reference Format:


1 INTRODUCTION

RNNs have been widely adopted for its high-accuracy on temporal sequence analysis, such as machine translation [4], speech recognition [9], or even stock-price prediction [26]. However, the increasingly large model size and tremendous computational workload of the RNN hampers its deployment on embedded (edge) devices, which strictly demand realtime processing with limited hardware resources. To address this issue, weight pruning techniques [11, 17, 30] have been proposed, which shrink the model size, reduce storage demand, and provide higher potential hardware performance by eliminating the redundant (close-to-zero) weights and the corresponding arithmetic operations in inference.

The weight pruning schemes in some existing works [11, 27] are in a non-structured fashion and with extreme irregularities in the computation, which is unfriendly to either the modern parallel device or the hardware architecture design. Thus the performance degradation from the hardware inefficiency encroaches upon the gains from model compression. Therefore, researchers start to explore other pruning approaches, i.e., structured pruning [19, 21, 30], in which the regular computation patterns are maintained. Although these structured-pruned models are relatively hardware-friendly, the coarse pruning granularity (structure) leads to either a significant degradation on model accuracy or a limited pruning rate (the weight count ratio of the original model to pruned model). To keep the accuracy loss acceptable, the attainable pruning rates delivered in the existing structured pruning schemes are far lower than that the ones with the non-structured pruning, wasting potential pruning opportunities.

In this paper, we aim to overcome the above limitations. We first propose a novel fine-grained structured pruning technique (CSB pruning) that provides a similar compression rate (and test accuracy) as non-structured pruning while offering a higher potential for hardware acceleration than the non-structured methods. During the training phase, each weight matrices are divided into fine-grained blocks, and a structured pruning is conducted on every
Figure 1: Computation flow of RNN inference. Note that there are multiple RNN cell types. The main workload is matrix-vector multiplication (MVM).

block independently. The pruned blocks are encoded in a novel compressed structured block (CSB) sparse format for inference acceleration, which significantly reduces the weight storage demand while retaining the fine-grained content in the original model.

To realize a real-time inference with parallel hardware, there are still multiple challenges to design an architecture that can exploit the benefits of CSB pruning in a seamless manner. In particular, the parallel architecture should handle massive fine-grained blocks with imbalanced workloads (sparsity) but maintain a high hardware efficiency (utilization). Meanwhile, the architecture should be programmable for various RNN cell types (e.g., LSTM [12], GRU [4]), although the existing RNN architectures are designed for a particular cell type. To address the issues above, we propose an architecture-compilation co-design to realize the best flexibility and acceleration performance. A programmable RNN dataflow architecture is designed that supports existing RNN cell types. In particular, the CSB-Engine in our architecture is designed with a novel workload sharing technique. With the one-shot compilation, the workload is automatically balanced among processing elements (PEs) in CSB-Engine, which improves the hardware efficiency to a near theoretical value.

The major contributions are summarized as follows:

- We present CSB-RNN, an optimized full-stack RNN acceleration framework, which facilitates running various types of RNNs with faster-than-realtime latency. CSB-RNN includes three innovations: (1) an adaptive and fine-grained structured compression technique, CSB pruning; (2) a programmable RNN dataflow architecture equipped with CSB-Engine; (3) a compiler design with optimizations to achieve almost perfect workload balance.
- The proposed CSB pruning technique provides ultra-high (3.5×-25×) pruning rates without any loss on accuracy. Furthermore, CSB pruning does not incur high-degree computational irregularities, making highly efficient hardware acceleration possible.
- An architecture-compilation co-design is proposed to sufficiently exploit the benefits of CSB pruning and provide close-to-theoretical peak performance with automatic workload balancing.
- With experiments on 10 RNN models from various application domains, CSB pruning demonstrates 3.5×-25× lossless pruning rate, which is 1.6× to 3.9× over existing designs. With the proposed architecture-compilation co-design applied, the CSB-RNN delivers faster-than-real-time inference with the latency of 0.79µs-6.58µs in an FPGA implementation. The proposed framework

2 BACKGROUND

2.1 Temporal Sequence Processing with RNN

The recurrent neural networks (RNNs) deliver high accuracy in the temporal sequence processing. A typical schematic of RNN computation is depicted in Fig. 1. Successive frames (e.g., word, phoneme) from the temporal sequence (e.g., sentence, voice) are embedded as input neuron-vectors (x_t), and then sent to RNN cells for inference computation. t represents the time point. The output neuron-vector (h_t) contains the inference results (e.g., translation, prediction) that may have different dimensions with x_t.

Multiple RNN cell types exist that are composed of different computational dataflow but almost the same arithmetic primitives. Fig. 1 lists the arithmetic of two widely-used RNN cells, GRU [4] and LSTM [12]. The significant workload is matrix-vector multiplication (MVM) between the weight matrices and input/hidden neurons; And the rest workload is element-wise operations, including Sigmoid (σ)/Tanh (θ) activation function, element-wise multiplication (⊙) and addition. In particular, the RNN cell computation at time t invokes the intermediate vector c_t-1 and output vector h_t-1 from the previous timestamp. The data dependency results in a context link between two successive RNN cell iterations, which helps to memorize the temporal feature during inference.

2.2 RNN Weight Pruning Techniques

2.2.1 Non-structured Pruning v.s. Structured Pruning. The pruning technique has been proposed for deep learning models to reduce redundant (close-to-zero) weights and thus the computation workload. The early non-structured pruning [11] achieves a high pruning rate; however, the random sparse model (Fig. 2(a)) brings a high degree of irregularity to the inference computation, which is unfriendly to either the modern parallel device or the hardware architecture design. Some existing works [3, 10] address this issue by pruning model with region-balanced sparsity (between non-structured and structured sparsity), which reduced the attainable pruning rate. As Fig. 2(b), the structured pruning schemes [7, 31] were proposed for hardware friendly purpose that the entire row/column is removed as a whole in pruning. Although the pruned model maintains the regularity and can even be compacted to a dense matrix, the pruning rate with this scheme is relatively low due to the coarse pruning

Figure 2: CSB pruning takes advantage of both non-structured (random) pruning (a) and coarse-grained structured (row/column) pruning (b).
granularity. With the advantages of both the non-structured and coarse-grained structured pruning methods, the CSB pruning in this work is a fine-grained structured method that not only achieves a high pruning rate but also makes the hardware acceleration possible.

2.2.2 Model Training with ADMM-based Pruning Technique. The training process is to find a proper set of weight values that reach the minimal classification loss compared to the ground truth. The objective of training an N-layer RNN can be formulated as,

$$\min_{\{W_i, b_i\}} f(\{W_i\}_{i=1}^N, \{b_i\}_{i=1}^N)$$

s.t. $W_i \in S_i, \ i = 1, \ldots, N$ \ (1)

where function $f$ represents inference loss on the given dataset, $S_i$ is the feasible set of $W_i$, which is subject to the user constraints. In the regular RNN training, $S_i$ is $\mathbb{R}$ (i.e., no constraint), and thus the optimal weights ($W_i$) and bias ($b_i$) for each layer can be obtained by classical stochastic gradient descent (SGD) method [1]. However, once the weight pruning is conducted along with the training process, the constraint of weight-sparsity represented by $S_i$ becomes combinatorial and no longer convex, which prevents the Eqn. 1 from being solved by classical SGD. The advanced Alternating Direction Method of Multipliers (ADMM) method [2] is leveraged in our CSB pruning scheme. The ADMM separates the training process, the constraint of weight-sparsity represented by $S_i$, becomes combinatorial and no longer convex, which prevents the Eqn. 1 from being solved by classical SGD. The advanced Alternating Direction Method of Multipliers (ADMM) method [2] is leveraged in our CSB pruning scheme. The ADMM separates the training process into two subproblems, which are iteratively solved until convergence. First, the problem is reformulated as,

$$\min_{\{W_i, b_i\}} f(\{W_i\}_{i=1}^N, \{b_i\}_{i=1}^N) + \sum_{i=1}^N g_i(Z_i)$$

s.t. $W_i = Z_i, \ i = 1, \ldots, N$ \ (2)

where $Z_i$ is an auxiliary variable for subproblem decomposition, and the indicator function (Eqn. 3) is used to replace the original constraint on feasible set.

$$g_i(Z_i) = \begin{cases} 0 & \text{if } W_i \in S_i, \\ +\infty & \text{otherwise.} \end{cases} \ (3)$$

Then the Eqn. 2 can be decomposed to two subproblems listed in Eqn. 4 and Eqn. 5 with the formation of augmented Lagrangian [6],

$$\min_{\{W_i, b_i\}} f(\{W_i\}_{i=1}^N, \{b_i\}_{i=1}^N) + \sum_{i=1}^N \frac{\rho_i}{2} ||W_i - Z_i + U_i^t||_F^2$$

$$\min_{\{Z_i\}} g_i(Z_i) + \sum_{i=1}^N \frac{\rho_i}{2} ||W_i^{t+1} - Z_i + U_i^t||_F^2$$

where $t$ denotes the iteration index in the ADMM process, $U_i^t$ is the dual variable that is updated in each iteration through $U_i^{t+1} = U_i^{t-1} + W_i^{t-1} - Z_i^{t-1}$. Following the ADMM process, the two subproblems are iteratively solved till convergence. The first subproblem (Eqn. 4) is solved by the classical SGD method, and the solution for the second subproblem (Eqn. 5) is obtained by

$$Z_i^{t+1} = \text{proj}(W_i^{t+1} + U_i^t, S_i)$$

where proj is the Euclidean projection onto constraint set $S_i$, which guarantees the weight matrices exhibit the specific sparse pattern defined in the constraint $S_i$ for each layer. In this work, we propose a new type of structured sparse matrix with the novel CSB sparse format, which is the target pattern ($S_i$) in our RNN weight pruning method. The detailed projection process for CSB formatted weight will be given in §3.2.
two neurons without a strong connection. The CSB pruning automatically groups the strongly-connected neurons into blocks with high density while leaving the weakly-connected ones in the low-density blocks. Further, the **pruning granularity** is adjustable via changing the block size. Such that different weight matrices in RNN model can be pruned with various granularities. The above flexibilities enable a high pruning rate while maintaining the model accuracy. On the other hand, the un-pruned weights values in each block compose a dense kernel matrix that makes the inference computation friendly to parallel hardware. Nevertheless, the blocks may have different-sized kernel matrices that result in a workforce imbalance issue while mapping computation of blocks to parallel hardware. This paper carefully addresses this issue with an architecture-compilation co-design in §4 and §5.

### 3.2 CSB Pruning Flow with ADMM

**Algorithm 1**: Auto Lossless CSB Pruning with ADMM

```
input: un-pruned RNN model W; lossless accuracy accu; block size in CSB M × N; weight matrix size W × H initial pruning rate initPR; initial step of pruning rate initPRStep
output: maximally compressed model with CSB pruning Z
// Initialization.
U = 0; Z = W; W' = W; Flag = False PruneRate = initPR, StepPruneRate = initPRStep
// Progressive iteration.
repeat
  foreach t ∈ [0, 100) // Re-train and Pruning Epoch.
    do
      // Solve Eqn. 4 in ADMM (1st subproblem)
      W' = SGDTrain(W', U, Z)
      // Solve Eqn. 5 in ADMM (2nd subproblem)
      // Project weight matrix to CSB pattern S
      Z_{i,j} = Partition(W' + U), i ∈ [0, W], j ∈ [0, H]
      foreach j ∈ [0, H/N) do
        Z_{i,j} = RowPrune(Z_{i,j}, 1 - √(1 - PruneRate))
      foreach i ∈ [0, W/M) do
        Z_{i,j} = ColumnPrune(Z_{i,j}, 1 - √(1 - PruneRate))
      U = U + W' − Z // Update U
    // Set progressive pruning rate.
    if Eval(Z) < accu then
      Flag = True
      StepPruneRate = StepPruneRate/2
      PruneRate = PruneRate × StepPruneRate
    else
      if Flag then
        StepPruneRate = StepPruneRate/2
        PruneRate = PruneRate × StepPruneRate
      initPRStep = 1
    until StepPruneRate ≤ 1/4 initPRStep & Eval(Z) ≥ accu;
```

With the ADMM-based pruning technique in §2.2.2, the weight matrices can be pruned to an arbitrary sparse pattern by defining the constraint S and applying the pattern-specific projection in Eqn. 6. To obtain the RNN model with CSB pattern, we develop the CSB pruning algorithm following the ADMM principle. Further, the maximum pruning rate under lossless constraint is automatically achieved via the **progressive** pruning. The entire CSB pruning flow is presented in Algorithm 1 with carefully specified annotations. Initially, the baseline model (with dense weight matrix W) is obtained via classical SGD training and input to the flow. Note that the bias vector (b) is omitted as the CSB pruning flow does not touch it. The lossless accuracy (accu) is given as the constraint of the progressive pruning. Two input parameters, initial pruning rate (initPR) and initial step of pruning rate reduction (initPRStep) are set for tuning the pruning rate in the progressive flow. We use the **progressive increase** manner in approaching the maximum value of lossless pruning rate. Therefore, we set initPR to a small value (e.g., 4x) as the starting point, which surely meets the lossless constraint. The variables PruneRate and StepPruneRate are initialized to initPR and initPRStep, respectively, at the beginning. In each progressive iteration, the flow performs re-training and pruning on the model with multiple epochs (e.g., 100 in Algorithm 1) to obtain the CSB-formatted weight matrix (Z) with the ADMM-pruning fashion. In each epoch, two subproblems are alternatively solved following the principle of the ADMM-pruning technique in §2.2.2. The function SGDTrain updates the weights with classical SGD (1st subproblem, Eqn. 4), and the subsequent process prunes the weight matrix and projects it to CSB-constrained set (2nd subproblem, Eqn. 5). The process in Algorithm 1 details the projection corresponding to the general representation in Eqn. 6. First, the weight from SGDTrain is partitioned to multiple blocks Z_{i,j} following the CSB method in §3.1. Then the RowPrune process is applied to each block-column independently. Specifically, for each block-column, the ℓ2-norm (accumulate the square of all elements) of each row (with the size of M) is obtained; Then, a row-wise pruning is conducted referring to the ℓ2-norm values. Subsequently, the ColumnPrune is applied to each block-row with the same behavior to RowPrune. Note that the pruning rate in both RowPrune and ColumnPrune is 1 − √(1 − PruneRate), which results in the target PruneRate after the combined processes. Once the CSB-formatted weight matrix Z is obtained, it will be sent to SGDTrain of the next epoch, along with un-pruned weight matrix W' and accumulated difference matrix U. With multiple epochs, weight Z will eventually meet the CSB pattern constraints and achieve good accuracy.

After each progressive iteration, the CSB pruning model is evaluated (Eval(Z)) and compared to the lossless accu. The PruneRate is increased by StepPruneRate in the next iteration if the accu is achieved. Once Eval(Z) < accu, the model is over-pruned and the optimum pruning rate is just between the PruneRate of the two neighboring iterations. Therefore, we reduce StepPruneRate by half and reduce the PruneRate by this new step to further approach the optimum point. The progressive CSB pruning flow terminates until the pruning rate reaches a target precision. For instance, as the last line in Algorithm 1, the flow terminates when the pruning rate precision (StepPruneRate) ≤ 1/4 initPRStep.

### 4 UNIFIED ARCHITECTURE FOR CSB-RNN

#### 4.1 Overview of Acceleration Framework

An overview of the CSB-RNN acceleration framework is illustrated in Fig. 4. Although the CSB pruning (STEP1) shrinks the model size and therefore reduces the computation in inference, parallel
hardware acceleration is still in demand to achieve real-time performance. The challenges in accelerating CSB-RNN are two-fold. First, the architecture should be adaptive to various RNN cell types, i.e., LSTM, GRU, etc. Second, the kernel matrix in fine-grained blocks may not provide enough inner-block parallelism for large-scale hardware. To further improve the concurrency, inter-block parallelism should be leveraged. However, the pruned blocks may have different sparsities, leading to the workload imbalance issue for inter-block parallelism, which usually causes a low utilization of processing element (PE). To address these challenges, CSB-RNN proposes an architecture-compiler co-design. In the architecture aspect (STEP2), we propose a unified RNN dataflow architecture that is programmable for different RNN cell types (§4.2). In particular, a novel CSB-Engine is designed with the support of workload sharing and is equipped in CSB-RNN architecture to address the workload imbalance issue (§4.3). In the compilation aspect (STEP3), we define control instructions for the hardware and propose the compilation algorithms to conduct the particular RNN type computation and balanced workload scheduling (§5).

4.2 Programmable RNN Dataflow Architecture

To generalize the architecture for different RNN cell types, we investigated the existing RNN cells and extracted the arithmetic primitives, which compose the RNN computation in a dataflow fashion. Fig. 5 presents the hardware components in this architecture, where each operation unit serves the corresponding arithmetic primitive. In particular, the CSB-Engine computes the main workload, MVM, with the weight matrices after CSB pruning (CSB-MVM). The units \( \times, + \) are the element-wise multiplication and addition. \( \delta \) and \( \theta \) operate the activation functions Sigmoid and Tanh, respectively. The datapaths (arrows on Fig. 5) interconnect the operation units and on-chip buffers, which transmit the intermediate results and compose the dataflow graph for RNN cell computation. Importantly, RNN dataflow architecture provides the programmable datapath (red arrows on Fig. 5). Thus, the proper operation units can be interconnected by programming control instructions for a particular RNN cell type.

4.3 CSB-Engine

The CSB pruning scheme greatly shrinks the weight matrix size and therefore reduces the main workload in inference. Although

![Figure 5: RNN dataflow architecture. Operation units serve the RNN arithmetic primitives; The programmable datapaths construct the proper dataflow for target RNN cell via instructions.](image_url)

the fine-grained structure of CSB contributes to the regularity and makes efficient hardware acceleration possible, it is still challenging to design a parallel architecture that can fully exploit the benefits of CSB pruning. The challenges in an efficient CSB-Engine design are two-fold. First, both the inner-block and inter-block parallelism should be fully exploited, as the regular inner-block computation provides very limited concurrency with small block size. Second, the inter-block workload imbalance issue exists due to the sparsity varies among blocks. The following §4.3.1 and §4.3.2 address these two challenges, respectively.

4.3.1 Hierarchical Design for Inner- and Inter-Block Parallelism.

As illustrated in Fig. 6, the CSB-Engine design is in a two-level hierarchy, processing element (PE) level and PEGroup level. The hardware instances in each level are organized in a 2D fashion that the architecture is composed of \( K \times L \) PEGroups, and each PEGroup contains \( P \times Q \) PEs. The parallel PEs inside one PEGroup process inner-block multiplication concurrently, while the PEGroups computing different blocks in parallel (inter-block parallelism).

Inside each PEGroup, because the size of CSB kernel matrix \( (m \times n) \) might be larger than that of PE array \( (P \times Q) \), multi-pass processing is required to handle the entire block. Thus, each PEGroup contains a NeuronAccumBuffer, which stores the partial results and sums up with the accumulation of horizontal PEs in each pass. The input neurons required by the current block are preloaded to the BLockNeuronBuffer and broadcasted to the PE array. Each PE column shares the same input neuron as the unpruned weights are vertically aligned in the structured block with CSB pruning. Importantly, the WeightBuffer provides the CSB-formatted weight (Fig. 3), including the weight values (kernel matrix) for PEs, column index for BLockNeuronBuffer to read the proper input neuron, row index for NeuronAccumBuffer to accumulate the multiplication results to proper address in NeuronAccumBuffer, and the kernel matrix size \( (m \times n) \) for the PEGroup control logic which conducts proper pass count in both axes.

In the higher-level of the design hierarchy, the PEGroups process blocks in the row-major order. The PEGroups in one column concurrently compute the vertical blocks. Therefore, the PEGroups in one column share the same partition of input neuron vector, while multi-ports are provided on BLockNeuronBuffer for concurrent access. Similarly, the blocks in horizontal axis are mapped to PEGroups in the same row, with multi-pass processing. After the computation
of each block-row, the results in NeuronAccumBuffers are accumulated in horizontal and output to ReorderLogic to obtain the output neuron vector.

4.3.2 Inter-PEGroup Workload Sharing

Workload Imbalance Challenge: The blocks in CSB pruned model may have different-sized kernel matrices, and the resultant inter-block workload imbalance brings challenges to exploit the inter-block parallelism on hardware. As Fig. 7(b) demonstrates, with the straightforward design, the workload imbalance issue results in low utilization of PEGroups. The presented MVM workloads are allocated to 2 × 2 PEGroups that each contains 4 PEs. During the execution, PEGroup1-3 enter the idle state before the PEGroup4 accomplishes its workload, which results in a severe under-utilization of the overall hardware. In fact, the imbalanced sparsity naturally exists in the RNN models. However, existing works [3, 10] relieve the hardware pain by pruning the model with a region-balanced sparsity compulsively. As a result, the neglect of natural sparsity-imbalance significantly harms the pruning ratio and model accuracy. By contrast, we handle this issue by improving the architecture with the workload sharing technique.

Inter-PEGroup Workload Sharing: The concept of workload sharing is illustrated in Fig. 7(c). Each PEGroup processes not only the originally allocated block but also a partition of block from the neighboring PEGroup, which is arranged with a heavier workload. In the hardware aspect, as Fig. 7(c), dedicated workload sharing paths (red arrows) are set for the inter-PEGroup data transmission, and the interconnection adopts the torus topology in both dimensions. With the hardware support of workload sharing, PEGroup4 migrates the extra workloads to PEGroup2 and PEGroup3. And PEGroup2 migrates the Block2 workload partition to PEGroup1. That significantly balances the workload and improves the utilization.

Considerations in the workload sharing design are two-fold. (i) The input neurons should be sharable between the PEGroups; (ii) the output neuron accumulation should be performed inter-PEGroups. We discuss these issues and our strategies within two cases, in which the workload is shared between neighboring PEGroups in horizontal or in vertical, respectively. For the horizontal sharing case, an extra data port is set on the BlockNeuronBuffer to solve the issue (i), which enables the PEGroup to access input neurons from the neighboring PEGroup in horizontal. The issue (ii) is naturally solved by the hierarchical CSB-Engine design, as the PEGroup can store the partial results of the shared workload partition in its local NeuronAccumBuffer, which will be accumulated in horizontal after processing the entire block-row. For the vertical sharing case, the PEGroup-column shares the same BlockNeuronBuffer, thus the issue (i) is naturally solved by hardware. About the issue (ii), the PEGroup should accumulate the vertically shared workload to its original PEGroup, as the vertical PEGroups compute different block-rows that cannot be accumulated in a mixed manner. However, concurrent accumulation to one address in NeuronAccumBuffer leads to the read-after-write (RAW) data hazard. To address this issue, an accumulation path is set between vertical PEGroups and connected to the adder, which accepts parallel results from neighboring PEGroups, sums up and stores to the NeuronAccumBuffer for one shot. With the hardware support on workload sharing, we propose the compilation scheme in next section that schedules the partition and sharing by analyzing the CSB pruned matrix and generates the instruction to control the hardware-sharing behavior.

5 COMPILATION FOR CSB PRUNED MODEL

The proposed RNN dataflow architecture is controlled by the pre-compiled instructions. The instruction set includes the macro-instruction and micro-instruction, where the former one conducts the operation units (in Fig. 5) for the proper RNN dataflow (cell type); and the later one instructs the CSB-Engine with inter-PEGroup workload sharing behavior as described in §4.3.2. Correspondingly, the compilation is composed of two phases, RNN dataflow compilation (§5.1) and workload sharing scheduling (§5.2).

5.1 RNN Cell to Dataflow Architecture

5.1.1 Macro-Instruction Set. We define the macro-instruction set for our RNN dataflow architecture (§4.2). As Fig. 8, the micro-instruction is composed of multiple sections, that each section provides control signals for corresponding RNN primitive hardware. All sections are concatenated to form a very long instruction word (VLIW) item. Note that each section contains Count operand to indicate the size of workload for corresponding hardware primitive. Thus, one VLIW instruction is regarded as accomplished until all hardware primitives finish the workload. The operands in each instruction section are classified into two types, the Count type controls the hardware iteration count, and the other operands indicate the proper data source or destination for each primitive. For the first type, the value of Count in element-wise operation units (only CSB-Engine excluded) is measured by data element as these units perform element-wise operation. Differently, the Count/H/V in CSB-Engine section represents the horizontal/vertical block iteration counts over the entire CSB-Engine in processing the particular weight matrix. For the second operand type, Addr(Memory) and Addr(Buffer) give the address access of external memory (normally DRAM) and built-in buffers in the architecture, respectively. Importantly, the programmable datapaths in the architecture (Fig. 5) are indexed, and the DataFlowIdx is set in the operand to indicate the proper data source or destination for hardware primitive.
5.2 Workload Scheduling on CSB-Engine

5.2.1 Micro-Instruction Set. The micro-instructions are generated for each PEGroup individually, which control the CSB-MVM operations on CSB-Engine. Specifically, the micro-instruction contains the CSB-compression information (i.e., kernel matrix size, row- and column-index in Fig. 3(c)) for the block workload allocated to the certain PEGroup. In particular, the kernel matrix workload is partitioned to three submatrices and shared to neighboring PEGroups (as Fig. 9(a)), the micro-instructions for one block iteration include three items, (i) local workload that is originally allocated, excluding the portion shared to other PEGroups; (ii) workload shared from the neighboring PEGroup in horizontal; (iii) workload shared from the neighboring PEGroup in vertical. The micro-instruction contains 4 operands, as Fig. 9(b). The operand Sharing gives a flag (local/horizontal/vertical) to indicate the data source, where local means the input and output neurons are in local PEGroup; horizontal (sharing) indicates the input neurons should be read from the BlockNeuronBuffer of left PEGroup; vertical (sharing) means the output should be accumulated to the NeuronAccumBuffer of upper PEGroup. The operand TripCount gives the size of workload. Note that, for each block, the kernel matrix is divided to tree regular partitions as Fig. 9(a), for local (no-sharing), vertical- and horizontal-sharing, respectively. The sizes of partitioned matrices are denoted as $m' \times n'$, $\Delta m \times \Delta n_v$, and $\Delta m \times \Delta n_h$, which are turned to TripCount values in the three micro-instruction items. The operands RowIdx and ColIdx provide the non-zero row and column indices of each submatrix. Note that each micro-instruction item may contain multiple RowIdx and ColIdx corresponding to the TripCount value. Further, these two operands are stored in individual instruction memories that are read and reused periodically in the submatrix computation.

5.2.2 Micro-Instruction Compilation. The compilation of micro-instruction is essentially searching the workload partition scheme to achieve the optimal balance, which facilitates a higher hardware utilization (efficiency). Specifically, the compiler analyzes the weight...
matrices and selects the proper partition variables (as Fig. 9(a)) for each kernel matrix. Every $K \times L$ blocks (one block iteration) are analyzed individually, which are executed on PEGroups in concurrent. Within one block iteration, each PEGroup is supposed to take the equivalent workload after balancing.

We regard the search of optimal partition variable as a satisfiability modulo theories (SMT) problem [33], which searches the feasible solutions in the constrained region. The existing SMT solver [5] takes the constraints with logic programming and gives the satisfiability (existence of solution) and feasible solutions. In the compilation for each block iteration, we declare the integer variables including $m'(k, l)$, $n'(k, l)$, $\Delta m_k(k, l)$, $\Delta n_k(k, l)$, $\Delta m_v(k, l)$, $\Delta n_v(k, l)$, where $k \in [1, K]$ and $l \in [1, L]$. The constraints are represented with the constraint logic programming (CLP), in which each clause gives a specific search limitation. The CLP in compilation is listed in Eqn. 7, where $\land$ represents logic AND and $\lor$ represents OR. CLP1,2 constraint the feasible search region, as the size of the partitioned workload should $\leq$ kernel matrix size $(m \times n)$. CLP3,4 guarantee regular partitions as Fig. 9(a). CLP5 determines the values of $m'$ and $n'$. To improve the PEGroup utilization, we set CLP6 constraint that the size of partition workload should be integer-multiple of the PEGroup size. Thus, the PEs are fully utilized on the shared workload partition. Also, it helps to shrink the search space and speed up the compilation. Within the idealized situation, each PEGroup is scheduled with workload that is the average value over all PEGroups in the current block iteration. Otherwise, the PEGroup with maximum workload determines the run time (clock cycle) for this iteration. CLP7 gives the constraint on the maximum workload that, to all PEGroups, the exceeding part of scheduled workload to the average value (avg) should $\geq$ margin, which is given before search. The last CLP combines all above constraints to a conjunctive form, which is subsequently sent to SMT solver (a feasible solution).

\[
\begin{align*}
\text{CLP}_1 & : \{0 \leq \Delta m_k(k, l) \leq m(k, l)\} \land \{0 \leq \Delta n_k(k, l) \leq n(k, l)\} \\
\text{CLP}_2 & : \{0 \leq \Delta n_v(k, l) \leq \lfloor m(k, l)/2 \rfloor\} \land \{0 \leq \Delta n_v(k, l) \leq m(k, l)\} \\
\text{CLP}_3 & : \{\Delta m_k(k, l) = m(k, l)\} \land \{\Delta n_v(k, l) + \Delta n_k(k, l) = n(k, l)\} \\
\text{CLP}_4 & : \{\Delta n_v(k, l) = n(k, l)\} \land \{\Delta m_k(k, l) + \Delta n_v(k, l) = m(k, l)\} \\
\text{CLP}_5 & : \{(m'(k, l) = m(k, l) - \Delta m_v(k, l)) \land (n'(k, l) = n(k, l) - \Delta n_v(k, l))\} \\
\text{CLP}_6 & : \{\Delta m_k(k, l) \% P = m'(k, l) \% P = 0\} \land \{\Delta n_v(k, l) \% Q = n'(k, l) \% Q = 0\} \\
\text{CLP}_7 & : \{(m'(k, l) \times n'(k, l) + \Delta m_v(k, l) \times n'(k, l) - 1) \\
& + \Delta m_v(k-1, l) \times n'(k-1, l)) - \text{avg} \mid \leq \text{margin}\} \\
\text{CLP} & : \text{CLP}_1 \lor \text{CLP}_2 \lor (\text{CLP}_1 \lor \text{CLP}_4) \land \text{CLP}_3 \land \text{CLP}_5 \land \text{CLP}_6 \land \text{CLP}_7
\end{align*}
\]  

Based on the above formulation, we propose the compilation scheme in Algorithm 2 that seek out the optimal scheduling solution. For a given CSB formatted weight matrix $W_{csb}$, the compiler partitions it to $[W/M/L] \times [H/N/K]$ temporal block iterations and schedules each iteration individually. Before the multi-round search, the compiler firstly analyzes the weight partition for current block iteration that gives the kernel matrix size $(m, n)$ for each block and the average workload (avg). The margin is initialized to 0 that targets to schedule an idealized average workload on each PEGroup. In the search round, BuildCLP constructs the constraints representation, which is input to SMT Solver. In case the constraints cannot be satisfied (Satisfiability is False) over the feasible region, the margin value is supposed to increase by $P \times Q$ in the next round.

**Algorithm 2: Micro-Instruction Compilation**

**input**: $W_{csb}$ pruned weight matrix $W_{csb}$; 
block size in CSB $M \times N$; weight matrix size $W \times H$; size of each PEGroup $P \times Q$; PEGroup count $K \times L$ 
**output**: Micro-instruction list $\text{MicroInst}$ 

// Temporal block iterations in vertical. 
for $i \leftarrow 1$ to $[H/N/K]$ do 

// Temporal block iterations in horizontal. 
for $j \leftarrow 1$ to $[W/M/L]$ do 

$\text{margin} = 0$ 

// $\forall k \in [1, K]$, $\forall l \in [1, L]$. 

$m(k, l), n(k, l), \text{avg} = \text{Analyse}(W_{csb,k,l})$ 

// Search with multiple rounds. 
repeat 

$\text{CLP} = \text{BuildCLP}(m(k, l), n(k, l), \text{avg}, \text{margin})$ 

// Give solution if satisfied. 

[Satisfiability, PartitionVar] = $\text{SMTSolver}(\text{CLP})$ 

$\text{margin} = P \times Q$ 

until Satisfiability = True; 

$\text{MicroInst} = \text{Append}(\text{MicroInst}, \text{PartitionVar})$

Once the SMT problem is satisfied, the search stops and the partition variables ($m', n'$, $\Delta m_v$, $\Delta n_v$, $\Delta m_k$, $\Delta n_k$) for each PEGroup are assembled and appended to the micro-instruction list, that conducts the CSB-Engine computation in a workload balanced fashion.

### 6 EVALUATION

In this section, we first brief the implementation of the CSB-RNN framework ($§6.1$), and then give deep evaluations from the performance of CSB pruning algorithm ($§6.2$) to the improvement with the architecture-compilation co-design ($§6.3$). Meanwhile, 10 mainstream RNN models from multi-domains are invoked as the evaluation benchmarks and presented in Table 1, in which we also list the non-structured pruning rates as the theoretical optimum.

#### 6.1 Implementation and Experiments Setup

The CSB pruning flow was implemented with PyTorch [23], a framework for deep learning model development. The benchmark models were first trained with the SGD and the accuracy is regarded as the lossless target value in the subsequent CSB pruning. These baseline models were fed in the CSB pruning flow and get compressed with the lossless constraints. In regarding the architecture-compilation co-design, the proposed RNN dataflow architecture was realized with Verilog RTL and implemented on an FPGA vendor evaluation board (Xilinx-ZCU102), on which the FPGA contains enough resources for our architecture with different design scales. The compiler was implemented in C++ with the strategies in §5 and Z3 [5] as the SMT solver. With the CSB pruned model, the compiler dumps the macro-instructions ($§5.1$) to build the proper RNN dataflow and micro-instructions ($§5.2$) for the workload balancing. These instructions are loaded to the RNN dataflow architecture before processing sequence continuously. With regard to the detailed hardware efficiency (i.e. CSB-Engine utilization), cycle-level RTL simulation was performed to profile the inference behavior.
Table 1: Benchmark Models in CSB-RNN Evaluation

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>MT1</td>
<td>Machine Translation</td>
<td>PTB[20]</td>
<td>2</td>
<td>LSTM[12]</td>
<td>128</td>
<td>256</td>
<td>Perplexity (PPL, lower is better)</td>
<td>393K×1K 1K</td>
<td>13.2× 29.8K 111.62</td>
</tr>
<tr>
<td>2</td>
<td>MT2</td>
<td>Machine Translation</td>
<td>PTB[20]</td>
<td>2</td>
<td>LSTM[12]</td>
<td>1500</td>
<td>1500</td>
<td>Perplexity (PPL)</td>
<td>18M×6K 1M</td>
<td>16.5× 1.1M 82.33</td>
</tr>
<tr>
<td>3</td>
<td>SR1</td>
<td>Speech Recognition</td>
<td>TIMIT[8]</td>
<td>2</td>
<td>LSTM[25]</td>
<td>512</td>
<td>1024</td>
<td>Phoneme Error Rate (PER, lower is better)</td>
<td>3.25M×4K 4.72M×4K</td>
<td>14.5× 325.4K 19.70%</td>
</tr>
<tr>
<td>5</td>
<td>SR3</td>
<td>Speech Recognition</td>
<td>TIMIT[8]</td>
<td>2</td>
<td>Li-GRU[24]</td>
<td>39</td>
<td>1024</td>
<td>PER</td>
<td>564.2K 1M</td>
<td>7.1× 147.2K 17.30%</td>
</tr>
<tr>
<td>6</td>
<td>SR4</td>
<td>Speech Recognition</td>
<td>TIRGFT[15]</td>
<td>2</td>
<td>Li-GRU[24]</td>
<td>39</td>
<td>1024</td>
<td>Accuracy</td>
<td>22.6K×0.8K 99.98%</td>
<td>25.7× 8.8K 99.21%</td>
</tr>
<tr>
<td>7</td>
<td>SPP</td>
<td>Stock Price Prediction</td>
<td>S&amp;P500[13]</td>
<td>2</td>
<td>LSTM[12]</td>
<td>128</td>
<td>128</td>
<td>Normalized Price Dist (lower is better)</td>
<td>66K×0.5K 13K×0.5K</td>
<td>4.1× 16.1K 0.51</td>
</tr>
<tr>
<td>8</td>
<td>SC1</td>
<td>Sentiment Classification</td>
<td>IMDB[18]</td>
<td>3</td>
<td>LSTM[12]</td>
<td>128</td>
<td>128</td>
<td>Accuracy (higher is better)</td>
<td>1.1M×3K 2.1M×2K</td>
<td>10.4× 201.6K 85.65%</td>
</tr>
<tr>
<td>9</td>
<td>SC2</td>
<td>Sentiment Classification</td>
<td>IMDB[18]</td>
<td>3</td>
<td>LSTM[12]</td>
<td>512</td>
<td>512</td>
<td>Accuracy (higher is better)</td>
<td>333K×1K 107.1K</td>
<td>7.2× 43.5K 76.31%</td>
</tr>
<tr>
<td>10</td>
<td>QA</td>
<td>Question Answering</td>
<td>BABI[32]</td>
<td>3</td>
<td>LSTM[12]</td>
<td>256</td>
<td>256</td>
<td>Accuracy</td>
<td>513.3K×1K 65.37%</td>
<td>7.9× 66.4K 64.51%</td>
</tr>
</tbody>
</table>

Figure 10: (a) shows the pruning rate comparison between non-structured pruning (optimum) and CSB pruning in different block sizes. (b) shows the normalized index overhead (NIO). Comparing (a) and (b), we gain the insight that CSB pruning dramatically reduces the NIO while maintaining a high pruning rate.

### 6.2 Evaluation of CSB pruning Rate

The CSB pruning is first evaluated in the aspect of pruning rate, which is a significant metric to score the model compression methods. Because the parameterizable block size determines the structural granularity in pruning, we present the attainable maximum pruning rate with various block sizes in §6.2.1. Further, comparison with the prior art RNN compression schemes is given in §6.2.2.

#### 6.2.1 Selection of Optimum Structural Granularity

The CSB pruning provides the flexibility that improves the pruning rate and also the hardware-friendly regularity. Importantly, a trade-off exists between these two targets that motivate the following investigation. Reducing the block size facilitates a more fine-grained pruning and thus a higher pruning rate. However, more individual blocks require extra storage for row and column index with the CSB-formatted weight matrix (Fig. 3). Therefore, we present both the attainable pruning rate and the index overhead with different block sizes in each benchmark model. The block is set to square with sizes of 16, 32, 64, 128, considering the weight matrix dimensions in different models. Note that for matrix with very small size (e.g., 256 × 39 in SR4), the short dimension (39) is partitioned to Q blocks uniformly after padding a zero-column. Multiple layers in one model adopt the same pruning rate. The attainable pruning rate for each case is presented in Fig. 10(a). Further, the index overheads are divided by the corresponding weight count for normalization, and the values of the normalized index overhead (NIO) are presented in Fig. 10(b). Notably, the results with non-structured pruning are given for comparison (leftmost bar for each application). And its index overhead is obtained by compressing the non-structured weight matrices with the compressed sparse row (CSR) format.

As a result, the CSB pruning rate ranges from 3.5× to 25×, which dramatically reduces the original model size by order of magnitude. With the growth of block size, the pruning rate decreases as the coarse-granularity block reduces the pruning flexibility. We note that, in all benchmarks, the CSB pruning is capable of reaching a maximum pruning rate with the block size of 16 or 32, which is close to non-structured pruning. In the aspect of NIO, the index overhead of non-structured pruning exceeds 100%, as at least one index is required for a non-zero element. Nevertheless, for CSB pruning, the NIO is below 50% in most cases due to index reusability in the structured blocks. The NIO shows a significant decay while enlarging the block size. With the block size of 32, the NIO declines to ~ 20%, which is 1/5 of that in non-structured pruning. Interestingly, we gain the insight that with a block size of 32 and...
16, most models achieve the close pruning rate. For instance, 13× and 12× in MT1; both are 20× in SR2. Therefore, the larger block size (32) is preferable for its low index overhead.

### 6.2.2 Comparison with Prior Art Compression Schemes

The CSB pruning rate is further compared to the prior art RNN compression techniques in Table 2. The listed competitive techniques are proposed to enable a faster, hardware-friendly RNN inference with the compressed model. Note that these competitors quantized the weight to 16-bit fixed-point numbers; Thus, we do the same quantization on CSB pruned model and report the corresponding results for a fair comparison. In Table 2, row-column [30] technique prunes each weight matrix as an entire block. Comparing to it, our fine-grained CSB pruning improves the compression rate to 3.9×. The row balanced [10] or bank balanced [3] techniques compulsively train the model to a balanced sparse pattern; However, CSB pruning remains the natural unbalanced sparsity in RNN model and achieves a higher (1.6×) pruning rate. Overall, the CSB pruning improves the pruning rate to 1.6×-3.9× of the existing schemes, while maintaining an even better model accuracy.

### 6.3 Evaluation of RNN Dataflow Architecture with CSB Pruned Model

#### 6.3.1 Hardware-resource Consumption

The hardware-resource consumption (cost) of the RNN dataflow architecture is given in Fig. 11, with various CSB-Engine configs ($P,Q,K,L$ and max supported block size). Notably, the CSB-Engine with different workload sharing configs, including no-sharing, vertical-sharing, horizontal-sharing, 2D-sharing, are synthesized individually to evaluate the hardware overhead on workload sharing technique. The consumption of hardware logic and memory from the FPGA vendor tool are presented in Fig. 11. The configurable logic block (CLB, left axis) is the FPGA building block for logic, which is used as the logic resource metric; The memory resource is given in megabit (Mb in the right axis). Note that most memory resource on our FPGA device is configured as the weight buffer, although they may not be fully used by small RNN models. The multiplier in each PE (16-bit fixed-point) is mapped to digital signal processor (DSP) on FPGA, and the DSP count in design is $P \times Q \times K \times L$ that is omitted here. As a result, the hardware support of workload sharing costs an acceptable overhead, which is 11.6%, 3.8%, and 15.6% for three sharing cases (vertical/horizontal/2D-sharing), respectively.

#### 6.3.2 Performance

Due to the workload imbalance issue, the processing performance of RNN dataflow architecture, CSB-Engine in specific, is not deterministic. Hardware efficiency, the ratio of effective computation on PEs, is invoked to evaluate the improvement of our workload sharing technique. We obtained the CSB-Engine efficiency by measuring the PE pipeline utilization using 10 benchmarks listed in Table 1 with different design choices of workload sharing. Moreover, CSB pruned models with different block sizes are used to evaluate the impact of block size on efficiency. The efficiency is measured layer-by-layer on hardware with 4×4 PEGroups and each contains 4×4 PEs. The results are presented in Fig. 12. Overall, for the CSB-Engine without workload sharing, the efficiency is 42% on average, which results from the imbalanced workload (sparsity) of blocks. The single dimensional sharing (vertical or horizontal) improves the efficiency to an average of 72%. After the 2D-sharing is adopted, the efficiency is further improved to 94% on average, i.e., only 6% execution time of CSB-Engine is invalid. This 6% pipeline gap is inevitable, as a few extremely imbalanced sparsity exists in some weight matrices. For instance, we found diagonal dense matrix exists that the blocks on the matrix diagonal contain significant workload compared to other blocks. In this case, the workload sharing path in the current design is not enough, while adding more sharing paths brings extra hardware costs.

Comparing the efficiency within the same layer but different pruning block sizes, it is apparent that the smaller block size is applied, the lower hardware efficiency CSB-Engine can achieve, particularly in the no-sharing CSB-Engine cases. This is because the small block includes less workload (with the same pruning rate) but more temporal block iterations, which lead to PE idle more easily. As mentioned in §6.2.1, using smaller block sizes in compression guarantees higher model pruning rates, which benefits are significantly encroached by the performance degradation with small compression block in the no-sharing cases. Nevertheless, we gain the insight that our architecture-compilation co-design for 2D-sharing cases significantly subdues the degradation. For instance, in Layer-2 (L2) of MT1 case, the no-sharing degradation from block-64 to block-32 is 12%, while it is reduced to 3% by the 2D-sharing. On average, the degradation is reduced from 15% to 4%. In summary, with the proposed workload sharing technique, a smaller block size in CSB pruning does not bring significant degradation on hardware efficiency anymore (only 4% on average), so that the benefits from higher pruning rates can be more sufficiently exploited.
## Table 3: Latency and Power Efficiency Comparison

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>MT1</td>
<td>BBS [3]</td>
<td>1518</td>
<td>200</td>
<td>1.30</td>
<td>19</td>
<td>40.49</td>
<td>1x</td>
</tr>
<tr>
<td></td>
<td>CSB-RNN</td>
<td>512</td>
<td>200</td>
<td>0.79</td>
<td>8.9</td>
<td>142.72</td>
<td>3.53x</td>
</tr>
<tr>
<td>C-LSTM [28]</td>
<td></td>
<td>2680</td>
<td>200</td>
<td>8.10</td>
<td>22</td>
<td>5.61</td>
<td>19.35x</td>
</tr>
<tr>
<td>E-RNN [16]</td>
<td></td>
<td>2660</td>
<td>200</td>
<td>7.40</td>
<td>24</td>
<td>5.63</td>
<td>19.41x</td>
</tr>
<tr>
<td>ESE [10]</td>
<td></td>
<td>1504</td>
<td>200</td>
<td>82.70</td>
<td>41</td>
<td>0.29</td>
<td>1x</td>
</tr>
<tr>
<td>CSB-RNN</td>
<td></td>
<td>512</td>
<td>200</td>
<td>6.58</td>
<td>8.9</td>
<td>17.08</td>
<td>58.89x</td>
</tr>
<tr>
<td>SR2</td>
<td>E-RNN [16]</td>
<td>2230</td>
<td>200</td>
<td>6.70</td>
<td>29</td>
<td>5.15</td>
<td>1x</td>
</tr>
<tr>
<td>CSB-RNN</td>
<td></td>
<td>512</td>
<td>200</td>
<td>5.18</td>
<td>8.9</td>
<td>21.69</td>
<td>4.21x</td>
</tr>
</tbody>
</table>

### 6.3.3 Comparison with Related Works

The overall performance of CSB-RNN, i.e., CSB pruned model inference speed on the proposed RNN dataflow architecture, is listed in Table 3 and compared with the prior art designs. We collected the statistics including the PE count (#PE), operating frequency, latency in processing one input frame and the power of design. As Table 3 shows, with the same benchmark applications, the CSB-RNN reduces the latency by 39%-92% that speeds up the processing from 1.12x to 12.57x correspondingly; Nevertheless, CSB-RNN only uses 1%-34% PE counts (hardware resource) of the competitors to attain this performance. The latency ranges from 0.79µs to 6.58µs with different model sizes. For generic high-precision speech recognition, at most ≈ 2000 frames should be processed per second, which requires a latency ≤ 500µs to meet the realtime performance. As the achieved latency with benchmark models is much lower than this requirement, the CSB-RNN provides a faster-than-realtime performance and facilitates the device processing more complex RNN models in the future. Besides the latency, we compare the power efficiency (k-frames per Watt) among these competitive designs. The results show the CSB-RNN achieves significant improvements from 3.53x to 58.89x on power efficiency in processing the same model, which makes the CSB-RNN quite suitable for embedded scenarios. Further, while the existing works were designed for a particular RNN cell type, CSB-RNN can be reprogrammed to adapt to different cells.

### 7 CONCLUSION

This paper presents CSB-RNN, an optimized full-stack RNN acceleration framework. The fine-grained structured CSB pruning significantly improves the pruning rate compared to existing hardware-friendly pruning schemes. Meanwhile, an architecture-compilation co-design is proposed that sufficiently exploits the benefits of the CSB pruned model. The experiments show that the entire CSB-RNN acceleration framework delivers a faster-than-realtime performance on extensive RNN models, and dramatically reduces the latency and improves the power efficiency compared with the existing works.

**Future work:** We are extending the CSB technique to other neural network layers. In particular, the transformer models are composed of more complex dataflow, however, the same MVM primitive as RNN. With improvement on the dataflow abstraction, the proposed CSB pruning and CSB-Engine will contribute to the realtime transformer inference.

**ACKNOWLEDGMENTS**

We would like to thank the anonymous reviewers for their valuable comments. This research was supported in part by the Croucher Foundation (Croucher Innovation Award 2013), the Research Grants Council of Hong Kong grant number CRF C7047-16G, GRF 1745716. This research was supported in part by the U.S. DOE Office of Science, Office of Advanced Scientific Computing Research, under award 66150: "CENATE - Center for Advanced Architecture Evaluation". This research was supported, in part, by the NSF through awards CCF-1618303, CCF-1919130, CCF-1937500, CNS-1909172, and CCF-1919117; the NIH through awards R41GM128533 and R44GM128533; and by a grant from Red Hat.

**REFERENCES**


