TOWARDS EFFICIENT HARDWARE ACCELERATION OF DEEP NEURAL NETWORKS ON FPGA

by

Sicheng Li

M.S. in Electrical Engineering,

New York University, 2013

B.S. in Electrical Engineering,

Beijing University of Posts and Communications, 2011

Submitted to the Graduate Faculty of

the Swanson School of Engineering in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

University of Pittsburgh

2017

UNIVERSITY OF PITTSBURGH

SWANSON SCHOOL OF ENGINEERING

This dissertation was presented

by

Sicheng Li

It was defended on

September 29 2017

and approved by

Hai (Helen) Li, Ph.D., Adjunct Associate Professor, Department of Electrical and Computer Engineering

Yiran Chen, Ph.D., Adjunct Associate Professor, Department of Electrical and Computer Engineering

Zhi-Hong Mao, Ph.D., Associate Professor, Department of Electrical and Computer Engineering

Ervin Sejdic, Ph.D., Associate Professor, Department of Electrical and Computer Engineering

Yu Wang, Ph.D., Associate Professor, Department of Electrical Engineering, Tsinghua University

Dissertation Director: Hai (Helen) Li, Ph.D., Adjunct Associate Professor, Department of Electrical and Computer Engineering

TOWARDS EFFICIENT HARDWARE ACCELERATION OF DEEP NEURAL NETWORKS ON FPGA

Sicheng Li, PhD

University of Pittsburgh, 2017

Deep neural network (DNN) has achieved remarkable success in many applications because of its powerful capability for data processing. Their performance in computer vision have matched and in some areas even surpassed human capabilities. Deep neural networks can capture complex non-linear features; however this ability comes at the cost of high computational and memory requirements. State-of-art networks require billions of arithmetic operations and millions of parameters. The brute-force computing model of DNN often requires extremely large hardware resources, introducing severe concerns on its scalability running on traditional von Neumann architecture. The well-known memory wall, and latency brought by the long-range connectivity and communication of DNN severely constrain the computation efficiency of DNN. The acceleration techniques of DNN, either software or hardware, often suffer from poor hardware execution efficiency of the simplified model (software), or inevitable accuracy degradation and limited supportable algorithms (hardware), respectively. In order to preserve the inference accuracy and make the hardware implementation in a more efficient form, a close investigation to the hardware/software co-design methodologies for DNNs is needed.

The proposed work first presents an FPGA-based implementation framework for Recurrent Neural Network (RNN) acceleration. At architectural level, we improve the parallelism of RNN training scheme and reduce the computing resource requirement for computation efficiency enhancement. The hardware implementation primarily targets at reducing data communication load. Secondly, we propose a data locality-aware sparse matrix and vector multiplication (SpMV) kernel. At software level, we reorganize a large sparse matrix into many modest-sized blocks by adopting hypergraph-based partitioning and clustering. Available hardware constraints have been taken into consideration for the memory allocation and data access regularization. Thirdly, we present a holistic acceleration to sparse convolutional neural network (CNN). During network training, the data locality is regularized to ease the hardware mapping. The distributed architecture enables high computation parallelism and data reuse. The proposed research results in an hardware/software co-design methodology for fast and accurate DNN acceleration, through the innovations in algorithm optimization, hardware implementation, and the interactive design process across these two domains.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1.0 INTRODUCTION

Following technology advances in high performance computation systems and fast growth of data acquisition, machine learning, especially deep learning, made remarkable success in many research areas and applications, such as image recognition [4], object detection [5], natural language processing [6] and automatic speech recognition [7]. Such a success, to a great extent, is enabled by developing large-scale deep neural networks (DNN) that learn from a huge volume of data. For example, in *Large Scale Visual Recognition Challenge 2012*, Krizhevsky et al. beat out the second-best team 10% in Imagenet classification accuracy by training a deep convolutional neural network with 60 million parameters and 650,000 neurons on 1.2 million images. The deployment of such a big model is both computation-intensive and memory-intensive. At software level, extending the depth of neural networks for accuracy optimization becomes a popular approach [8][9], exacerbating the demand for computation resources and data storage of hardware platforms.

The research on hardware acceleration for neural network has been extensively studied on not only the general-purpose platforms, e.g., graphic processing units (GPUs), but also domain-specific hardware such as field-programmable gate arrays (FPGAs) and custom chip (e.g., TrueNorth) [10][11][12][13][14]. High-end GPUs enable fast deep learning, thanks to their large throughput and memory capacity. When training AlexNet with Berkeley's deep learning framework Caffe ([10]) and Nvidia's cuDNN ([15]), a Tesla K-40 GPU can process an image in just 4ms. While GPUs are an excellent accelerator for deep learning in the cloud, mobile systems are much more sensitive to energy consumption. In order to deploy deep learning algorithm in energy-constraint mobile systems, various approaches have been offered to reduce the computational and memory requirements of deep neural networks).

1.1 DEEP NEURAL NETWORKS ON FPGAS

Field Programmable Gate Arrays (FPGAs) have shown massive speedup potential for a wide range of applications. Their ability to support highly parallel designs, coupled with their re-programmability have made them very attractive platforms. Custom pipelined datapaths allow the FPGA to execute in parallel what could take thousands of operations in software. Programmability, and ease of use deter many software developers from expanding into hardware development. FPGAs are notorious for complex designs, long debug cycles, and difficult verification among other things. However, some of these issues can be alleviated by advances in hardware design tools. Major FPGA manufactures are actively developing High Level Synthesis (HLS) tools to help software software developers utilize their boards. Besides, FPGAs now have easy access to significantly larger memory spaces, which allows researchers to consider much larger real-world problems. However, the larger memories come at a cost of higher latencies.

Various FPGA-based DNN accelerators ([16][17]) have proven that it is possible to use reconfigurable hardware for end-to-end inference of large CNNs like AlexNet and VGG. An important problem faced by designers of FPGA-based DNNs is to select the appropriate DNN model for a specific problem to be implemented using optimal hardware resources. Moreover, the progress of hardware development still falls far behind the upscaling of DNN models at software level. With the high demand for computation resources, memory wall [14] that describing the disparity between fast on-chip data processing rate and slow off-chip memory and disk drive I/O demonstrates more prominent adverse impact [18][19].

1.2 DISSERTATION CONTRIBUTION AND OUTLINE

To overcome the above challenges in neural network acceleration, both hardware and software solutions are investigated. The software approaches, in contrast, mainly concentrate on reducing the scale and connections of a DNN model while still keeping the state-of-the-art accuracy. The hardware approaches attempt to build a specialized architecture for the customized network models adapting to it.

Our proposed work can be decoupled as following two main research scopes: 1) FPGA acceleration of recurrent neural network based language model; 2) A data locality-aware design framework for reconfigurable sparse matrix-vector multiplication kernel. 3) The software-hardware co-design on sparse convolutional neural networks.

For research scope 1, we proposed an FPGA-based acceleration for recurrent neural networks, which includes three major technical contributions:

- At architectural level, the framework extends the inherent parallelism of RNN and adopts a mixed-precision scheme. The approach enhances the utilization of configurable logics and improves computation efficiency.
- The hardware implementation integrates a groups of computation engines and a multi-thread management unit. The structure successfully conceals the irregular memory access feature in data back-propagation stage and reduces external memory accesses. Our framework is designed in a scalable manner to benefit the future investigation for ever-larger networks.

We realized the RNNLM on Convey HC-2^{ex} system. The design was trained with a dataset of 38M words. It consists of $1,024$ nodes in hidden layer. Our design performs better than traditional class-based modest-size recurrent networks and obtains 46.2% in accuracy in *Microsoft Research Sentence Completion* (MRSC) challenge. The experiments at different network sizes on average achieve $14.1\times$ speedup over the optimized CPU implementation and a comparable performance with high-end GPU system, demonstrating a great system scalability.

For research scope 2, we developed an efficient SpMV computation kernel for sparse neural networks. The main contributions of this work are:

- By analyzing the impact of the sparse structure of matrix and various hardware parameters on system performance and accordingly propose a data locality-ware co-design framework for iterative SpMV.
- We integrate conventional sparse matrix compression formats with a locality-aware clustering technique. A sparse matrix will be reorganized into sub-blocks, each of which has regularized memory accesses. At hardware level, we develop a scalable architecture made of high-parallel processing elements (PEs) that enable simultaneous MACs and customized data path for inter-PE communications.

The experiments based on the University of Florida sparse matrix collection shows dramatic improvement in computational efficiency. Our FPGA-based implementation has a comparable runtime as GPU and achieves 2.3× reduction than CPU, with substantial saving in power consumption, say, $8.9\times$ and $8.3\times$ better than the implementations on CPU and GPU, respectively.

To enable the CNN sparsification and acceleration on FPGA, for both convolutional and fully connected layers within CNNs, we propose a co-design framework by combining innovations in software and hardware domains. More specific, the main contributions of this work include:

- We profile the impact of sparse network structures and hardware parameters on overall system performance and demonstrate that the software/hardware co-design is necessary to accelerate sparse CNNs.
- At software level, we focus on the data locality enhancement during model sparsification. Alone with a low-cost compression scheme, kernel weights are partitioned into sub-blocks with regularized data layout. At hardware level, a scalable architecture composed of processing elements (PEs) that simultaneously execute compressed kernel weights is developed. Zero-skipping and extensive data reuse scheme are applied to improve the operation efficiency of sparse feature map.
- As the sparse regularization affects the connections over layers, we introduce a sparsi
cation strategy which can adapt the design optimization according to the available hardware resource.

We evaluate the proposed design framework through three representative CNNs on two Xilinx FPGA platforms - ZC706 and VC707 boards. Our design can significantly improve the computation efficiency and effiective memory bandwidth, achieving an average 67.9% of the peak performance. This result is $1.8\times$ and $4.7\times$ higher than that of the implementations on high-end CPUs and GPUs, respectively. Very importantly, our design effectively reduces the classification time $2.6\times$, compared to state-of-the-art FPGA implementation.

The outline of this dissertation is organized as follows: Chapter 1 presents the overall picture of this dissertation, including the research motivations, research scopes and the research contributions; The details and applications of our acceleration framework for recurrent neural networks are illustrated in Chapter 2. Then, the sparse matrix-vector computation kernel will be presented in Chapter 3. Chapter 4 demonstrates the benefits of our proposed software-hardware co-design framework, and its evaluation result on sparse convolutional neural networks. Chapter 6 finally summarizes the research work and presents the potential future research directions, as well as our insights for efficient acceleration of deep neural networks on FPGA.

2.0 FPGA ACCELERATION TO RNN-BASED LANGUAGE MODEL

In this chapter, we will present the details of our hardware acceleration framework for recurrent neural network-based language model. The structure of this chapter is organized as the follows: Section 2.1 introduces the language model and RNN algorithm; Section 2.2 presents our analytical approach for accelerator design optimization; Section 2.3 and 2.4 explain our proposed architecture and the corresponding hardware implementation, respectively; Experimental results and analysis are shown in Section 2.5.

2.1 PRELIMINARY

2.1.1 Language Models

Rather than checking linguistic semantics, modern language models based on statistical analysis assign a probability to a sequence of words by means of a probability distribution. Ideally, a meaningful word sequence expect to have a larger probability than an incorrect one, such as $P(\underline{I}\ saw\ a\ dog) > P(\underline{Eye}\ saw\ a\ dog)$.

Among developed language models, *n-gram model* is the most commonly used. In an n-gram model, the probability of observing the i^{th} word w_i in the context history of the preceding $i-1$ words can be approximated by the probability of observing it in the shortened context history of the preceding $n-1$ words. For example, in a *2-gram* (also called as *bigram*) model, the probability of "$I\ saw\ a\ dog$" can be approximated as:

$$\begin{aligned} P(I\ saw\ a\ dog) =& P(I|-) \times P(saw|I) \times P(a|saw) \\ & \times P(dog|a) \times P(-|dog) \end{aligned}. \tag{2.1}$$

A conditional probability, *e.g.*, $P(a|saw)$ in Eq. (2.1), can be obtained through statistical analysis based on training data. The number of conditional probabilities required in n-gram increases exponentially as n grows: for a vocabulary with a size of V, an n-gram model need store V^n parameters. Moreover, the space of training data becomes highly sparse as n increases. In other words, a lot of meaningful word sequences will be missed in the training data set and hence statistical analysis cannot provide the corresponding conditional probabilities. Previous experiments showed that the performance of n-gram language models with a larger n ($n > 5$) is less effective [20]. N-gram model can realize only the short-term perspective of a sequence, which is clearly insufficient to capture semantics of sentences [21].

2.1.2 RNN & RNN based Language Model

Figure 1 illustrates the structure of a standard *recurrent neural network* (RNN). Unlike feedforward neural networks where all the layers are connected in a uniform direction, a RNN creates additional recurrent connections to internal states (*hidden layer*) to exhibit historical information. At time t, the relationship of input $\vec{x}(t)$, the temporary state of hidden layer $\vec{h}(t)$, and output $\vec{y}(t)$ can be described as

$$\vec{h}(t) = f\left(W_{ih}\vec{x}(t) + W_{hh}\vec{h}(t-1) + \vec{b}_h\right), \text{ and} \tag{2.2}$$

$$\vec{y}(t) = g\left(W_{ho}\vec{h}(t) + \vec{b}_o\right). \tag{2.3}$$

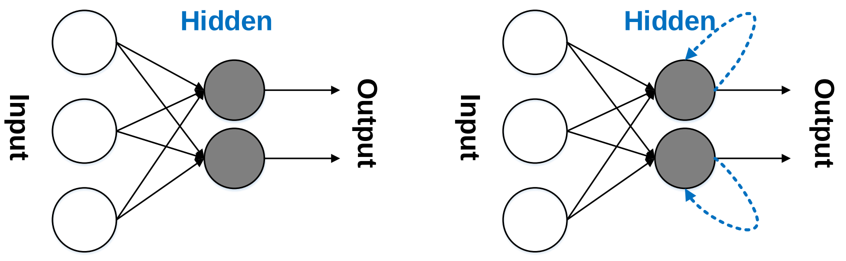


Figure 1: (a) Feedforward neural network; (b) Recurrent neural network.

Where, W_{ih} is the weight matrix connecting the input and hidden layers, W_{ho} is the one between the hidden and output layers. W_{hh} denotes the recurrent weight matrix between the hidden states at two consecutive time steps, *e.g.*, $\vec{h}(t-1)$ and $\vec{h}(t)$. $\vec{b}_h$ and $\vec{b}_o$ are the biases of the hidden and output layers, respectively. $f(z)$ and $g(z)$ denote the activation functions at the hidden and output layers, respectively.

The input/output layer of *RNN-based language model* (RNNLM) corresponds to the full or compressed vocabulary. So each node represents one or a set of words. In calculating the probability of a sentence, the words will be input in sequence. For instance, $\vec{x}(t)$ denotes the word at time t. And output $\vec{y}(t)$ represents the probability distribution of the next word, based on $\vec{x}(t)$ and the historical information stored as the previous state of network $\vec{h}(t-1)$.

RNNLM uses internal states at hidden layer to store the historical information, which is not constrained by the length of input history. Compared with n-gram models, RNNLM is able to realize a long-term perspective of the sequence. Note that the hidden layer usually has much less nodes than the input/output layer and its size shall reflect the amount of training data: the more training data are collected, the larger hidden layer is required. Moreover, the aforementioned sparsity of the training data in n-gram language model is not an issue in RNNLM, indicating that RNNLM has a stronger learning ability [22].

2.1.3 The RNN Training

When training a network of RNNLM, all data from training corpus are presented sequentially. In this work, we used *back-propagation through time* (BPTT) algorithm. As illustrated in Figure 2, the approach truncates the infinite recursion of a RNN and expands it to a finite feed-forward structure, which then can be trained by following the regular routine of feed-forward networks.

For a given input data, the actual output of network shall first be calculated. Then the weights of each matrix will be updated through back-propagating the deviations between the actual and desired outputs layer by layer. The update of weight w_{ji} between node i of the current layer and node j of the next layer at time t can be expressed as $w_{ji} \leftarrow w_{ji} + \eta \cdot \sum_{t=1}^{T} \delta_j(t) \cdot x_i(t)$,where $x_i(t)$ is the input of node i; η is the learning rate; $\delta_j(t)$ is the error back-propagated from node j; and T is the BPTT step for RNN training.

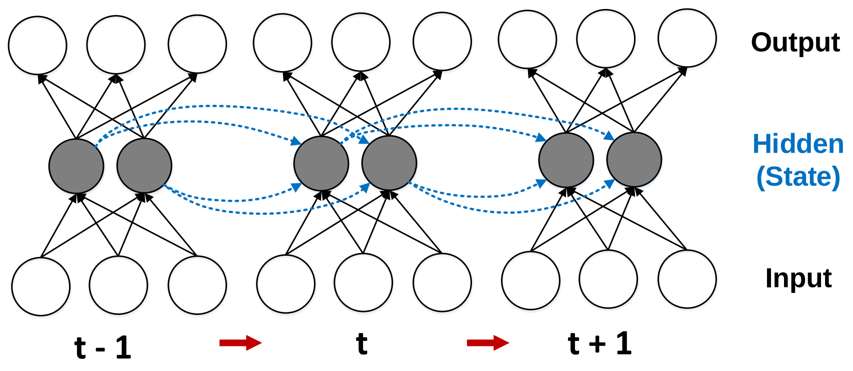


Figure 2: Unfold RNN for training through BPTT algorithm.

At the output layer, we adopted *softmax* activation function $g(z) = \frac{e^z}{\sum_k e^{z_k}}$ as the cross-entropy loss function. The error derivative of node p $\delta_p(t)$ can be obtained simply from RNN's actual output $o_p(t)$ and the desired one $t_p(t)$:

$$\delta_p(t) = t_p(t) - o_p(t). \tag{2.4}$$

Sigmoid function $f(z) = \frac{1}{1+e^{-z}}$ is utilized at the hidden layer. The error derivative of node k $\delta_k(t)$ is calculated by

$$\delta_k(t) = f'(x)|_{f(x)=h_k(t)} \cdot \delta_{\text{BPTT}}(t). \tag{2.5}$$

Where, $h_k(t)$ is the state of node k in hidden layer at time t. $\delta_{\text{BPTT}}$is the accumulation of the errors back-propagated through time, that is,

$$\delta_{\text{BPTT}}(t) = \sum_{o \in output} w_{ok}\delta_o(t) + \sum_{h \in hidden} w_{hk}\delta_h(t+1). \tag{2.6}$$

Here, $\delta_o(t)$ denotes the error of output layer at time t, while $\delta_h(t+1)$ is the error of hidden layer back-propagated from the following time step $t+1$. w_{ok} and w_{hk} are the transposed weights of W_{ho} in Eq. (3) and W_{hh} in Eq. (4), respectively.

2.2 ANALYSIS FOR DESIGN OPTIMIZATION

We first analyze the utilization of computation and communication resources in RNNLM as these are two principal constraints in system performance optimization.

Computation resource utilization. To analyze the computation cost, we implemented RNNLM on a CUBLAS-based NVIDIA GPU and profiled the runtime of every major function. The result in Table 1 shows that the matrix-vector multiplication consumes most of computation resource. The activation functions, as the second contributor, consume more than 20% of runtime. So we mainly focus on enhancing the computation efficiency of these two functions.

Memory accesses. During training, the matrix-vector multiplication in the back-propagation phase requires the transposed form of weight matrices as shown in Eq. (8). Such a data access exhibits irregular behavior, making the further performance improvement very difficult. To explore this effect experimentally, we mapped RNNLM on a multi-core server with Intel's *Math Kernel Library* (MKL). Figure 3 shows the normalized system performance. As more cores are utilized, the major constrain changes from computation resource to memory bandwidth. Accordingly, the speedup becomes slower and eventually saturated when the memory bandwidth is completed consumed.

Scalability. A scalable implementation must well balance the use of computation units and memory bandwidth. As the configurable logic elements on FPGA grow fast, the implementation shall be able to integrate additional resources. Our approach is to partition a design into multiple identical groups and migrate the optimized development of a group to bigger and ore devices for applications in larger scale.

Table 1: RNN Computation Runtime Breakdown in GPU

Matrix-vector Multi.	**Activation Functions**	**Sum of Vector Elem.**	**Vector Scaling**	**Delta**	**Others**
71.0%	21.4%	2.3%	1.9%	1.0%	2.4%

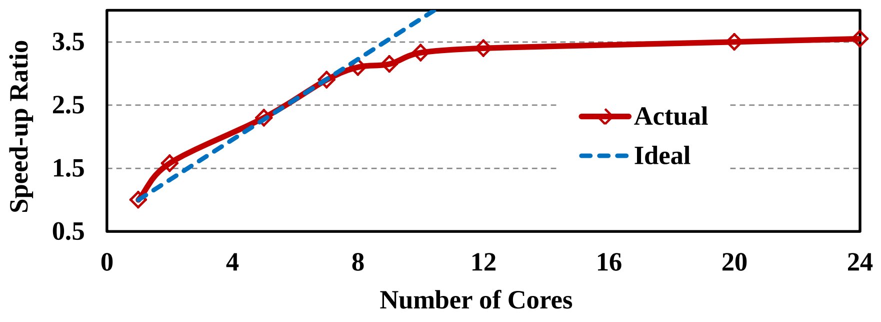


Figure 3: The system speedup is saturated with the increment of CPU cores as the memory accesses become the new bottleneck.

2.3 ARCHITECTURE OPTIMIZATION

This section describes the optimization details at the architectural level. We propose a parallel architecture to improve the execution speed between the hidden and output layers. Moreover, the computation efficiency is enhanced by trading off data and function precision.

2.3.1 Increase Parallelism between Hidden and Output Layers

Previously, Li *et al.* proposed a pipeline architecture to improve the parallelism of RNN [1]. As illustrated in Figure 4, it partitions the feed-forward phase into two stages: the data flow from input to hidden layer represented by gray boxes and the computation from hidden to output layer denoted in white boxes. Furthermore, it unfolds RNN along time domain by tracing B previous time steps (usually $2 \sim 10$) and pipelines these calculations.

However, our analysis reveals that the two stages have extremely unbalanced throughputs. Assume a RNN with V nodes in the input and output layers and H nodes in the hidden layer. The input layer activates only one node at a time, so $W_{ih}\vec{x}(t)$ in Eq. (3) can be realized by extracting the row of W_{ih} corresponding to the activated node, that is, copying a row of W_{ih} to the destination vector. Thus, the computation complexity of $\vec{h}(t)$ is mainly determined by $W_{hh}\vec{h}(t-1)$, which is $O(H \times H)$. The calculation of $\vec{y}(t)$ has a computation complexity of $O(H \times V)$ because $W_{ho}\vec{h}(t)$ is

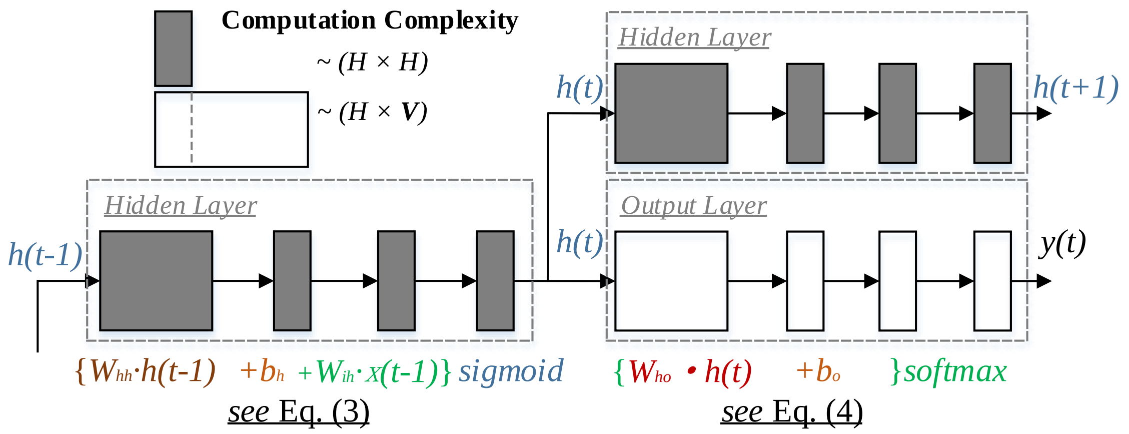


Figure 4: The data flow of a two-stage pipelined RNN structure [1]. The computation complexity of white boxes in output layer is much bigger than that of gray boxes in hidden layer.

dominant. Usually V is related to the vocabulary and can easily reach up to a size of $10\text{K} \sim 200\text{K}$ while H can maintain at a much smaller scale like $0.1\text{K} \sim 1\text{K}$. Thus, the execution time of the second stage is much longer than that of the first one. Such a pipelined structure [1] is not optimal for the entire workload.

Our effort is dedicated in further improving the execution of the second stage. As illustrated in Figure 5, we duplicate more processing elements of the output layer. More specific, our proposed architecture conducts the calculation of the hidden layer in serial while parallelizing the computation of the output layer. For example, assume B is 4. At time step $t-3$, the result of the hidden layer goes to *Output Layer I*. While *Output Layer I* is in operation at $t-2$, the hidden layer will submit more data to the next available output layer processing element, e.g., *Output Layer II*. As such, the speed-up ratio of the proposed design over the two-stage pipelined structure can be approximated by

$$\textit{Speed-up} = \frac{(t_V + t_H) + t_V \times (B-1)}{t_H \times B + t_V}, \qquad (2.7)$$

where t_V and t_H are the latencies of the output layer and the hidden layer, respectively. For instance, assume V= 10K, H= 0.1K, and B= 4, the execution of our architecture is about $3.86\times$ faster than the design of [1].

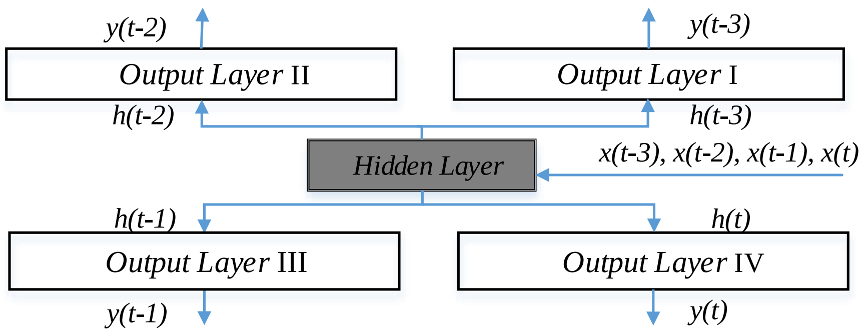


Figure 5: Our proposed parallel architecture for RNNLM.

For the proposed design, B shall be carefully selected based upon application's scale. From the one hand, a bigger B indicates more time steps processed in one iteration and therefore requires more resources. From the other hand, the higher B is, the faster execution can be obtained. Moreover, by introducing more time steps, more historic information are sustained for better system accuracy too.

2.3.2 Computation Efficiency Enhancement

Through appropriately trading off data and function precision of RNNLM, we can greatly improve its computation efficiency without degrading the training accuracy.

Fixed-point data conversion. The floating-point data are adopted in the original RNNLM algorithm and the corresponding hardware implementation, which demand significant computation resources and on-chip data space. The fixed-point operation is more efficient in FPGA implementation but the errors caused by precision truncation could accumulate iteratively. Fortunately, neural networks exhibit self-recovery characteristics, which refers to the tolerance to noise in decision making.

Mixed-precision data format. As the computation of output layer is more critical, lowering the data precision of W_{ho}, if possible, would be the most effective option. We analyze RNNLM using the Fixed-Point MATLAB Toolbox and evaluate the quality of different data format by examining the *perplexity* (PPL) of word sequences [22]. Figure 6 shows that when W_{ho} has 16 or more bits

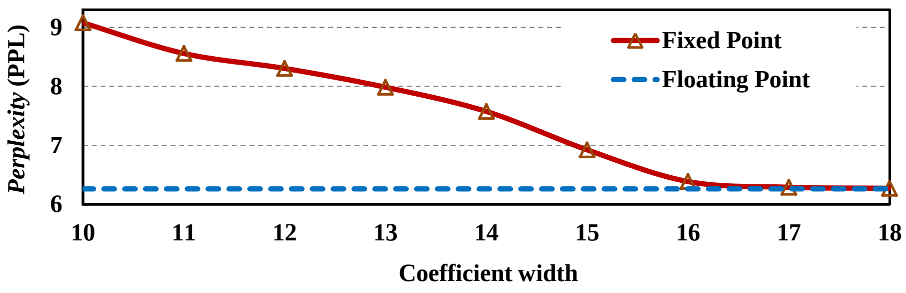


Figure 6: The impact of the reduced data precision of W_{ho} in RNNLM on the reconstruction quality of word sequences.

and keep the other training parameters as well as the states of hidden and output layers in original 64 bits, a fixed-point implementation can achieve the same reconstruction quality as a floating-point design. In other words, this scheme improves the runtime performance while maintaining the system accuracy to the maximum extent.

Approximation of activation functions. Our preliminary investigation in Table 1 reveals that the activation functions are the second contributor in runtime. This is because the complex operations in *sigmoid* and *softmax* functions, such as exponentiation and division (Section 2.1.3), are very expensive in hardware implementation. Instead of precisely mapping these costly operations to FPGA, we adopt the *piecewise linear approximation of onlinear function* (PLAN) [23] and simplify the activation functions with the minimal number of additions and shifts. Our evaluation shows that on average, the error between our approximation and the real *sigmoid* calculation is only 0.59%, which doesn't affect much on the convergence properties in RNNLM training.

2.4 HARDWARE IMPLEMENTATION DETAILS

The hardware implementation in FPGA will be presented in this section. We map the proposed architecture to *computation engines* (CEs), each of which is divided into a few *processing elements* (PEs). Moreover, the memory efficiency is improved through data allocation and reuse.

2.4.1 System Overview

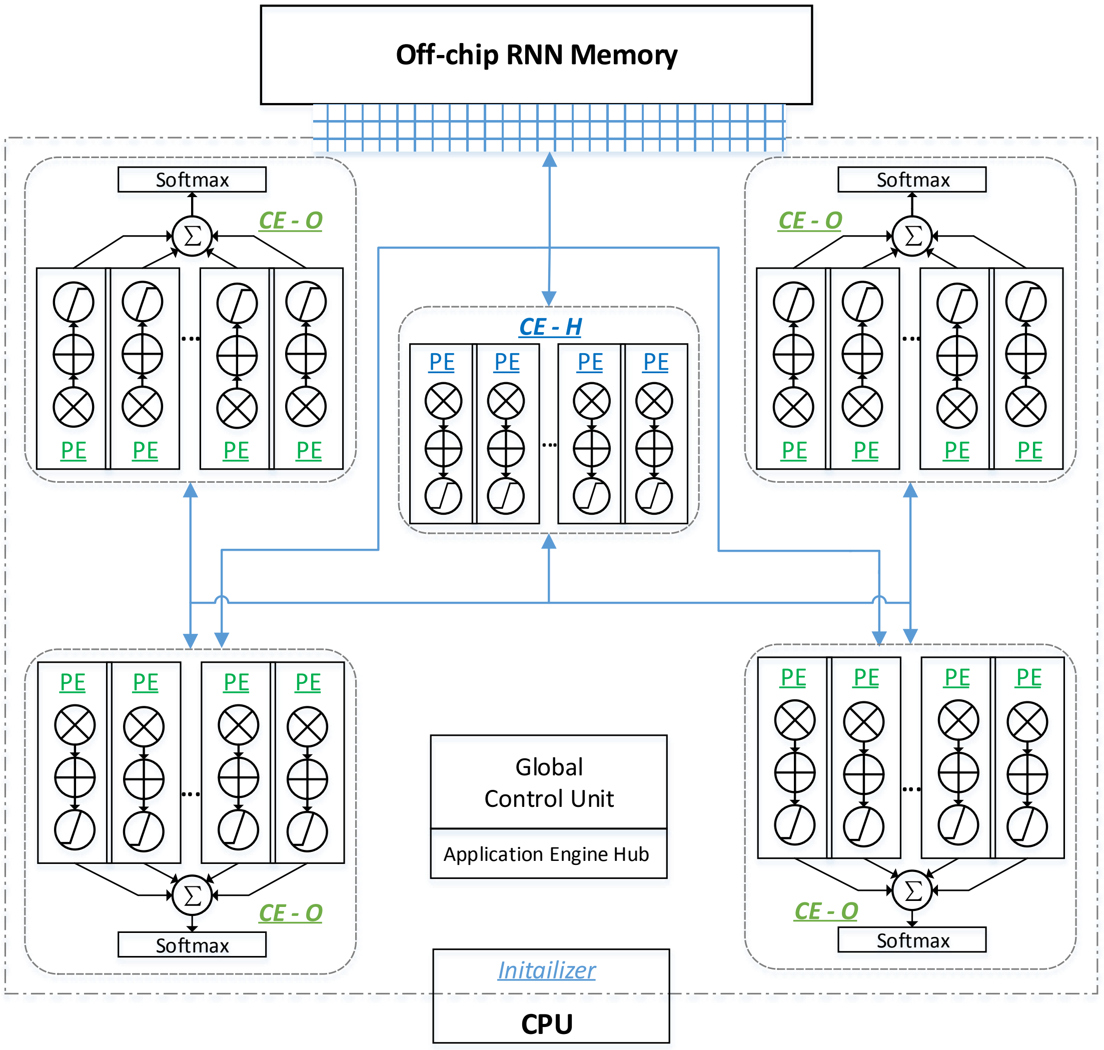


Figure 7: An overview of the RNNLM hardware implementation.

Figure 7 presents an overview of our hardware implementation on the Convey HC-2^{ex} computer system. The CPU on the host side is used for accelerator startup and weight initialization. There are 16 DIMMs and 1024 banks in the off-chip memory. Thus the chance of bank conflicts is low even the parallel accesses are random. The global control unit receives commands and configuration parameters from the host through *application engine hub* (AEH).

We map the proposed parallel architecture of RNNLM into two types of *computation engines* (CEs): *CE-H* for the hidden layer and *CE-O* for the output layer. According to Figure 5, one *CE-H* and multiple *CE-O* are required. The two types of CEs are customized for high efficiency, with the only difference in the design of activation function. The system configuration, *e.g.*, the number and scale of CEs, is upon users' decision. Moreover, each CE is segmented into several identical *processing elements* (PEs). Since the major of RNNLM execution is performed through these PEs, the proposed implementation can easily be migrated to a future device by instantiating more PEs.

The matrix-vector multiplication not only consumes the most runtime but also demands a lot of data exchange as shown in Table 2. The situation in the feed-forward phase can be partially alleviated by data streaming and datapath customization. However, the multiplication operations of transposed matrices in the back-propagation phase exhibit very poor data locality, leading to nontrivial impact on memory requests. The long memory latencies potentially could defeat the gains from parallel executions. In addition, Table 2 implies that the data accesses in RNNLM have very diverse characteristics, each of which shall be considered specifically in memory access optimizations. These details will be presented in the following subsections.

2.4.2 Data Allocation

From the one hand, the RNNLM implementation is associated with an extremely large data set, including a training data set (*e.g.*, $38\mathrm{M}$ words in our test) as well as the weight parameters (*e.g.*, $40\mathrm{Mb}$ for a vocabulary of $10\mathrm{K}$ words and the hidden layer of $1\mathrm{K}$ nodes). From the other hand, only a small amount of index data are required to control the RNNLM training process: at a time step,

Table 2: Memory Access Requirement

Dataset	**Operation**	**Total #**	**Size (byte)**
Training data	read only	$38\mathrm{M}$	$152\mathrm{M}$
W_{ih}, W_{ho}	read & write	$V \times H$ ($10\mathrm{K} \times 1\mathrm{K}$)	$40\mathrm{M}$
W_{hh}	read & write	$H \times H$ ($1\mathrm{K} \times 1\mathrm{K}$)	$4\mathrm{M}$
b_o, $\vec{y}(t)$	read & write	V ($10\mathrm{K}$)	$40\mathrm{K}$
b_h, $\vec{h}(t)$	read & write	H ($1\mathrm{K}$)	$4\mathrm{K}$

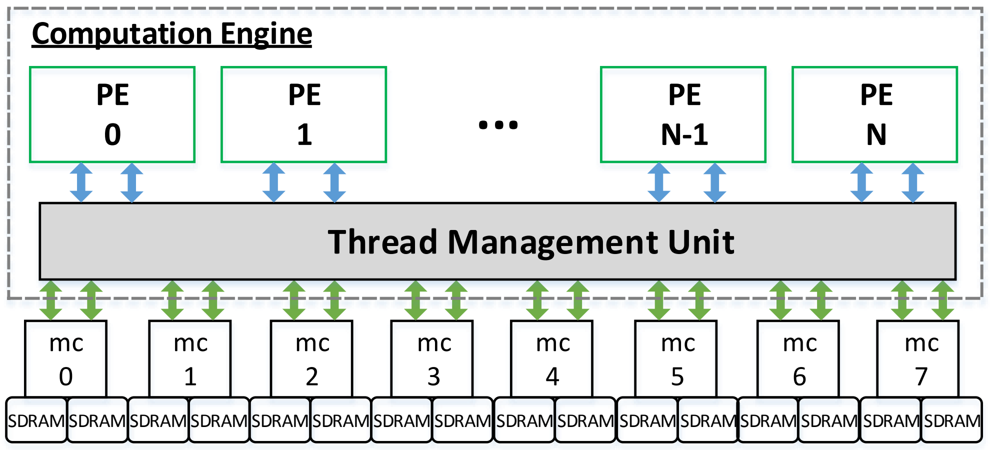


Figure 8: The thread management unit in a computation engine. PEs are connected through a crossbar to the shared memory structure.

only one input node will be activated and only one output node will be monitored. Therefore, we propose to store the training data in the host main memory and feed them into the FPGAs during each training process.

Though FPGA in the Convey HC-2^{ex} system (Xilinx Virtex6 LX760) has a large on-chip block RAM (about $26\mathrm{MB}$), not all the space is available for users. Part of it is utilized for interfacing with memory and other supporting functions. Therefore, we keep the intermediate data which are frequently access and update in the training process, such as all the parameters (W_{ih}, W_{hh}, W_{ho}, b_h, and b_o) and all the states of hidden and output layers, in the off-chip memory instead of on-chip memory. Only a subset of data is streamed into the limited on-chip memory at runtime for the best utilization and system performance.

2.4.3 Thread Management in Computation Engine

How to increase the effective memory bandwidth is critical in CE design. Previously, Ly and Chow proposed to remove the transpose of a matrix by saving all the data in on-chip block RAMs [24]. At a time, only one element per row/column of the matrix is read out through a carefully designed addressing scheme. As such, a column or row of the matrix is obtained from one memory thread.

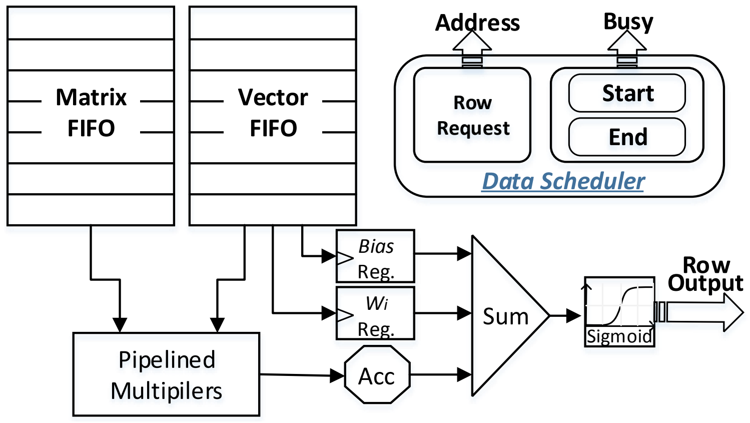


Figure 9: The multi-thread based processing element.

However, the approach requires that the weight matrix fits on-chip memory and the number of block RAMs for the weight matrix equals to the number of neurons in different layers. It is not applicable to our RNNLM in a much larger scale.

There are 16 channels in the system. To improve the memory efficiency, we introduce a hardware supported multi-threading architecture named as *thread management unit* (TMU). Figure 8 illustrates its utilization in CEs. To process all the elements of a matrix row through a single memory channel, TMU generates a thread for each matrix row and the associated start and end conditions. All the *ready* threads are maintained by TMU. Once a channel finishes a row, it can switch to another ready tread, which usually has been prefetched from memory so the memory latency is masked. Each PE holds a *busy* flag high to prevent additional threads from being assigned. When all the PEs are busy, TMU backloads threads for later assignment.

TMU supports the data communication among a large number of PEs and improves the execution parallelism. Note that there is only one TMU in a CE. Increasing the number of PEs does not introduce more hardware overhead.

Algorithm 1 Data flow from input layer to hidden layer

```
1: for $t = 0; t < BPTT; t++$ do
2:     if $t$ != 0 then
3:         $mvmulti\ (W_{hh}, hidden(t-1), hidden(t))$;
4:         $vdadd\ (hidden(t), b_h, h(t))$;
5:         $vdadd\ (hidden(t), \vec{w_{ih}^{k}}, hidden(t))$;
6:     else
7:         $vddadd\ (\vec{w_{ih}^{k}}, b_h, hidden(t))$;
8:     end if
9:     $sigmoid\ (hidden(t), hidden(t))$;
10: end for
```

2.4.4 Processing Element Design

The computation task within a CE is performed through *processing elements* (PEs). These PEs operate independently, each of which takes charge of a subset of the entire task. For example, when realizing a matrix-vector multiplication, each PE is assigned with a thread that computes a new vector value based on a row of the weight matrix. Data transition can operate in the burst mode: based on the start and end addresses, a thread fetches all the requested data in a row from the off-chip memory, as shown in Figure 9.

CE controls the memory requests of the weight matrix and vector arrays. The Convey system supports the in-order return of all memory requests, so the reordering of memory accesses can be done through TMU assisted by the crossbar interface from FPGAs to memory modules. Data returned from memory can be buffered in *Matrix and Vector FIFOs*, using the corresponding thread id as the row index. When a new thread is assigned to a PE, it raises a *busy* flag and requests the weight and vector data from memory based on the start and end addresses. Once all the memory requests for the thread are issued, the flag is *reset*, indicating that the PE is ready for another thread even through the data of the prior thread is still in processing. As such, the memory access load can be dynamically balanced across all PEs.

2.4.5 Data Reuse

Off-chip memory accesses take long time. For example, the memory latency on our Convey platform is 125 cycles at $150\mathrm{MHz}$. To speed up the execution of RNNLM, we can reduce off-chip memory accesses through data reuse.

Algorithm 1 presents the data flow from input to hidden layer, during which the state of hidden layer is frequently accessed. Similar data access pattern has also been observed in the calculation of output layer. We propose reuse buffers for matrix and bias vector respectively named as *Wi Reg.* and *Bias Reg.* as shown in Figure 9. First, a row of weight matrix are fed into an array of multipliers that are organized in fine-grain pipeline and optimized for performance. While data goes into the accumulator and completes the matrix multiplication, the weight and bias are buffered registers. After data summation is completed, PE enables its activation function, *e.g.*, *sigmoid* in *CE-H* or *softmax* in *CE-O*, to obtain the state of hidden/output layer.

2.5 EXPERIMENTAL RESULTS

In this section, we present the experimental results of the RNNLM implementation and evaluate the proposed framework in terms of training accuracy, system performance, and efficiency of computation engine design.

2.5.1 Experiment Setup

We implemented the RNNLM on the Convey HC-2^{ex} platform [25]. We described the design in System C code, which then was converted to Verilog RTL using Convey Hybrid Threading HLS tool ver. 1.01. The RTL is connected to memory interfaces and the interface control is provided by Convey PDK. Xilinx ISE 11.5 is used to obtain the final bitstream. Table 3 summarizes the resource utilization of our implementation on one FPGA chip. The chip operates at $150\mathrm{MHz}$ after placement and routing.

2.5.2 Training Accuracy

Microsoft Research Sentence Completion (MRSC) is used to validate our FPGA based RNNLM. The challenge consists of fill-in-the-blank questions [26]. The experiment calculates the score (probability) of a sentence filled with each given option and takes the option that leads to the highest score as the final answer of the model.

A set of the 19^{th} and early 20^{th} Century novels were used in training. The dataset has $38\mathrm{M}$ words and the vocabulary of the training data is about $60\mathrm{K}$. In the implementation, we merge the low-frequency words and map the remaining $10,583$ words to the output layer. For better accuracy, we set the hidden layer size to $1,024$ and BPTT to $B = 4$.

Table 13 compares the training accuracy of various language models. Our FPGA-based implementation effectively realizes long-term perspective of the sentence and beats the n-gram model. RNNME that integrates RNN with maximum entropy model [27] is able to further improve the training accuracy to 49.3%. Though vLBL+NCE5 [28] obtains the best training effect, it has far more computation cost than our RNNLM because vLBL+NCE5 used a much larger dataset $(47\mathrm{M})$, integrated a data pre-processing technique called *noise-contrastive estimation* (NCE), and analyzed a group of words at the same time.

2.5.3 System Performance

We evaluated the performance of the proposed design by comparing it with implementations on different platforms. The configuration details are summarized in Table 5. A simplified training set of $50K$ words was selected because the training accuracy is not the major focus. The vocabulary size of the dataset is $10,583$.

Table 3: Resource Utilization

	LUTs	**FFs**	**Slice**	**DSP**	**BRAM**
Consumed	176,355	284,691	42395	416	280
Utilization	37%	30%	35%	48%	39%

Table 4: Accuracy Evaluation on MSRC

Method	**Accuracy**
Human	91%
vLBL+NCE5 [28]	60.8%
RNNME-300 [27]	49.3%
RNNLM (this work)	**46.2%**
RNN-100 with 100 classes [22]	40%
Smoothed 3-gram [26]	36%
Random	20%

To conduct a fair comparison, we adopted the well-tuned CPU and GPU-based design from [1]. Furthermore, we tested different network sizes by adjusting the BPTT depth and the hidden layer size. The results are shown in Table 6. Here, *CPU-Single* represents the single-thread CPU implementation; *CPU-MKL* is multi-thread CPU version with Intel *Math Kernel Library* (MKL); *GPU* denotes the GPU implementation based on CUBLAS; and *FPGA* is our proposed FPGA implementation.

Compared to CPU-single, the performance gain of CPU-MKL is mainly from the use of MKL, which speeds up the matrix-vector multiplication. However, general-purpose processor has to

Table 5: Configuration of Different Platforms

Platform	**Cores**	**Clock**	**Memory Bandwidth**
NVIDIA GeForce GTX580	512	772 MHz	192.4 GB/s
Intel® Xeon® CPU E5-2630 @ 2.30 GHz	12	2.3GHz	42.6 GB/s
Convey HC-2ex	-	150 MHz	19.2 GB/s

Table 6: Runtime (in Seconds) of RNNLMs with Different Network Size

Hidden Layer Size	**BPTT = 2**				**BPTT = 4**				**BPTT = 8**			
	CPU-Single	*CPU-MKL*	*GPU*	*FPGA*	*CPU-Single*	*CPU-MKL*	*GPU*	*FPGA*	*CPU-Single*	*CPU-MKL*	*GPU*	*FPGA*
128	404.89	139.44	39.25	35.86	627.72	182.4	61.84	43.93	1313.25	370.37	97.448	90.74
256	792.01	381.39	49.89	69.72	1317.34	625.656	76.11	86.86	2213.64	770.02	114.71	146.46
512	1485.56	764.29	73.78	139.44	2566.80	1218.70	110.01	160.72	4925.50	2173.68	159.39	290.97
1024	2985.31	1622.56	130.45	278.88	5767.08	2242.24	191.82	327.44	10842.51	4098.85	271.13	625.94

follow the hierarchical memory structure so the space for hardware-level optimization is limited. Our FPGA design customizes the architecture and datapath specific to RNNLM's feature. On average, it obtains $14.1\times$ and $4\times$ speedups over CPU-single and CPU-MKL, respectively.

At relative small network scales, our FPGA implementation operates faster GPU because of GPU's divergence issue. Besides, GPU spends significant runtime on complicated activation functions, while the approximation in FPGA requires only a small number of additions and shifts. However, as the hidden layer and BPTT increase, the limited memory bandwidth of the Convey system constrains the speedup of FPGA. The GPU implementation, on the contrary, is less affected because GTX580 offers $10\times$ memory bandwidth. This is why GPU performs better than FPGA at large scale networks. By augmenting additional memory bandwidth to system, the performance of FPGA shall be greatly improved.

Table 6 also demonstrates the effectiveness of our proposed parallel architecture. Let's take the example of $1,024$ nodes in hidden layer. As BPTT increases from 2 to 4, implying the doubled timesteps within an iteration, the FPGA runtime increases merely 17.6%. This is because the CEs operate in a parallel format when calculating the output layer results at different time steps. When increasing BPTT from 4 to 8, the runtime doubles because only four CEs were implemented.

Besides performance, the power efficiency is also an important metric. Currently, we do not have a setup to measure the actual power. So the maximum power rating is adopted as a proxy. Table 7 shows the power consumption comparison when implementing a modest size network with 512 nodes in hidden layer and BPTT depth of 4. Across the three platforms, our FPGA implementation achieves the best energy efficiency.

Table 7: Power Consumption

	Multi-core CPU	**GeForce GPU**	**FPGA**
Run time (s)	2566.80	110.01	160.72
Power-TDP (W)	95	244	25
Energy (J)	243846 (60.69×)	26842 (6.68×)	4018 (1×)

Table 8: Computation Engine Efficiency

Platform	**Cores**	**Clock**	**Peak**	**Feed-**	**BPTT**	**Average**
	#		**GOPS**	**forward**		**Efficiency**
Single-core CPU	1	2.3 GHz	2.3	1.03	0.83	**40.43%**
Multi-core CPU	6*2	2.3 GHz	27.6	3.6	2.6	**11.23%**
FPGA-Hidden	8	150 MHz	2.4	1.68	1.11	**58.10%**
FPGA-Output	8*4	150 MHz	9.6	5.2	3.5	**45.52%**

2.5.4 Computation Engine Efficiency

The memory access optimization is reflected by the design of CEs. As a CE is partitioned into multiple individual PEs and each PE executes a subset of the entire workload, the peak performance can be calculated by [29]

$$Throughput = PE \cdot Freq \cdot Width \cdot Channel. \tag{2.8}$$

Where, *PE* is the number of PEs in each layer; *Width* represents the bit width of weight coefficients; *Freq* is the system frequency in *MHz*; and *Channel* denotes the number of memory channels.

Table 8 compares the computation energy efficiency of FPGA and CPU implementations, measured in *giga operations per second* (GOPS). The *actual sustained performance* of the feed-forward and BPTT phases are calculated by the total number of operations divided by the execution time. Note that a PE is capable of two or more fixed-point operation per cycle.

Though CPU runs at a much faster frequency, our FPGA design obtained higher sustained performance. By masking long memory latency through multi-thread management technique and reduce external memory accesses by reusing data extensively, the computation engine exhibits a significant efficiency that is greater than 45%.

2.6 CONCLUSION

In this work, we proposed a FPGA acceleration framework for RNNLM. The system performance is optimized by improving and balancing the computation and communication. We first analyzed the operation condition of RNNLM and presented a parallel architecture to enhance the computation efficiency. The hardware implementation maps neural network structure with a group of computation engines. Moreover, a multi-tread technique and a data reuse scheme are proposed to reduce external memory accesses. The proposed design was developed on the Convey system for performance and scalability evaluation. The framework shall be easily extended to large system and other neural network applications, which will be the focus of our research.

3.0 THE RECONFIGURABLE SPMV KERNEL

Sparse matrix-vector multiplication (SpMV) plays a paramount role in many scientific computing and engineering applications. It is the most critical component of sparse linear system solvers widely used in economic modeling, machine learning and information retrieval, etc. [30][31][32]. In these solvers, SpMV can be performed hundreds or even thousands of times on the same matrix. For example, the well-known Google's PageRank eigenvector problem is dominated by SpMV, where the size of the matrix is of the order of billions [33]. Using the power method for PageRank could take days to converge. As problem scale increases and therefore matrix size grows up, the runtime of SpMV is likely to dominate these applications.

3.1 PRELIMINARY

3.1.1 Sparse Matrix Preprocessing

SpMV is a mathematical kernel in the form of $y = Ax$. A is a fixed sparse matrix which iteratively multiplies with different x. The SpMV implementations in CPUs and GPUs usually are constrained by limited memory bandwidth to supply the required data and therefore cannot fully utilize the computational resources [34]. Thus the preprocessing of sparse matrix becomes very crucial.

Compression is the most common solution among sparse matrix preprocessing techniques. Many compression formats are computationally effective but they are usually restricted to highly structured matrices, such as diagonal and banded matrices [35]. The compressed row storage (CRS), instead, can effectively improve the data efficiency of generic sparse matrices. As illustrated in Figure 10, CRS utilizes three arrays to store individual nonzero elements ($value$), to keep

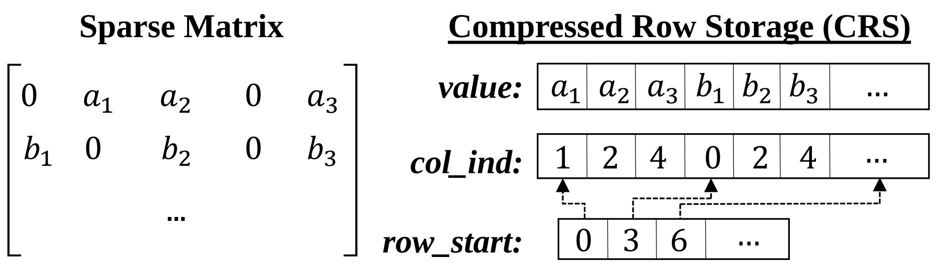

Sparse Matrix
Compressed Row Storage (CRS)
value:
col_ind:
row_start:
a_1 a_2 a_3 b_1 b_2 b_3 ...
1 2 4 0 2 4 ...
0 3 6 ...

Figure 10: An example of compressed row storage (CSR).

the column index of those nonzeros (col_ind), and to index where each individual row start in the previous two arrays (row_start), respectively. $value$ and col_ind each requires n_{nz} memory space, where n_{nz} denotes the number of non-zeros of matrix A. row_start requires only m words of space. As a result, CRS reduces the memory requirement from $\mathcal{O}(m \times n)$ to $\mathcal{O}(2n_{nz} + m)$, where n_{nz} could be two to three orders less than $m \times n$.

3.1.2 The Existing SpMV Architectures

The significance of SpMV kernel inspired many optimizations on general computing platforms. Specialized software libraries for sparse linear algebra problems, such as MKL for CPUs [36], and cuSPARSE for GPUs [37], provide standardized programming interface with subroutines optimized for the target platform.

However, even with the use of CRS or its variants [38], SpMV obtains only limited computational efficiency on CPUs and GPUs, mainly for two reasons. First, SpMV is memory bounded and thus exhibits a low flop/byte ratio. The implementations of SpMV typically demonstrate a much larger computational capability than the available memory bandwidth, leading to a low utilization of computation resources. Second, the indirect memory references of vector x introduces uncertainty to the memory accesses. Such an irregular data pattern can increase the cache misses and thus degrade the overall performance. GPUs tend to hide the latency overhead by interleaving dozens of threads on a single core. The approach works well for computation-bounded algorithms, but do not benefit much to SpMV kernel that is constrained by memory bandwidth.

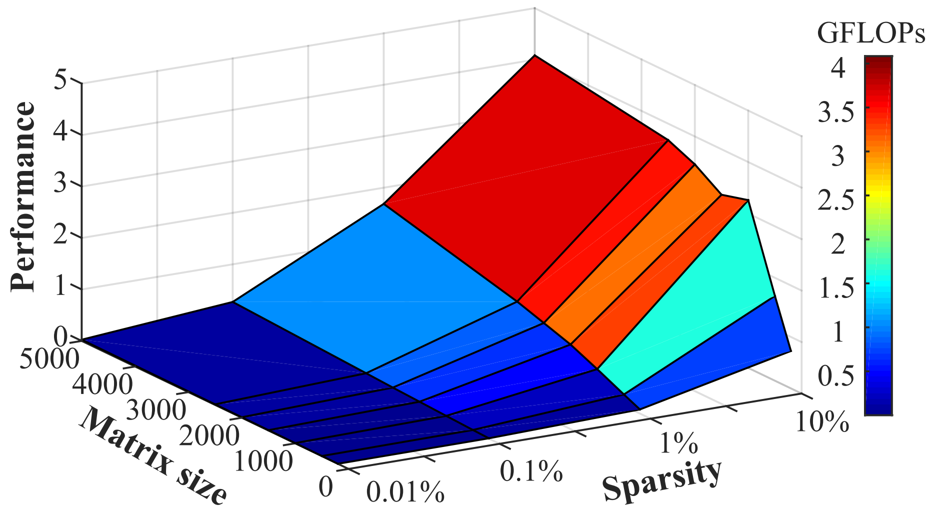


Figure 11: The sparsity affects SpMV performance.

Leveraging the flexible design fabric of FPGAs for SpMV acceleration has also been studied [39][40][41]. The efficient attempt is to parallelize multiplication operations by replicating vectors in BRAM for every multiplier and streaming in the large amount of matrix entries from external memory. The approach limits the usage of BRAM for other common purpose, *e.g.*, external transfer buffers. So it works only for the applications with small vectors. The use of the on-chip BRAM will increase rapidly for applications with large vectors, limiting the parallelism scale of implementation. The situation will get even worse considering the exponential growth of SpMV problem sizes.

3.2 DESIGN FRAMEWORK

The memory efficiency is maximized when data from main memory is stored contiguously. Every data set is used repeatedly in a short period of time and evicted afterwards without further reference. However, the operation of SpMV demonstrates an opposite situation – irregular data accesses throughout memory, leading to inefficient execution. Figure 22 shows our preliminary analysis on CRS-based SpMV implementation on Intel Xeon E5-2630 by varying the matrix size and sparsity. As the matrix sparsity decreases from 10% to 0.01%, the system performance drops rapidly because the shipping of the matrix and vector data become a bottleneck. It motivated us to analyze the performance modeling and explore new design platforms.

We started with an estimation on the lower-bound execution time needed by SpMV algorithm on an ideal architecture, assuming it has unbounded amount of hardware resources. Initially, the matrix A and the input data vector x are saved in external memory. The output data vector $y = Ax$ will be shipped out of FPGA after completing the computation. Element y_i of y is obtained through

$$y_i = \sum_{j=0}^{n} a_{ij} x_j \quad (0 \leq i \leq m). \tag{3.1}$$

Assume n_{nz} nonzero elements in A. For efficiency, most of SpMV algorithms and storage formats only operate on nonzero elements. A set of floating-point operations including one addition and one multiplication are required for each nonzero element. So the computation time required by a SpMV algorithm $T_{comp} = 2n_{nz}/F$, where F denotes the number of operation sets that can be completed in a second. Row pointers are used in storage formats, the column indies of A will also be moved into FPGAs' local memories. If n_p pointers are needed, the total I/O requirement $n_{IO} \propto n_{nz} + n_p + m + n$. Assume that the memory bandwidth is B. The total memory I/O time $T_{I/O} = n_{IO}/B$. Moreover, we consider the time used to preload data into FPGAs (T_{init}), which includes the hardware initialization and matrix preprocessing. Due to the random sparse structure of the input matrices, $T_{overhead}$ is used to represent the latency brought by the irregular data access pattern. The total execution time of a SpMV problem hence becomes

$$T = max(T_{comp}, T_{IO}) + T_{init} + T_{overhead}. \tag{3.2}$$

The goal of our design is to accelerate iterative SpMVs that have large matrices with millions of elements and vectors as large as tens of thousands of elements. Our design choices are guided by two principles to reduce T: (a) enabling high parallelism to increase F while keeping hardware complexity low, and (b) eliminating the latency overheads by improving the data-locality of the input matrix.

Figure 23 illustrates the framework structure. It clusters a large sparse matrix into modest-sized blocks with enhanced data-locality. Each block will be mapped to one computation component,

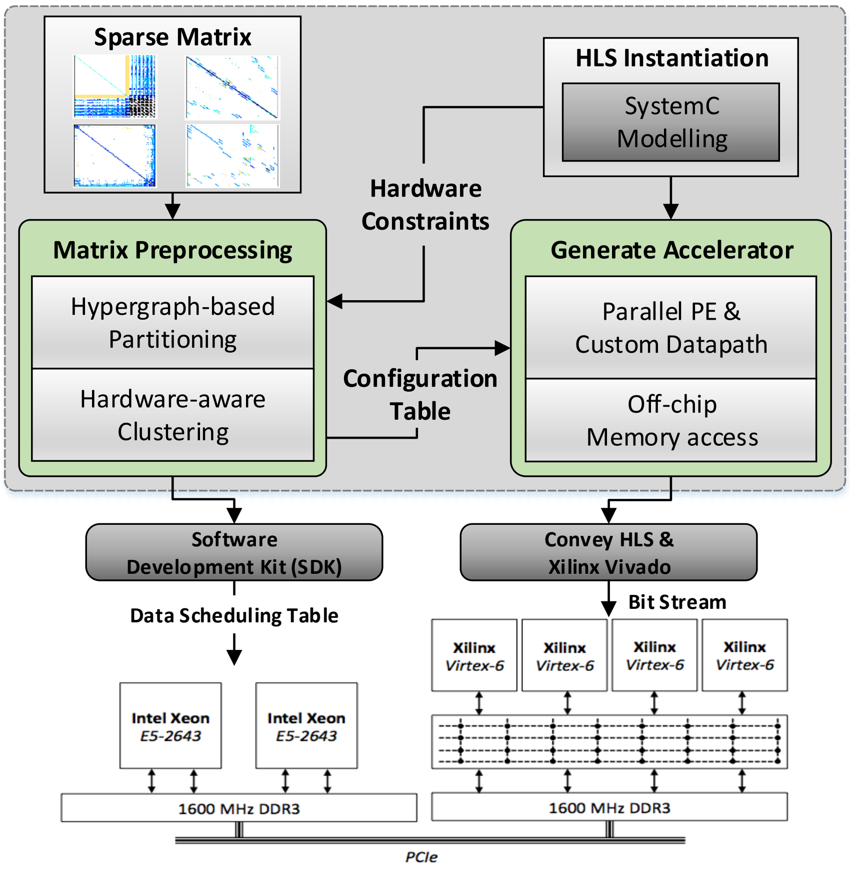


Figure 12: Our proposed data locality-aware design framework for SpMV acceleration.

namely, processing element (PE), so it can visit the same region of the input/output vectors for extensive data reuse. The large amount of configurable logic blocks on FPGA allows many PEs that can be configured according to the requirement and pattern of the given sparse matrix. The framework realizes a close coordination across software and hardware layers through the following features:

(1) Given a large-scale sparse matrix, the hypergraph-based partitioning is used to balance workload for all PEs;
(2) The partitioned matrix is clustered to separated block-diagonal (SBD) form. This step optimizes memory accesses by taking the hardware configuration into consideration;

(3) An explicit data distribution strategy maps these matrix blocks to hardware representation;
(4) The parallel execution on PEs and the inter-PE communication through customized datapath design promise the effectiveness of the SpMV kernel.

We tend to provide a general design framework that apply to different SpMV without requiring special hardware initialization or input data preparation. The design adopts a simple interface that needs only the start signal and matrix/vector addresses. The host CPUs are not required to participate in the SpMV computation. Besides, we discussed various hardware optimization approaches and their impact on computation accelerations. Examples include the configuration of PEs to enhance parallelism and the optimization of the matrix mapping to reduce I/O operations.

3.3 SPARSE MATRIX CLUSTERING

On the one hand, the parallelism of SpMV kernel relies on the preprocessing procedure which partitions and clusters a large sparse matrix into multiple memory friendly sub-matrix enhanced data-locality. On the other hand, the implementation efficiency is determined by the task and data distribution on hardware. Thus, we propose to include crucial hardware constraints, *i.e.*, the number and computation capability of PEs, into the software optimization.

3.3.1 Workload Balance

As we shall describe in Section 4.5, the computation of SpMV kernel will be distributed into parallel PEs. The matrix partitioning aims at finding a map $\pi_A : \{a_{ij} \in A\} \to \{0, \ldots, p-1\}$, which assigns each nonzero element of A to a PE. If $\pi_A(a_{ij}) = s$, then a_{ij} is said to be *local* to PE s ($0 \leq s < p$). When a PE accesses elements that are not local to it, inter-PE data movement occurs. Our goal is to minimize this communication while keeping the number of local elements balanced across all PEs.

The most common partitioning is to evenly divide m rows of the entire matrix to p PEs, say, each PE owns $\sim m/p$ consecutive rows. It is simple but could result in great load imbalance. For example, $130\times$ difference in nonzero distribution across PEs have been observed in our experiments. This disparity can cause some PEs to run out of memory, or to be much slower than other

PEs, since the SpMV computation is in fact dominated by the number of nonzero elements (n_{nz}), or, the matrix sparsity. Ideally, the most effective optimization of workload balancing is to assign each PE with the same amount of nonzeros (n_{nz}/p).

A more preferred partitioning is to model the sparsity structure of matrix A by a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{N})$ [42]. Let a vertex $v_j \in \mathcal{V}$ correspond to the j^{th} column of A. The net (or *hyperedge*) $n_j \in \mathcal{N}$ is a subset of $\mathcal{V}$ that contains exactly those vertices v_j for each nonzero a_{ij} ($a_{ij} \in n_i$). As such, partitioning a matrix A into p parts becomes to divide $\mathcal{V}$ into subsets $\mathcal{V}_0, \cdots, \mathcal{V}_{p-1}$. The connectivity λ_i of a net $n_j \in \mathcal{N}$ then is defined as $|\ \{\mathcal{V}_j \mid \mathcal{V}_j \cap n_i \neq 0\}\ |$; λ_i equals the number of parts n_i spans. The communication volume incurred during SpMV is $\sum_{i:n_i \in \mathcal{N}}(\lambda_i - 1)$, which is the $(\lambda - 1)$-matric [34]. The partitioning starts with a complete $\mathcal{V}$, and each iteration splits a $\mathcal{V}$ into two parts. After $n-1$ iterations, a final partition of $\mathcal{V}_0, \cdots, \mathcal{V}_{p-1}$ is generated which minimizes the communication volume and satisfies the load constraint of $n_{nz}(\mathcal{V}_i) \leq (1+\epsilon)\frac{n_{nz}}{P}$, where ϵ is a predefined unbalance factor. Figure 13(a) gives a 3937×3937 matrix with a sparsity of 0.163%. The hypergraph-based partitioning can divide it into four parts denoted in different colors shown in Figure 13(b).

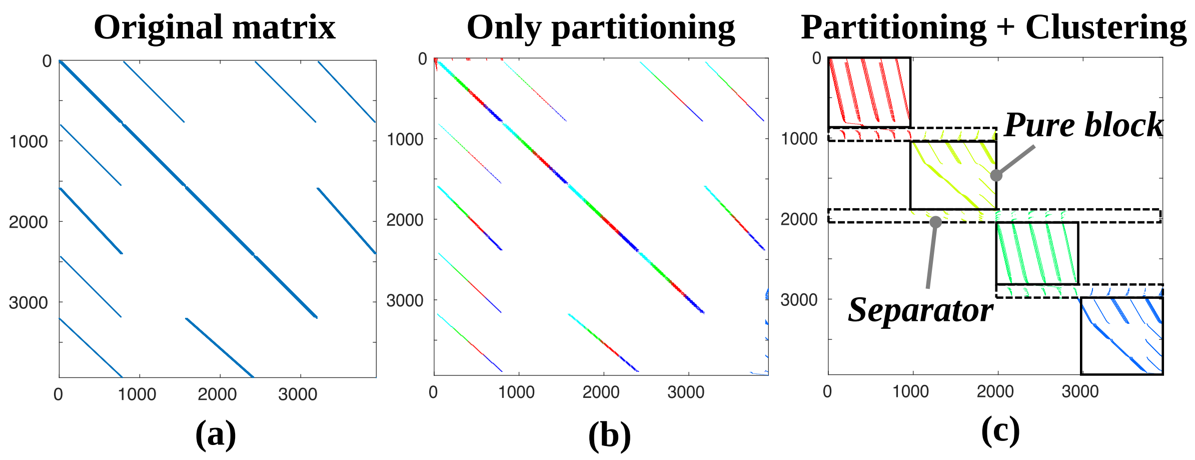


Figure 13: Applying the partitioning and clustering on a sparse matrix (`lns_3937`).

3.3.2 Hardware-aware Clustering

While partitioning breaks up the large amount of nonzeros for parallel computing, the data locality structure inherent with the original sparse matrix has not been fully explored. Considering that each element of the sparse matrix is used only once, repeated accesses can be made only for vector data. The key of improving memory efficiency then is to allocate the matrix elements in contiguous chunk of memory and keep the associated vector components in on-chip memory as long as needed. Here, we apply row and column permutations to form p smaller local matrices $A^{(s)}$ $(s = 0, \cdots, p-1)$ with higher data density. As such, each PE only needs to load a single local matrix and a small section of the input vector for calculation, *i.e.*, $y^{(s)} = A^{(s)}x^{(s)}$.

The clustering problem can be solved by recursive bisection. During each iteration that splits a subset of $\mathcal{V}$ into two parts, $\mathcal{V}_{left}$ and $\mathcal{V}_{right}$, the hypergraph nets can be divided into three categories: $\mathcal{N}_+$, $\mathcal{N}_-$ and $\mathcal{N}_c$ which respectively contain nonzeroes from only $\mathcal{V}_{left}$, only $\mathcal{V}_{right}$, and both $\mathcal{V}_{left}$ and $\mathcal{V}_{right}$. Having defined these categories, we can reorder the rows of the matrix as follows: all the rows in $\mathcal{N}_+$ are permuted towards the top of the matrix, whereas those in $\mathcal{N}_-$ are permuted to the bottom. The remaining rows are left in the middle. This creates two row-wise boundaries in the matrix, *i.e.*, dividing lines between blocks of rows. Altogether, the boundaries split the permuted matrix into three blocks: the large upper-left and lower-right blocks are denoted as *pure blocks*, while the block in the middle is referred as a *separator*. Applying the bisection scheme recursively can obtain p partitions, corresponding to p $pure\ blocks$ and $p-1$ $separators$ and cluster the matrix to separated block diagonal (SBD) form.

To improve the efficiency in regularizing the matrix sparsity structure, we integrated the matrix clustering with a lightweight one-dimensional partitioning. Optimal hypergraph-based partitioning is known to be computational intensive. Our adopted partitioner instead maps nonzero elements only according to its row index. Comparing with its alternatives such as two-dimensional [43] and fine-grained [44] partitioning, it processes the matrix with running time linear to the matrix size. Furthermore, this combinational solution splits the output vector y in contiguous blocks. Each PE thus accesses a unique block of y and avoids concurrent writes on the output vector, reducing the data hazards and complexity in hardware design. Figure 13(c) gives the clustering results of the given example when $p = 4$.

Algorithm 2 Partitioning & hardware-aware clustering

```
Input:  Recursive function call : SBD_Gen($A$, p, $\epsilon$)
        $p$ = number of PEs, which take consideration of computation efficiency;
        $\epsilon$ = allowed load imbalance, $\epsilon > 0$.
Output: Matrix $A$ translated in SBD form.
1: if $p > 1$ then
2:     ($A_0^r$, $A_1^r$) : = hypergragh_partition(A, $\epsilon$);
3:     ($A_0$, $A_1$) : = clustering ($A_0^r$,$A_1^r$);
4:     $maxnz$ : = $\frac{n_{nz}(A)}{p}(1+\epsilon)$;
5:     $\epsilon_0$ : = $\frac{maxnz}{n_{nz}(A_0)} \cdot \frac{p}{2}$-1; SBD_Gen($A_0$, p/2, $\epsilon_0$);
6:     $\epsilon_1$ : = $\frac{maxnz}{n_{nz}(A_1)} \cdot \frac{p}{2}$-1; SBD_Gen($A_1$, p/2, $\epsilon_1$);
7: end if
```

3.3.3 Strong Scaling vs. Weak Scaling

After matrix clustering, each PE performs local computations for one sub-matrix $A^{(s)}$ and all PEs can operate in parallel. The major performance penalty then comes from the computation on $separators$ which may require multiple subsets of input vector, causing inter-PE communications.

The examples in Figure 14 show the impact of the $separator$ computation and induced data transition. The two benchmarks are selected from the University of Florida sparse matrix collection and the performance are evaluated on an Intel Xeon E5-2630 with Matlab Parallel Toolbox [35]. `Memplus` shows a *strong scaling* case, that is, the runtime decreases as more cores are deployed. In contrast, for `stanford`, distributing the matrix into more PEs results in a large number of communications induced by $separators$. As the number of PEs increases, the computation costs become relatively small while the communication time starts dominating the system performance. Deploying more PEs makes the SpMV execution slightly slower, demonstrating a typical *weak scaling*. The observation inspired us to develop a customized datapath within PE for inter-PE communications, which shall be introduced in Section 3.4.2.

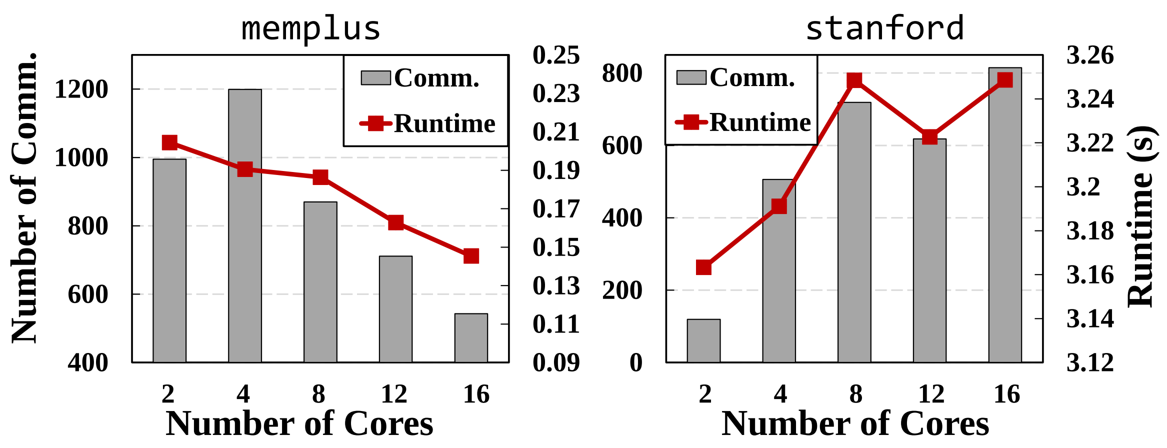


Figure 14: The performance penalty brought by inter-PE communications.

3.3.4 Hardware Constraints

Ideally, SpMV reaches its ultimate performance when the number of PEs $p \to \infty$ with a memory hierarchy that is able to feed data at any time with no latency. Although a large p is preferred theoretically, there is an optimal range of p in FPGA implementation, which is mainly determined by the available computation resource and the memory efficiency.

The number of MACs. As exclusive compute elements in PEs, MACs are implemented using dedicated DSP blocks or reconfigurable logic. Assume r_{mac} units of resource are needed to construct a single MAC and R units in total are available, the number of MACs is limited by $\frac{R}{r_{mac}}$.

The effective BRAM space to store vector for data reuse. Although FPGAs usually provide a large BRAM capacity (*e.g.*, 26Mb in Xilinx Virtex6 LX760 used in this work), not all of them is available for user applications. Excluding the portion for interfacing to memory and other support functions, αM can be used for buffering the vector data. Since a sparse matrix is clustered into many sub-matrices with a vector size of m_{sub}, p cannot exceed $\frac{\alpha M}{m_{sub} \cdot DW}$, where DW represents the data width.

The memory channels to off-chip DRAM ($N_{channel}$). An input sub-matrix is initially stored in off-chip memory. The nonzero elements and indices are streamed into a PE by deploying a specific number of memory channels, *e.g.*, 2 in our design as explained in Section 3.4.3. This limitation will constrain p as $\frac{N_{channel}}{2}$.

The smallest data block. As p increases, less workload will be distributed to each PE. Note that the initialization cost of very small matrices negatively affects system performance, we define a minimal number of nonzeros in a sub-matrix as nz_{inital}. So the selection of p should not be larger than $\frac{n_{nz}}{nz_{initial}}$.

These hardware constraints shall be taken during the matrix partitioning and clustering process by considering a reasonable number of PEs limited by

$$p = \left\lfloor min(\frac{R}{r_{max}}, \frac{\alpha M}{m_{sub} \cdot DW}, \frac{N_{channel}}{2}, \frac{n_{nz}}{nz_{initial}}) \right\rfloor. \tag{3.3}$$

3.4 HARDWARE IMPLEMENTATION

Figure 15 presents the architecture of our hardware implementation on a Convey HC-2^{ex} computer system [25]. The CPU on the host side clusters matrices and generates configuration header files to initialize the SpMV kernel on FPGAs. The global control unit (GCU) receives the header files, maps local matrices to processing elements (PEs), and manages the communications in between. A thread management unit (TMU) controls the operation of PEs.

Our design requires only one copy of input vector in on-chip BRAM while storing the large matrix on off-chip DRAM. In order to mitigate the communication overhead brought by $separators$, we realize a customized one-way datapath to transit intermediate result of $separators$ between PEs. PEs are the main computation power, each of which consists of an optimized CRS-based SpMV kernel. The number of PEs is mainly determined by FPGA resources. Thus the proposed architecture can be mitigated into larger designs by instantiating more PEs.

3.4.1 Global Control Unit (GCU)

The GCU is used to map the clustered matrix into the FPGA architecture. More specifically, it maps $A^{(s)}$ to each PE and produces corresponding memory configuration. Figure 16 illustrates an example matrix which is clustered in SDB form and mapping to four PEs. The configuration header file of PE #1 is shown in the table. GCU will assign the data blocks from the same *resource* to one single PE. For a $pure\ block$, the computation can be completed locally within the PE. A $separator$,

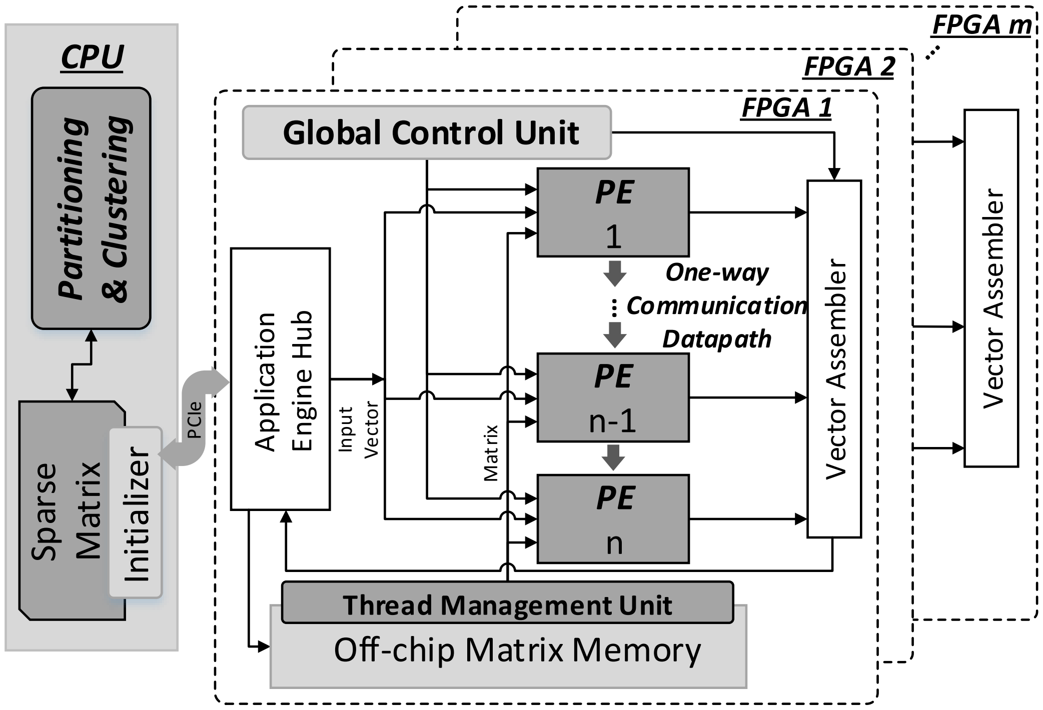
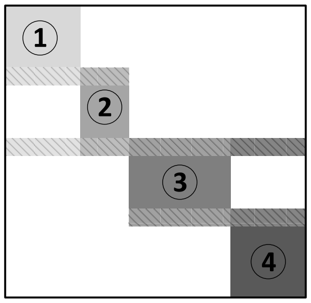


Figure 15: The architecture for SpMV acceleration.



Configuration Header File for PE #1

Data Type	***Resource***	***Destination***
pure block *#1*	①	①
separator *#1*	①	②
separator *#2*	①	④

Figure 16: Configuration header file is used to schedule inter-PE communication.

instead, need to be propagated through multiple PEs while the communications only occur between adjacent PEs. So a customized one-way datapath is designed to transfer intermediate results of $separator$ computation. The datapath is implemented by a low-latency FIFO structure within PE, as depicted in Figure 17.

After configuration, each PE performs the matrix operations in two steps. First, it reads a segment of input vector which is referenced by the $pure\ block$ # and copies it into an on-chip BRAM. Afterwards, the local SpMV multiplications will be conducted. For the $pure\ block$, the output will be directly written into the output buffer, while the result of a $separator$ will be delivered to its adjacent PE until it reaches the destination PE.

3.4.2 Processing Element (PE)

The core of the PE design is a double-precision streaming multiply-accumulator (MAC). To exploit the full computational throughput, we tend to enhance the pipelining of the dot-product accumulations. The accumulation depends on the result from its previous MAC operations, we deploy multiple parallel multipliers with an adder tree structure to interleave the independent dot products on a single floating-point MAC pipeline, as depicted in Figure 17.

With the matrix clustering, a local sub-matrix corresponds to only a portion of an input vector. A set of dual-port, high-bandwidth on-chip BRAMs are implemented in PE as a vector bank, to store and supply these vector elements. The data from the vector bank will be paired with its counterpart in the local matrix. Finally, a multiplier pulls the pair out of their respective FIFOs, conducts the multiplication, and feeds the result into the adder tree. The adder tree is responsible for the accumulation of both local pure block and the intermediate result of separator. When the multiplication and accumulation completes, the results of $pure\ block$ or $separator$ will be respectively placed in the output buffer or sent to the destination PE.

3.4.3 Thread Management Unit (TMU)

The memory efficiency of streaming large matrices from off-chip memory is also critical for system performance improvement. The Convey HC-2^{ex} system adopted in the work has four Xilinx Virtex-6 FPGAs connected to eight DRAM chips via a full crossbar that supports memory request

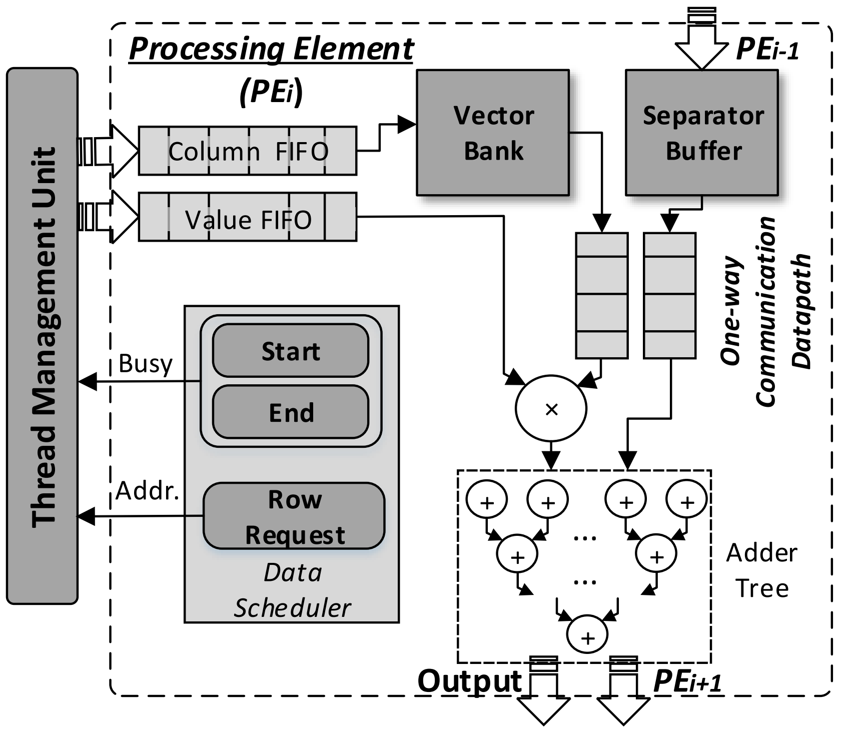


Figure 17: The internal structure of processing element design.

reordering [25]. Each memory controller provides two access ports, each of which can read/write eight bytes per cycle at $150\mathrm{MHz}$. Each FPGA has 16 memory channels, delivering a peak bandwidth of $19.2\mathrm{GB/s}$. We process the matrix data from off-chip memory through parallel channels to maximize the bandwidth utilization. The local matrices with the same *resource* can leverage the same channel.

We design TMU, a multi-threading architecture, to feed the clustered matrix to different PEs in parallel. For each row of $A^{(s)}$, TMU creates a thread and the associated start and end conditions. All the ready threads are maintained in TMU. Each PE buffers the received data in the value and column FIFOs for MACs. Once a channel exhausts a row, it switches to a new ready thread, the data of which usually has been prefetched from memory. A PE in operation holds a busy flag high to prevent additional threads from being assigned. When all the PEs are busy, TMU backloads

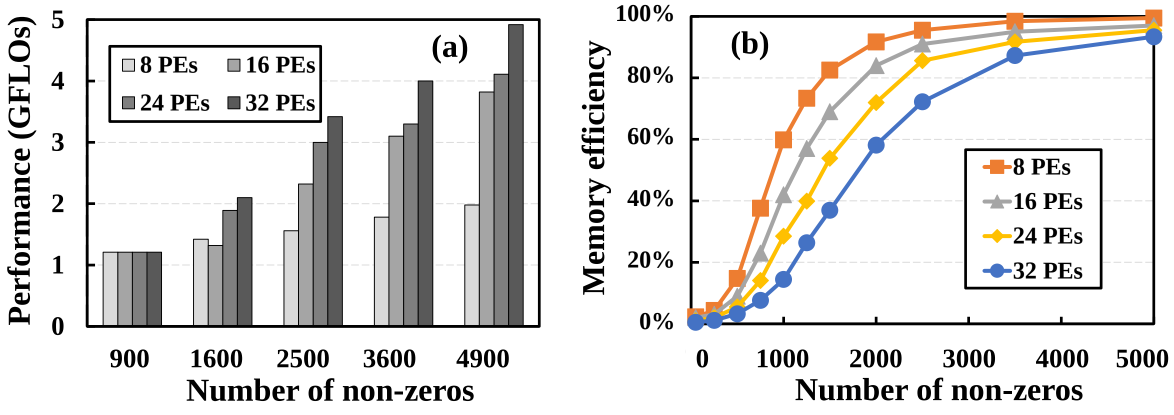


Figure 18: Hardware constraint analysis by varying the size of input matrix/PE.

threads for later assignment. TMU supports the data communication among a large number of PEs and improves the executable parallelism. Note that every FPGA requires only one TMU so its overhead is minimal.

3.4.4 Hardware Configuration Optimization

TMU can effectively mask the memory latency and assist the parallel operation of PEs, but the initialization stage is dominated by memory requests until TMU buffers sufficient ready threads. The initialization cost can be alleviated over the execution of large matrices while negatively affects the system performance for small matrices. Considering the scenario, utilizing a fixed configuration and occupying all the PEs could be a waste in some applications. The optimal selection of PEs is a trade-off of throughput and hardware resources, which in turn will guide the matrix clustering in the preprocessing procedure.

We analyze this hardware constraint by investigating the impact of PEs' parallelism on system performance and memory efficiency. The tests are conducted by distributing a single dense matrix block stored in CRS format to every PE. Particularly, dense matrices are chosen here to eliminate the irregular data access during execution and obtain a more accurate initialization cost estimation. The tests utilize all the four FPGAs on Convey HC-2^{ex} with 8 PEs implemented on each FPGA chip. As seen from Figure 18, if the number of non-zero (nz_{sub}) assigned on a single PE is less

Table 9: Resource Utilization

	LUTs	**FFs**	**Slice**	**DSP**	**BRAM**
Consumed	331,173	284,691	42395	440	249
Utilization	42%	30%	35%	51%	32%

than $nz_{init} = 2,500$, deploying more PEs does not lead to better operation speed due to the low memory efficiency brought by the initialization overhead. Therefore the optimal selection of PEs can be calculated by n_{nz}/nz_{init} when the input matrix is known.

3.5 EVALUATION

In this section, we present the experimental results of the SpMV implementation and evaluate the proposed framework in terms of system performance, computation efficiency and power consumption.

3.5.1 Experimental Setup

We implemented the proposed design framework on Convey HC-2^{ex} platform [25]. The SpMV kernel is described in SystemC fashion, which is converted to Verilog RTL using Convey Hybrid Threading HLS tool ver. 1.01. The RTL is connected to memory interfaces and the interface control is provided by Convey PDK. Xilinx ISE 11.5 is used to obtain the final bitstream. The chip operates at 150MHz after placement and routing. Table 16 summarizes the resource utilization of our implementation on one FPGA chip. The design consumes only one third of BRAMs benefiting from the matrix clustering and the vector reuse.

Ten benchmarks are selected for direct comparisons with the results reported in [45][46]. We include four more benchmarks to evaluate our approach on large-scale SpMV. The selected matrices cover a wide spectrum of non-zero distribution from 2,000 to more than 6 million. Irregularity

Table 10: Characteristics of Benchmarks

Sparse Matrix	**Rows** m	**Non-zero** n_{nz}	**Sparsity**	**PEs** p
lns_3937	3,937	25,407	0.163%	8
raefsky1	3,242	293,409	2.791%	8
psmigr_2	3,140	540,022	5.477%	16
dw8192	8,192	41,746	0.062%	16
t2d_q9	9,801	87,025	0.091%	32
epb1	14,734	95,053	0.006%	32
torso2	115,967	1,033,473	0.008%	32
memplus	17,758	99,147	0.031%	32
s3dkt3m2	90,449	4,427,725	0.054%	32
stanford	281,903	2,312,497	0.003%	32
rma10	46,835	2,329,092	0.106%	32
consph	83,334	6,010,480	0.086%	32
cant	62,451	4,007,383	0.103%	32
qcd5_4	49,152	1,916,928	0.079%	32

Table 11: Configuration and System Performance of Different Platforms

Platform	Opti.	Cores	Clock	Memory Bandwidth	Power (TDP)	Peak GFLOP	Sustained GFLOP	Computation Efficiency
NVIDIA GeForce GTX Titan X	cuSPARSE	3072	1075 MHz	336.5 GB/s	250 W	206.6	6.7	3.2%
Intel Xeon E5-2630	MKL	12	2.3 GHz	42.6 GB/s	95 W	110.4	2.4	2.1%
Convey HC-2^{ex}	-	32	150 MHz	19.2 GB/s	25 W	9.6	5.6	58.3%

within a matrix varies from a few non-zero to hundreds of non-zero elements per row. Configuration to hardware is generated for each matrix prior computation. Table 10 details these benchmarks and they are all publicly available from the University of Florida Sparse Matrix Collection [47].

3.5.2 System Performance

The system performance is measured in double precision giga operations per second (GFLOPs). For our FPGA implementation, each PE is capable of two floating-point operations per cycle. With maximal 32 PEs running at 150MHz, *the peak computation performance* is 9.6 GFLOPs.

We report *the sustained performance* as the ratio of $2n_{nz}$ over the entire runtime and *the computation efficiency* is calculated by the ratio of the sustained performance over the peak performance. Across all the benchmarks, our FPGA-based SpMV kernel achieves an average sustained performance of 5.6 GFLOPs, corresponding to a computation efficiency of 58.3%.

To demonstrate the effectiveness of the proposed design, we compare it to well-tuned SpMV implementations on CPU and GPU. The CPU measurements are carried out on an Intel Xeon E5-2630 running a multi-threaded SpMV implementation from Intel's MKL library. The GPU implementation is preformed on a high-end Nvidia GTX Titan X running the latest version of cuSPARSE. The configuration details are summarized in Table 15. Our design aims at accelerating iterative SpMV, so we repeated the execution of each sparse matrix with 100 random vectors to obtain the average performance. The runtime are measured without including the data transferring time between host and co-processor memories because our FPGA-based platform and GPU adopt different shared memory structures.

Figure 19 shows the detailed performance results across the experimental set. Small matrices perform poorly on GPU because of the initialization overhead and insufficient amount of workload

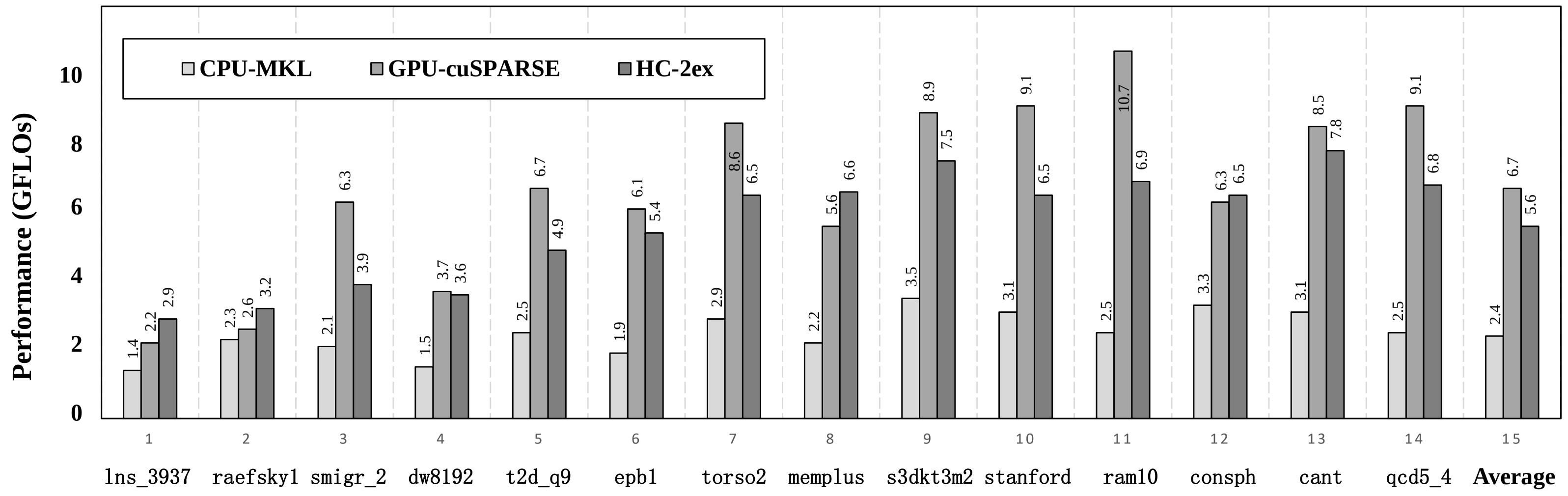


Figure 19: System performance on selected benchmarks.

to keep the GPU busy. These matrices obtain better performance on CPU because less parallelism is needed for the efficient execution on CPU. As the scale of matrices increases, the overhead dominated in the former category is alleviated over the long SpMV execution time and thus the performance improves.

Our FPGA design achieves 2.3× speedup over CPU implementation and a comparable performance as GPU implementation, not even mentioning that CPU/GPU runs at a much faster frequency and has a larger memory bandwidth. According to the performance model of Equation (2), the system performance is restricted by $max(T_{comp}, T_{IO})$. Though the sole computation time decrease as p increases, the limited memory bandwidth of the Convey system could constrains the speedup of matrices in very large size. The GPU implementation is less affected because GTX Titan offers much larger memory bandwidth (17.5×). $T_{overhead}$, the latency overhead brought by the irregular data access pattern, is another limiting factor. Compared to the FPGA platform, GPU potentially can obtain $21.5\times$ peak performance for the large amount of processing core. However, it allocates less cache space to each core, leading to rather high penalties in efficiency when cache misses happen. By augmenting additional memory bandwidth to the FPGA-based platform, the system performance of FPGA shall be further improved.

Besides performance, *power efficiency* is another important metric which is measured by sustained performance per thermal design power (TDP). Across the three different platforms, our FPGA implementation achieves the best power efficiency, which is 8.9×/8.3× higher than CPU/GPU.

3.5.3 The Impact of Data Preprocessing

We analyze the impact of the data preprocessing on overall system performance. Figure 20(a,b) reports the performance of two benchmarks `memplus` and `stanford`, respectively representing the strong and weak scaling matrices (refer Section 3.3.3), under different implementations. Here, the performance is measured by the ratio of operation number and runtime (in GFLOPs) to complete the data preprocessing followed by one hundred SpMV executions with random input vectors.

As expected, the CPU implementation without clustering exhibits the worst performance for the irregular data access patterns greatly reduce the cache efficiency. Deploying clustering can

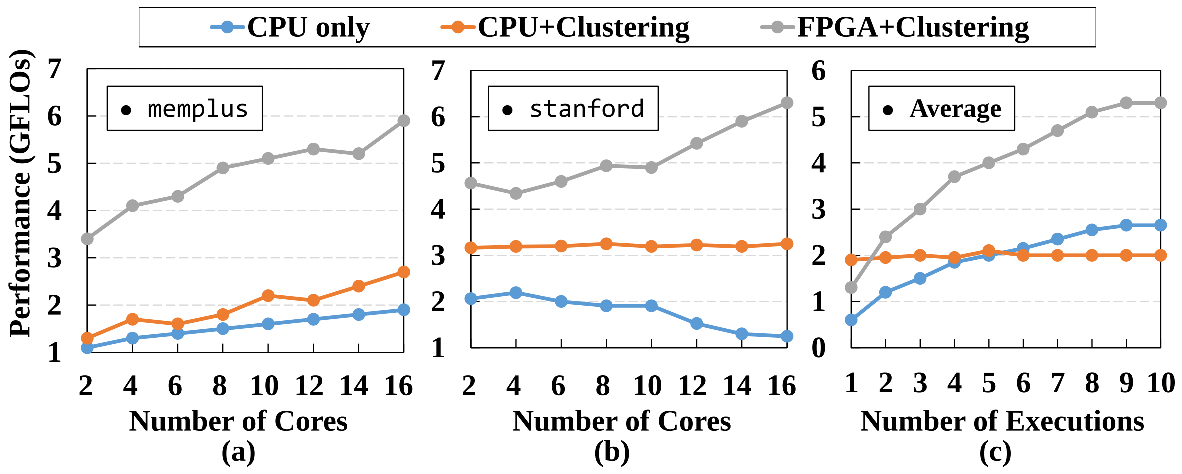


Figure 20: The analysis on the data preprocessing.

regularize the sparsity structure of the input matrix, increasing the cache hit rate and therefore improving the performance. The comparison of the CPU and FPGA implementations demonstrates the efficiency of our proposed design framework. After clustering, the major performance penalty comes from the inter-PE communications. The parallel structure of PEs and the customized one-way datapath on FPGA mitigate the overhead brought by *separators* Even `stanford` that exhibits a weak scaling obtains performance gain as more PEs are deployed.

Due to the slow execution of hypergraph-based partitioning, the data preprocessing takes longer time than one SpMV computation, even a lightweight partitioner is adopted in our design. We compare the average performance across the experimental set and shows the results in Figure 20(c). To investigate the overhead of data preprocessing, the performance is measured when executing the data preprocessing with varying the number of SpMV executions. The random data distribution (without preprocessing) exhibits a constant performance. While the clustering-based distribution loses some of its advantage over the random distribution at the very beginning, the fast computation will quickly amortize the additional cost in $2 \sim 3$ SpMV executions. For many applications like eigensolver or machine learning that typically need thousands of SpMV operations, the proposed design frame have a great advantage.

Table 12: System Properties for Previous Implementations and This Work

	[2012]	**[2014]**	**[2015]**	**[2014]**	**[2011]**	**This work**
FPGA	Virtex-5	Stratix-III	Virtex-II	Stratix-V	Virtex-5	Virtex-6
Frequency [MHz]	100	100	200	150	150	150
Number of PEs	16	6	4	32	32	32
Matrix Format	CVBV	COO	CRS	CISR	CRS	CRS
Precision	Double	Double	Single	Single	Double	Double
Computation Efficiency	4.48%	5.63%	42.6%	40.6%	25.1%	58.3%

3.5.4 Comparison to Previous Designs

We compare our design with a few existing SpMV FPGA architectures and summarize the results in Table 12. These works can be categorized into three approaches. The first one targeted at improving the effective memory bandwidth by encoding sparse matrix [48][49]. The compressed variable-length bit vector (CVBV) format [49] is applied, which obtained $1.14 \sim 1.40\times$ higher compression ratio than CRS. However, these techniques aggravate hardware complexity and have marginal improvements in computation efficiency.

The second approach is to parallelize several PEs with a reduction circuit or adder tree [50]. The partial products of output vector are added and the resulting sum is then fed into a customized accumulator that handles the potential data hazards. A drawback of this approach is that it requires zero padding to achieve a minimum row size. Furthermore, it is sensitive to the matrix sparsity structure and performs poorly for extremely sparse matrices ($< 0.1\%$ density). In contrast, our design is able to handle matrix densities below 0.01% and maintains a high computation efficiency.

Replicating input vectors to eliminate unnecessary computational stalls is another common approach [45][46], which usually requires very high usage of BRAM. Nagar et al. [46] implemented their design also on a Convey platform with Vertex-5 FPGA boards. As aforementioned that SpMV kernel is memory bounded and the computational resource is not the major constraint of system performance. However, their average computation efficiency is only 25.1%. One reason is that the proposed cache architecture performs poorly on large irregular matrix. For example, the performance on their largest benchmark `torso2` drops significantly. Our design, in contrast, is able to analyze the sparsity structure of each matrix, distributes workload according to optimized parallel PEs, and therefore maintains a higher GFLOPs on large matrices. Overall, our design achieves an average computation efficiency of 58.3%, 2.3× speed-up over the implementation of [46].

3.6 CONCLUSIONS

In this work, we propose a data locality-aware design framework for FPGA-based SpMV acceleration by maximizing the utilization of available memory bandwidths and computing resources. We first cluster a large matrix into memory-friendly blocks to enable efficient data reuse of the same regions of both the input and output vectors. Then an explicit mapping strategy is applied to distribute the matrix blocks onto parallel PEs, maximizing the number of simultaneous multiplication-accumulation computations. Experiments on Convey system shows that our technique achieves an average computation efficiency of 58.3%, which outperforms the optimized CPU and GPU counterparts $18.2\times$ and $27.8\times$, respectively.

4.0 SPARSE CONVOLUTIONAL NEURAL NETWORKS ON FPGA

Deep convolutional neural networks (CNNs) that have a large number of parameters have broken many performance records in image recognition and object detection applications. Recent studies show that network sparsification can effectively reduce the model size while retaining accuracy, which further extends the potential of CNNs. However, sparsification techniques at the algorithm level often generate irregular network connections, resulting in low hardware implementation efficacy and marginal speedup. Many prior FPGA practices that achieved great success in dense CNNs are not applicable to sparse models. The work presented in this paper leverages algorithm-level sparsification techniques to relax certain constraints on the underlying hardware, leading to a software/hardware co-design framework that achieves significant improvement in computational performance and energy efficiency.

4.1 INTRODUCTION

Following the technology advances in high-performance computing systems and the fast growth of data acquisition applications, machine learning achieved remarkable commercial success [51]. Particularly, convolutional neural networks (CNNs) that originate from the working mechanism of receptive fields in visual cortex [52] broke many records in image recognition and object detection problems [53]. The success of CNNs, to a great extent, is enabled by the fast scaling-up of network that learns from a huge volume of data. The deployment of deep CNN models are both memory-intensive and computation-intensive, facing severe challenges on efficient implementation.

In recent years, *sparsification* techniques that prune redundant connections of deep neural networks (DNNs) while still maintaining accuracy emerge as a promising solution to decrease the model size and therefore reduce the computation requirement. The approach is usually realized at the software level: kernel weights are sparsified and then compressed to minimize the memory footprint of CNN invocation and the use of computation units [54][55]. However, randomly removing network connections results in data misalignment so the memory accesses of the compressed network exhibit poor locality. The increased cache misses and latency overheads could greatly degrade the overall system performance. For example, previous sparse CNN implementations on general-purpose computation platforms reach only 0.1∼10% of system peak performance [56], even applying designated software libraries, e.g., MKL library for CPUs [36] and cuSPARSE library for GPUs [37]. Han *et al.* [57] proposed the EIE architecture for compressed networks that reduces the parameters of *fully-connected* (FC) layers. As the latest CNN models adopt fewer FC layers (e.g., only 0.4% arithmetic operations of VGG-16 from FC layers [58]), the acceleration of *convolutional* (Conv) layers that involves fewer parameters but more extensive computation time and hardware utilization becomes a critical concern.

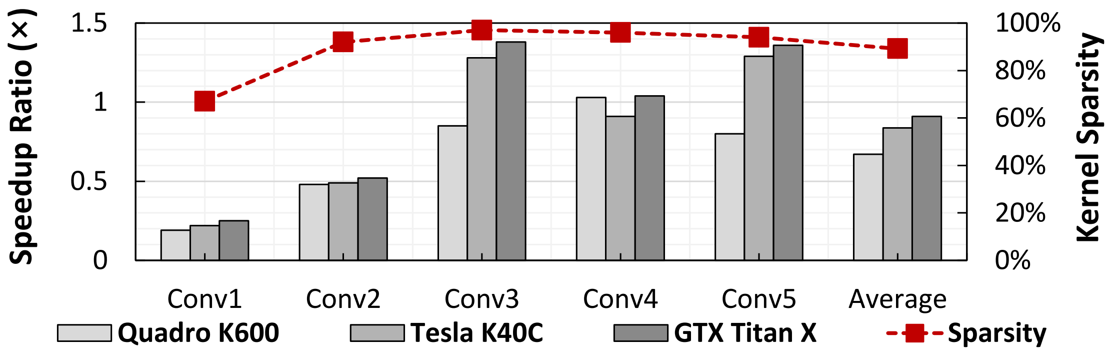


Figure 21: Our evaluation on AlexNet sparsity and speedup. Conv1 refers to convolutional layer 1, and so forth. The baseline is profiled by GEMM of Caffe. The sparse kernel weights are stored in compressed sparse row (CSR) format and accelerated by cuSPARSE.

Field programmable gate array (FPGA), as an instance of domain-specific hardware, emerged as a promising alternative for DNN accelerations [59]. In addition to the high energy efficiency, its reconfigurability enables the customization of hardware function and organization, adapting to various resource and data usage requirements. Prior FPGA explorations have comprehensively studied *dense* CNN models [60][61][62]. However, these optimization techniques cannot be directly applied to *sparse* models as the reconfigurable capabilities of FPGAs cannot be fully leveraged to maximize the overall system throughput, especially for Conv layers, due to the following two difficulties: 1) Poor data locality inherited from compressed kernel weights. The irregular data access pattern destroys the data streaming, complexing the resource allocation and optimization. Our experiment shows that the execution of sparse CNNs greatly offsets from the theoretical expectation and sometimes incurs performance degradation. 2) Lacking of attention on the sparsity of feature maps. Prior network compression methods mainly focus on kernel weights, but seldom seek solutions for intermediate feature maps. We observed that as the weight compression ratio increases, the uncompressed feature map further increases the requirement of memory and computation resources.

4.2 CNN ACCELERATION AND DIFFICULTY

4.2.1 Dense CNN Acceleration

A CNN model involves intensive convolution operations between pre-trained kernel weights and feature maps, thus takes most of the computation time on CNN inference. Early CNN accelerator designs optimize computation engines and explore different parallelism opportunities, such as the data-level parallelism within feature maps and convolution kernels [63], the "inter-output" and "intra-output" parallelism [64]. These techniques reduce the total communication traffic but ignore the data reuse patterns. So it is hard to generalize these methods to diverse networks and layer types. Another popular approach is the memory-centric accelerators that exploit the data access pattern of convolutional kernels [60]. These designs adopt a large on-chip memory and rely on continuous data streaming between memory and computation units for high throughput, which could be ruined by model compression. Quantization that decreases the degree of redundancy of model weights has been investigated to reduce storage requirement. As an extreme case, binary neural networks (BNNs) constrain some or all the arithmetic to single-bit values [65][66]. So the convolution operations that require the highest computation cost can be executed by bitwise operation kernels. However, BNNs have not demonstrated state-of-the-art recognition accuracies on large scale ImgeNet dataset.

4.2.2 Inefficient Acceleration of Sparse CNN

Imposing sparsity through network pruning [57] or regularization [56] can inhibit the knowledge from big and cumbersome networks meanwhile effectively shrinking network scale, reducing computational cost and alleviating bandwidth pressure. Minerva [67] developed by Reagen *et al.* exploits zero valued neurons on MLP for memory compression and power reduction, but not for computation speed-up. Han *et al.* [57] showed that with minimal or even no loss in accuracy, a large portion of weights can be pruned: the number of parameters of AlexNet or VGG-16 reduces $9\times$ or $13\times$, respectively.

It is worthwhile to mention that this technique reduces only the parameters of FC layers, resulting in $3 \sim 4\times$ layer-wise speedup for FC layers. No practical speedup on Conv layers was obtained, even though convolution operations consume more than 90% of total computation time. As latest CNN models adopt even fewer FC layers and therefore less arithmetic operations [58], *the efficient acceleration of convolutional layers becomes more important and shall be considered in CNN accelerator design.*

We note that convolution with sparse kernel does not necessarily improve the performance on general-purpose computation platforms, due to the lack of dedicated hardware support. For example, we tested the practical speedup of an AlexNet by adopting the widely used L1-norm

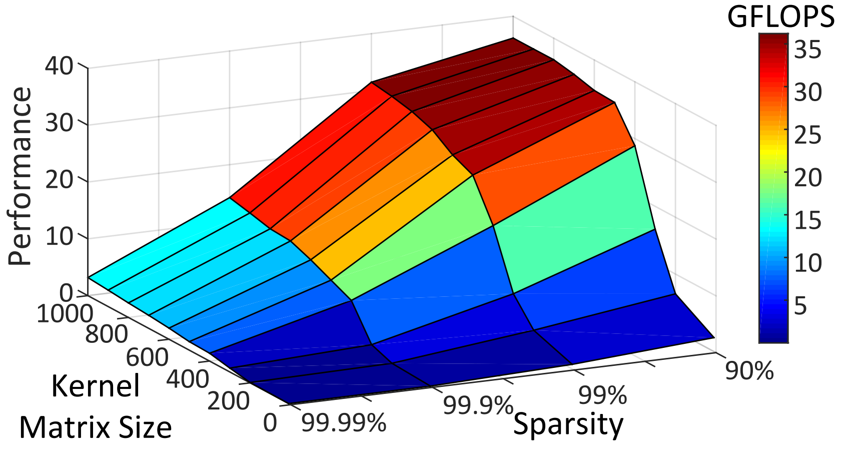


Figure 22: The impact of kernel matrix sparsity on convolution performance.

regularization [54] and controlling the accuracy loss within 2% from the original dense model. Figure 21 shows the performance gains of sparse Conv layers over the dense counterparts on multiple GPU platforms. The layer-wise speedups are all ¡1.5$\times$, even applying state-of-the-art sparse library cuSPARSE [37]. Moreover, performance degradation is observed for some layers with high sparsity[1].

4.3 THE PROPOSED DESIGN FRAMEWORK

When data from main memory is stored contiguously, the memory efficiency is maximized as every dataset is used repeatedly in a short period of time and evicted afterwards without further reference. Implementing sparse CNN model faces with the opposite situation of irregular data accesses throughout memory, causing execution inefficacy. Figure 22 shows our evaluation on compressed sparse row (CSR) [68] based sparse convolution implemented within Caffe framework [58], by varying the kernel matrix size and sparsity. As the kernel sparsity increases from 90% to 99.99%, the performance measured by GFLOPS drops rapidly as the data movement of the compressed kernel matrix and feature maps emerges as the major performance bottleneck.

To improve the data-locality of sparse kernel weights, we first analyze the performance modeling of FPGA-based platform. The estimation starts with the lower-bound execution time of a sparse Conv layer on an ideal architecture that has unbounded amount of hardware resources. Initially, the kernel weights (W) and the input feature map (iF) are stored in an external memory. The output feature map (oF) will be shipped out of FPGA after completing the computation. Note that the following analysis on Conv layers also applies to FC layers as an input feature map of FC layer is a simplified feature vector.

Assume nz_W and nz_{iF} are the numbers of nonzero elements in the kernel weight and input feature map, respectively. Most algorithms and storage formats only operate on nonzero elements for high efficiency. A set of floating-point/fixed-point operations including one addition and one multiplication are required for each nonzero element. The computation time required by a sparse convolution $T_{comp} = 2 \cdot nz_W \cdot nz_{iF}/P$, where P denotes the number of operation sets that can

[1]*Sparsity* is denoted as the number of zero elements divided by the total number of elements.

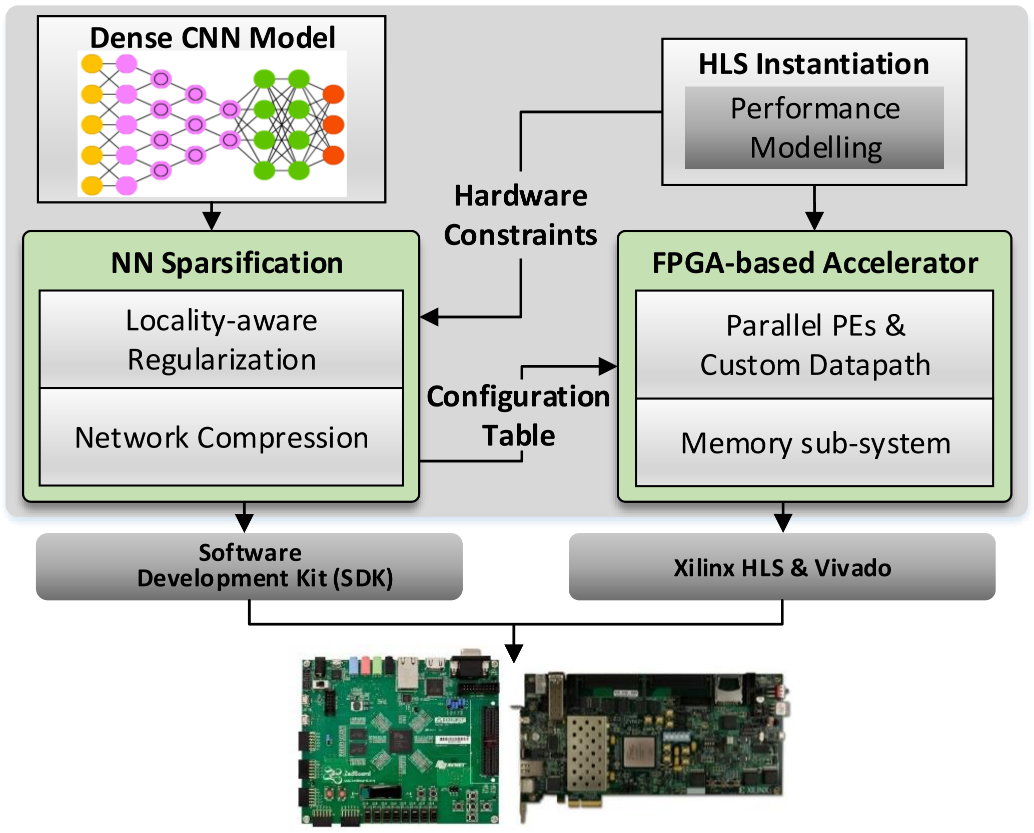


Figure 23: Our proposed SW/HW co-design framework for CNN sparsification & acceleration on FPGA.

be completed in one second. In order to keep track of each nonzero element, indices are required in compressed storage formats and needed to moved into FPGA's local memories at runtime. Let n_{index} represent the generated indices in total, then the I/O requirement $n_{IO} \propto nz_W + nz_{iF} + n_{index}$. The total memory I/O time $T_{IO} = n_{IO}/B$, where B is the available memory bandwidth. We consider the time used to configure FPGA and perload network topology information as T_{init}. Most importantly, due to the random sparse structure of the input data, $T_{overhead}$ is used to represent the latency brought by the irregular data access pattern. The total execution time is:

$$T = T_{overhead} + max(T_{comp}, T_{IO}) + T_{init}. \tag{4.1}$$

The reduction of T in this work is therefore guided by three principles: (a) eliminating the latency overheads by improving the data-locality of sparse kernel weights, (b) increasing P to enable high parallelism while keeping hardware complexity low, and (c) maximizing the effective memory bandwidth through network compressing with minimal index overheads.

Unlike prior acceleration frameworks that use hand-optimized templates to generate accelerator implementation for pre-trained dense CNN [69][70], the object of this work is to establish a sparse CNN design framework that can be adapted to various FPGA-based platforms. As Conv and FC layers respectively dominate the computation requirement and model size, we impose sparsity to both types of layers to optimize CNN model. The hardware constrains including the available computation resource and memory bandwidth are used to guide the sparsification at the training process, and therefore a close coordination across the software and hardware domains is realized.

Figure 23 illustrates the proposed design framework with the following features: (1) Given a dense CNN model, the locality-aware regularization selectively removes kernel weights in a hardware-friendly manner; (2) New sparse model is generated by taking consideration of hardware constraints and the kernel weights are compressed with minimal indexing overhead; (3) An explicit partitioning and distribution strategy is used to map the compressed kernel weights to hardware representation; (4) The parallel execution on processing elements (PEs) and memory sub-system promise the effectiveness of the sparse CNN acceleration. The details of the framework, including the model sparsification at the software level, the hardware architecture, and the optimization strategy shall be described respectively in Sections 4.4, 4.5, and 4.6.

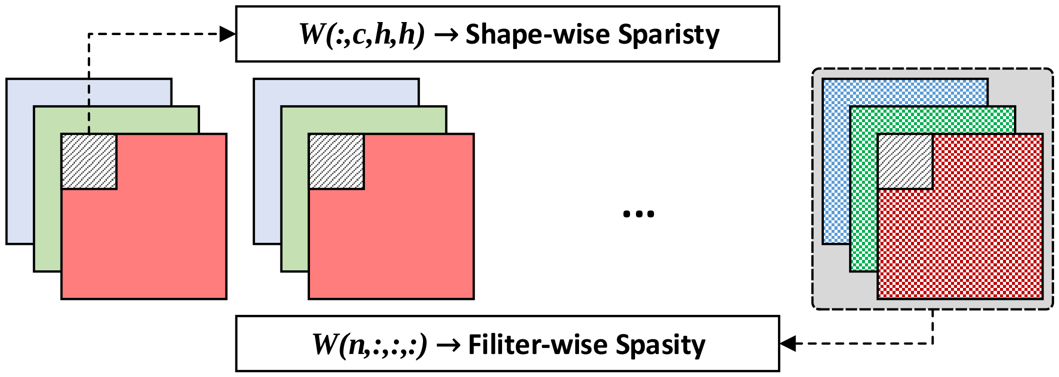

W(:,c,h,h) → Shape-wise Sparisty
...
W(n,:,:,:) → Filiter-wise Spasity

Figure 24: Kernel weights are split into pre-defined groups. A compact kernel is obtained through the locality-aware regularization.

4.4 CNN MODEL SPARSIFICATION

This section explains how to generate a sparse CNN model under our framework, through three key techniques—locality-aware regularization, sparse network representation, and kernel compression and distribution. Our work considers both Conv and FC layers. The description here focuses on Conv layers only as the sparsification of FC layers has been extensively studied.

4.4.1 Locality-aware Regularization

Considering that when all the non-zero parameters are gathered and placed within a compact space, the latency overhead during memory access can be greatly mitigated. Inspired by the fact that redundant connections exist across filters and within each filter [55][56][71], we adopt the *group lasso* regularization to prune weights of dense CNN model by groups. In this way, the data-locality is determined by the way of splitting groups. As illustrated in Figure 24, *filter-wise* and *shape-wise* sparsification that respectively remove a full 3-D kernel and the weights at the same location of each kernel are formulated.

Assume the kernel weights of each convolutional layer is formed as $W_{(n,c,h,h)}$, which is a bank of N filters across C input channels. The size of each feature is H. By applying sparsity regularization, the training optimization target is defined as:

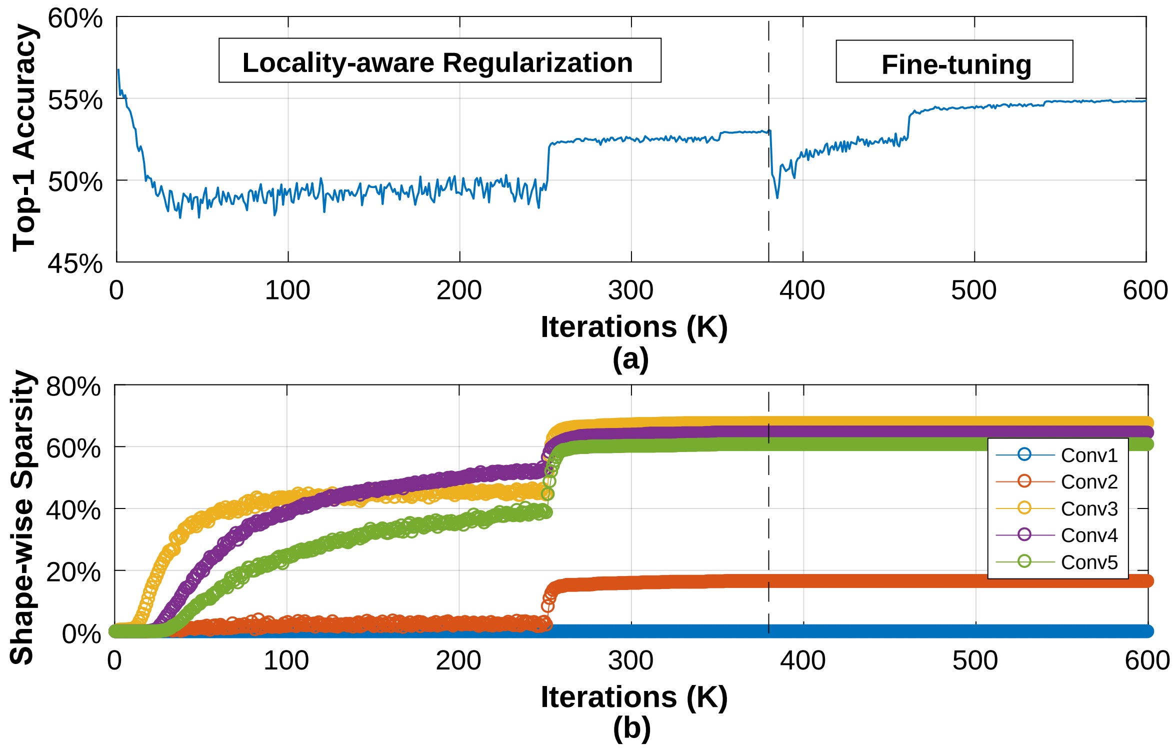


Figure 25: The locality-aware regularization first imposes sparsity on a pre-trained dense AlexNet, fine-tuning is applied to retain accuracy.

$$E(W) = E_D + \lambda_g \cdot R_g(W_{(n,c,h,h)}), \tag{4.2}$$

where E_D denotes the data loss, $R_g(\cdot)$ is the *group lasso* that zeros out the weights in specific groups. λ_g is the regularization constraint. Suppose $W_{(n,:,:,:)}$ is the n^{th} filter and $W_{(:,c,h,h)}$ are the weights located in the 2-D filter across the c^{th} channel. Applying *group lasso* to $W_{(n,:,:,:)}$ and $W_{(:,c,h,h)}$ leads to filter-wise and shape-wise sparsification, respectively. The regularizer in Eq. (4.2) becomes $\lambda_{g\_FC} \cdot R_g(W_{(n,:,:,:)}) + \lambda_{g\_Conv} \cdot R_g(W_{(:,c,h,h)})$.

Figure 25 shows the effectiveness of locality-aware regularization when applying AlexNet on ImageNet [72]. The AlexNet is first trained by following Eq. (4.2); the groups with all zeros are removed once the training is converged; at the end, the network is fine-tuned to regain the accuracy. Table 13 summarizes the average filter-wise and shape-wide sparsity of three representative CNN models. The results show that the locality-aware regularization is able to realize both shape-wise and filter-wise sparsification. For AlexNet, the average shape-wise sparsity of all Conv layers is 25.3% without sacrificing accuracy. By sacrificing less than 2% accuracy loss, the shape-wise and filter-wise sparsity increases to 41.9% and 19.4%, respectively.

4.4.2 Sparse Network Representation

The convolution of feature maps and kernel weights involves intensive 3-D multiply and accumulate (MAC) operations. Traditional method that represents a Conv layer with a stack of 2-D images

Table 13: The average weight sparsity and accuracy of three selected CNN models after regularization.

Model	**Layers**	**Dataset**	**Shape Spasity**	**Filter Sparsity**	**Top-1 Accuracy**	**Accuracy Loss**
ConvNet	4	Cifar-10	27.3%	21.7%	82.1%	0%
AlexNet	8	ImageNet	25.3%	5.4%	56.8%	0%
AlexNet	8	ImageNet	41.9%	19.4%	54.9%	1.9%
VGG-16	16	ImageNet	68.5%	3.5%	65.5%	2.8%

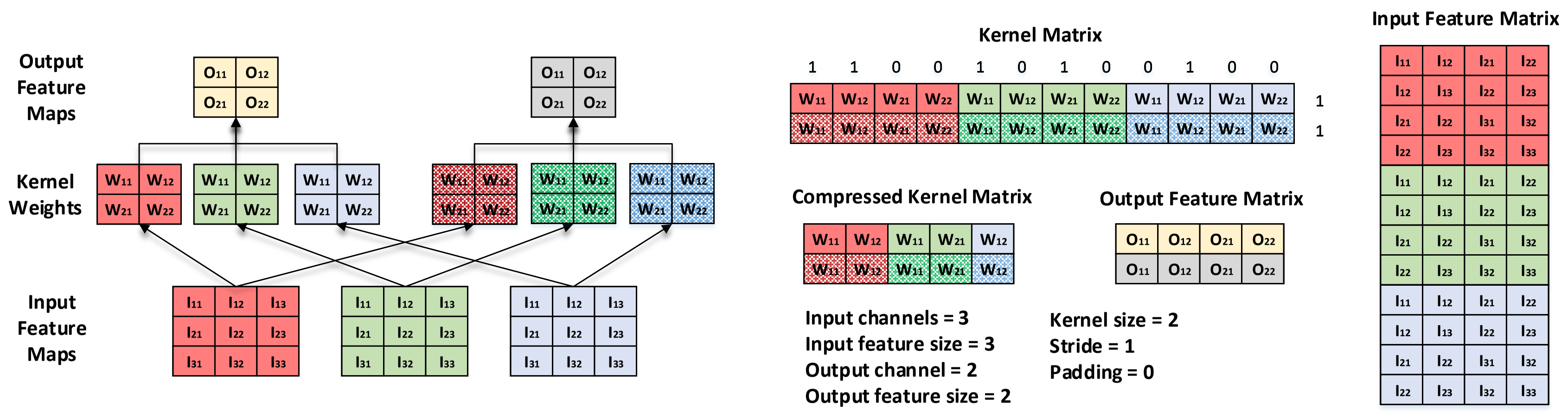


Figure 26: An illustration of our proposed sparse network representation for sparse CNN acceleration.

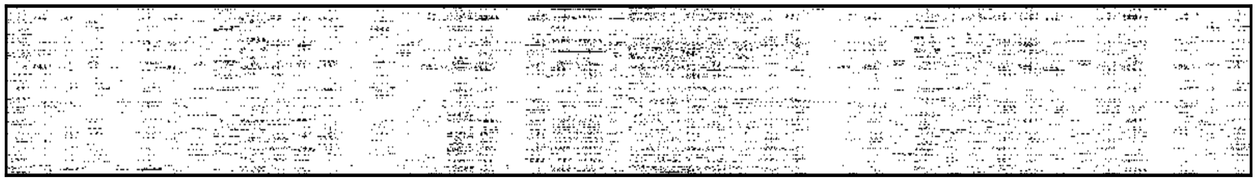
is not efficient for sparse CNN due to the aforementioned irregular data pattern. In this work, we propose to address the problem by reorganizing the input data and mapping the 3-D convolutions to matrix multiplication operations. Both of the kernel weights and feature maps can be represent as 2-D matrices with the following advantages:

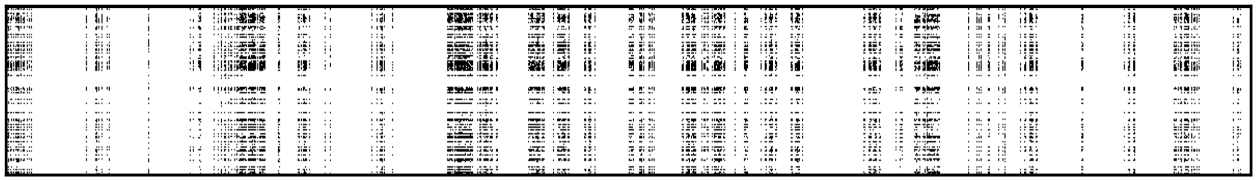
1) Data-locality is well preserved when accessing sparse kernel weights. Figure 26 illustrates the adopted network representation. 3D filter $W_{(n,:,:,:)}$ is reorganized to a row in the kernel matrix where each column is a collection of weights $W_{(:,c,u,v)}$. The filter-wise and shape-wise sparsity can directly map to the zero rows and columns.
2) The matrix representation provides a uniformed data layout and thus can be easily adopted to different Conv layers with various input feature/kernel sizes or sliding strides. Each output feature then corresponds to a column in the new kernel matrix which can be directly used for normalization or pooling. However, such a representation comes with data replication as shown in Figure 26. We develop a data reuse scheme to address the issue (see Section 4.5.4).
3) By adopting the proposed network representation, a matrix multiplication-based accelerator can handle the operation on both Conv layers and FC layers. Compared with traditional architectures, less computation resources are required to process FC layers.

4.4.3 Kernel Compression and Distribution

Kernel compression. Compression is widely used for sparse matrix storage. For example, by keeping the relevant matrix information with additional indices to trace non-zero elements, CSR [68] can reduce the memory requirement of a kernel matrix with n_r rows and n_c columns from $\mathcal{O}(n_r \times n_c)$ to $\mathcal{O}(2n_{nz} + n_r)$, where $n_{nz} = (1 - sparsity) \times n_r \times n_c$ denotes the number of non-zeros. CSR is effective for highly sparse matrices. However, for a kernel matrix with low sparsity, such as Conv1 and Conv2, the storage requirement of CSR-based compression is similar to the uncompressed version. Even worse, the CSR-based matrix multiplication incurs much higher hardware complexity.

Instead, we propose to apply a low-cost compression scheme to kernel matrices obtained from the locality-aware regularization. As shown in Figure 26, a binary string is used to indicate the status of the rows/columns, i.e., “0” represents an all-zero row/column while “1” denotes one with dense data. Figure 27 visualizes the data layout of Conv3 layer in AlexNet. The compressed kernel


(a) Random pruning by L1 regularization


(b) Locality-aware regularization

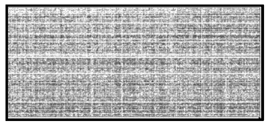

(c) Sparse convolutional layer compression

Figure 27: Locality-aware regularization on Conv layer.

matrix in Figure 27(c) is generated from Figure 27(b) that applies the locality-aware regularization. Utilizing the random pruning by L1-norm regularization [54], however, produces an irregular data layout as shown in Figure 27(a), which requires heavy indexing for compression.

We compare the execution performance of our approach and the CSR-based compression on matrices obtained by applying the locality-aware regularization on AlexNet. Figure 28 shows the performance variance when increasing the number of cores deployed in Intel Xeon E5-2630 CPU. Our compression scheme shows a strong scaling feature—the performance measured by GFLOPS increases linearly till the memory bandwidth gets saturated. The performance of our compression is approximately 1.25× over that of the CSR-based implementation.

Matrix blocking and distribution. We partition the compressed kernel matrix into sub-blocks to enable parallel matrix multiplication. Each sub-block, with a size of $S_c \times S_c$, is processed independently by a PE regardless of the size of the original kernel matrix (see Section 4.5). As illustrated in Figure 26, the matrix multiplication is accomplished by repeatedly sliding a $S_c \times S_c$ window column-wise in the compressed kernel matrix and row-wise in the input feature matrix, resulting in $S_c \times S_c$ elements in the output feature matrix.

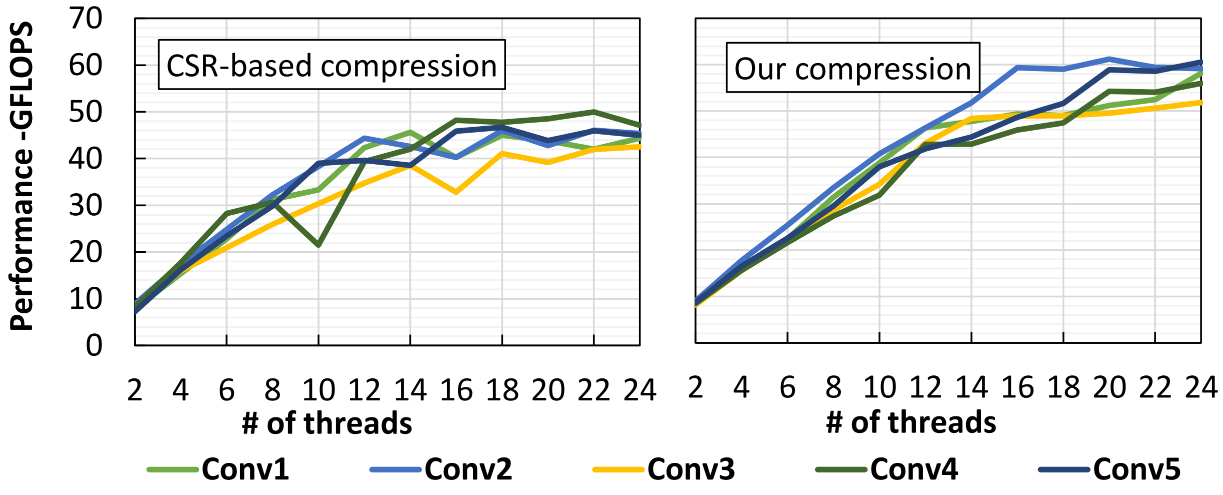


Figure 28: Our string-based compression balances computation and memory, showing a strong scalability.

When the dimension of an original input matrix is not a multiple of S_c, zero padding is needed for the matrices of convolution layers. Suppose that there are S_p PEs instantiated in the system and each PE performs S_c parallel MAC operations. Increasing the size of sub-blocks helps improve throughput as it fetches a larger number of inputs to the local memory and performs computations without waiting for external data. However, the execution time could be prolonged if the zero-padding is excessive. An appropriate combination of S_c and S_p to maximize the overall system throughput needs to take the hardware constraints into the consideration. The related discussion shall be presented in Section 4.6.2.

4.5 HARDWARE IMPLEMENTATION

4.5.1 The System Architecture

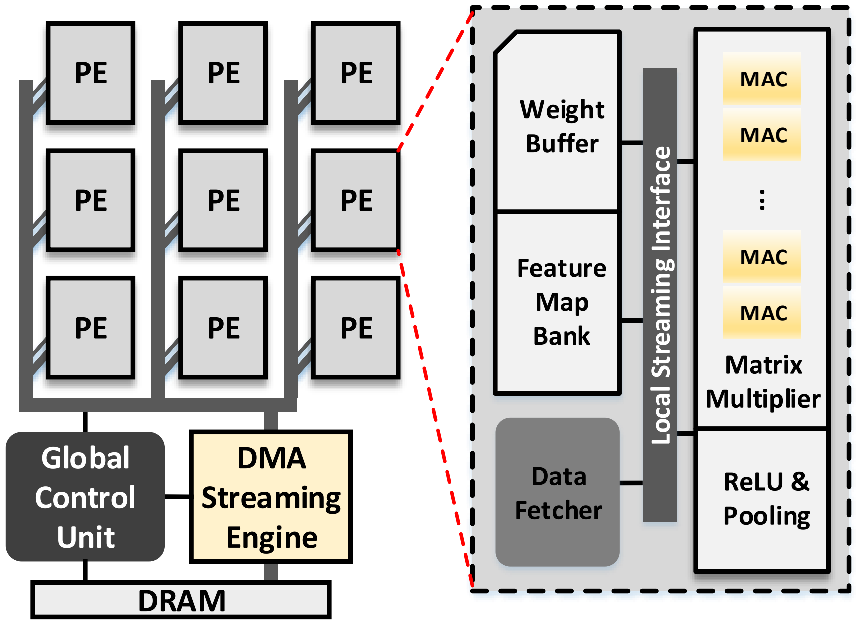


Figure 29: The system architecture overview of the FPGA-based sparse CNN accelerator.

Figure 29 gives an overview of the proposed system architecture designed to implement sparse CNNs effectively. The design is deployed on a single FPGA and uses DRAM as external storage. A systolic array of uniformed PEs are the main computation power of the accelerator. The global control unit initialize the accelerator and distribute kernel weights and feature maps to PEs at runtime. The data from/to the external memory is handled by a multi-port DMA streaming engine. The optimal number of PEs is determined by the available hardware resource and memory bandwidth of the FPGA board.

Each PE takes a subset of the overall computation by following the distribution strategy explained in Section 4.4.3. The PE controller sets up registers according to the received configuration instructions, then enables *Data Fetcher* to load vector arrays of an input feature map into *Feature Map Bank* at runtime. When streaming a sub-block of the compressed kernel matrix, *Weight Buffer* insures the continuity of DMA service. The PE integrates *ReLU* activation and *Pooling* function.

4.5.2 The PE Optimization

To obtain the full computational throughput, we propose three techniques to enhance the pipeline structure of PE and minimize the latency overhead.

Removing the carried dependency. As shown in Algorithm **??**, a matrix multiplication is usually implemented with 3 nested loops. The inner-most loop *Product* performs MAC operation where each iteration takes 2 clock cycles. When pipelining the nested loops, Vivado HLS automatically applies the loop flattening – collapsing the nested loops, removing the loop transitions and mapping arrays w, iFM and oFM into block RAMs (BRAMs). Loop *Product* cannot achieve 1-cycle pipeline interval due to the carried dependency—a dependency between an operation in

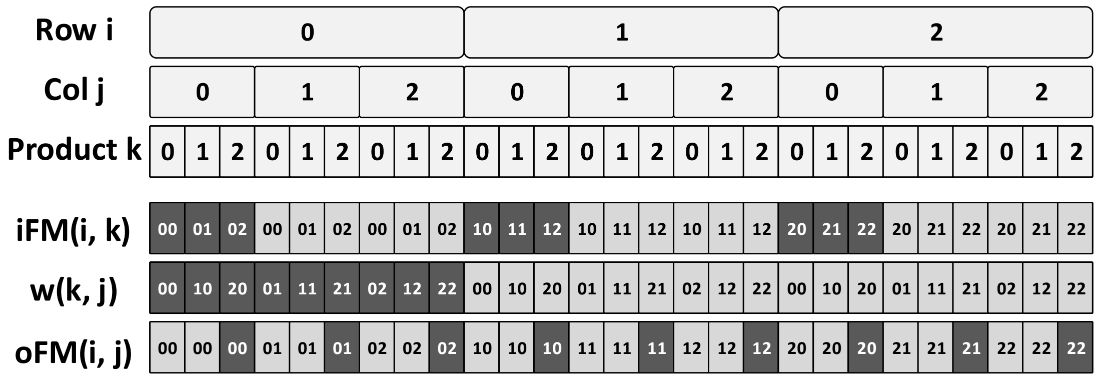


Figure 30: The address access pattern during matrix multiplication within one PE.

one iteration of a loop and another operation in the following iteration of the same loop. During pipelined MACs, the write to BRAM port for oFM in the first iteration is still on-going when the second iteration tends to apply another address for a read operation. Since the two requests are located to different addresses, they cannot be applied to a BRAM simultaneously. In this loop, we use a temporary variable (tmp) for the accumulation. The BRAM port is only be written when the final result is computed and therefore the carried dependency in loop *Product* is removed.

Improve data parallelism. We unroll inter-loop Col so that S_c rows of a column from w can be processed at a time. A dual-port BRAM provides up to two ports. So accessing w or iFM through a single BRAM cannot read all values in one clock cycle due to the lack of sufficient ports. As the loop index for loop *Product* is k, both matrices should be partitioned along their respective dimensions. More specifically, w is on column-wise because its access patterns is $w(k, j)$ while iFM is along row-wise. Partitioning a matrix creates S_c arrays, meaning S_c BRAM ports for the unrolled loop Col.

A data streaming interface. To enable a streaming interface, data must be accessed in a sequential order. Figure 30 illustrates the I/O access pattern of the matrix multiplication in our design, assuming $S_c = 3$. The addresses to read w and iFM and write to oFM change as variables i, j and k iterate. In our design, only those ports in dark color will be accessed by following the illustrated sequence. The streaming data are cached internally to avoid repeated reading and the computation result of oFM is streamed out in a continuous pace.

Table 14: The average sparsity and replication rate of the input feature maps of Conv layers in AlexNet.

Layer	**Conv1**	**Conv2**	**Conv3**	**Conv4**	**Conv5**
Sparsity	0%	18.9%	25.1%	35.3%	32.4%
Replication Rate	$7.2\times$	$6.1\times$	$2.1\times$	$9.0\times$	$9.0\times$

4.5.3 Zero Skipping for Computation Efficiency

The matrix representation of convolution operations facilitates not only the static sparsity of kernel weights, but also the dynamic sparsity of input feature maps. Table 14 shows the sparsity of the input feature maps propagated along Conv layers in AlexNet. Since the sparsity structure varies with different images, the result is averaged over 50,000 validating images from ImageNet. Without applying any sparsification technique, the feature maps already demonstrate sparsity and it increases with the depth of layers. This is because the kernel in the first layer are essentially the detectors of edges and lines obtained from a dense image in real-world scenes. As networks going deeper, the concepts represented by feature maps become more abstract—from edges and lines, to shapes and object parts that demonstrate higher sparsity.

The dynamic sparsity of feature maps inspires the thought of skipping the computation of those non-zeros in w associated with zero elements from iFM. Albericio *et al.* proposed to index each non-zero with an offset for this purpose [73]. Similar to the compression of kernel weights, augmenting indices increase the on-chip storage requirement. We proposed a dynamic hardware approach which actively skips over zero parameters in the array of iFM, by adding a conditional statement in line 10 of Algorithm. As w is streamed in and iFM is loaded into *Feature Map Bank* in a PE, the matrix multiplier selectively pulls the pair out of their respective BRAMs and conducts multiplication. During synthesis, we found this conditional statement only incur an initial latency of 4 $\sim$ 5 clock cycles, which can be amortized by the data streaming.

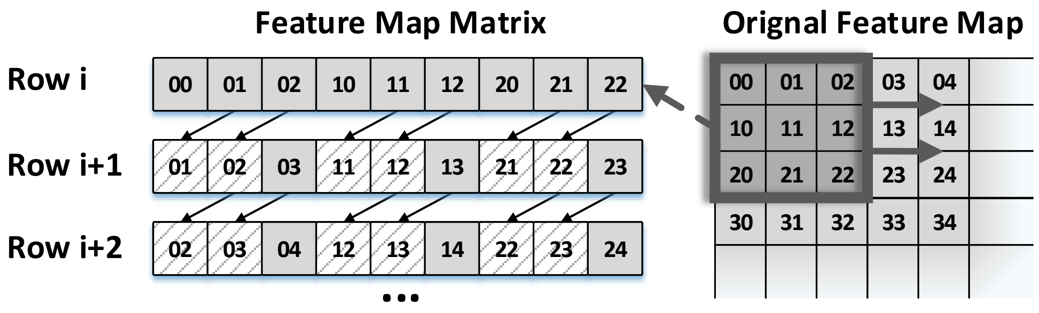


Figure 31: The data reuse pattern of feature map.

4.5.4 Data Reuse to Improve Effective Bandwidth

Converting 3-D convolution down to 2-D matrix multiplication (see Section 4.4.2) is effective for sparse CNN accelerations. The data reorganizing process will not incur replication of kernel weights and is able to keep the number of computation the same as traditional convolution. However, additional storage for feature maps increases the number of off-chip memory accesses. In Table 14, our exploration on AlexNet shows that the unfolding of a feature map introduces $2 \sim 9\times$ additional data.

We analyze the data access pattern of iFM. Figure 31 depicts an example when the kernel size is 3 and the stride equals to 1. As the filter kernel slides and performs the inner-products over a feature map, row $i + 1$ of iFM is formed by shifting row i forward and updating three nearest elements. *Data Fetcher* is designed to load elements of iFM according to the data access pattern. This is a logic unit that implements the mapping process. The input is the location of elements in iFM updated by *Feature Map Bank*. The output is the corresponding address in the external memory. All the parameters to compute the address are set by *Gloable Control Unit*, including the kernel/feature size, channel number and the address of the first feature map. The addresses are generated as a stream after a latency of 24 clock cycles. Thus, iFM is reorganized on chip, off-chip memory only stores the data in the form of original input feature maps.

As the computation unit is driven by the operands streaming from *Feature Map Bank* and *Weight Buffer*, *Data Fetcher* is placed between *Feather Map Bank* and the system bus. We use the double buffering technique to overlap the fetching and computing. When *Data Fetcher* loads a block from one feature map, the next prefetch is being loaded into the other pre-prefetch buffer. As the computation unit is accessing one block of *Feature Map Bank*, *Data Fetcher* continues its loading to the other pre-fetching buffer.

4.6 HARDWARE SPECIFIC OPTIMIZATION

This section discusses the design trade-offs when mapping sparse CNNs to specific FPGAs. The proposed optimization techniques take consideration of the hardware constrains and sparsity structure of each layer.

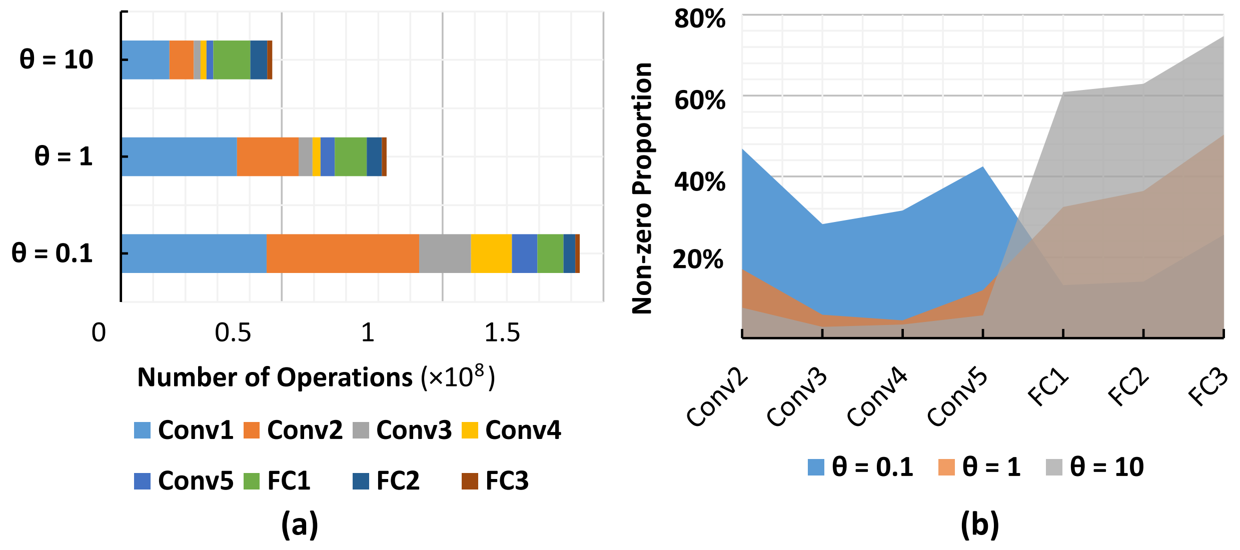


Figure 32: The trade-off between computation requirement (a) and model size (b) under different sparse regularization on Conv and FC layers of AlexNet. Conv 1 with low sparsity is omitted in (b).

4.6.1 Design Trade-offs on Cross Layer Sparsity

As Conv and FC layers respectively dominate the computation and model size, controlling the sparsity of these two types of layers is critical in performing trade-offs between classification speed and storage requirement. When targeting similar accuracy loss from sparse models, we find that applying the same regularization constraints across all the layers usually leads to a higher sparsity on FC layers but lower sparsity on Conv layers. In other words, the convolution computation is the main bottleneck. A small increase in parameter number of FC layers helps speedup Conv layers and optimize the speed/size of sparse CNN implementations. This trade-off can be obtained by applying different regularization constraints to Conv (λ_{g_Conv}) and FC (λ_{g_FC}) layers in Eq. (4.2).

Figure 32 shows how the balancing affects the trade-off between speedup (a) and model size (b) on AlexNet. In the experiment, we control the balance by setting λ_{g_Conv} and λ_{g_FC} respectively. The proposed optimization is determined by $\theta = \lambda_{g\_Conv}/\lambda_{g\_FC}$. For instance, $\theta > 1$ indicates stronger regularization on Conv layers than FC layers. The trained sparse model will be suitable for the performance-oriented FPGA implementation. Setting $\theta < 1$ leads to a sparse model for the storage-limited implementation.

4.6.2 Hardware Constraints

Ideally, the computation of a sparse CNN reaches its ultimate performance when the number of PEs $S_p \rightarrow \infty$ with a memory hierarchy that is able to feed data at any time without delay. Although a large S_p is preferred theoretically, there is an optimal range of S_p, which is mainly determined by the available on-chip computation resource and memory bandwidth.

The number of MACs. As exclusive compute elements in performing matrix multiplication, MACs are mainly implemented using dedicated DSP blocks. Assume r_{mac} units of resource are needed to construct a single MAC and S_c MACs for the inner parallelism of a PE, the number of PE is limited by $\frac{R}{r_{mac} \cdot S_c}$ if R DSPs are available on chip.

The effective BRAM space. Although FPGAs provide a certain amount of BRAM (e.g., 19.1Mb in Xilinx Zynq XC7Z045 used in this work), not all of them are available for user applications. Excluding the portion for interfacing to memory and other support functions, $\alpha \cdot M$ can be used for implementing *Weight Buffer* and *Feature Map Bank*. Since the compressed network is partitioned into many sub-matrices with a block size of m_{sub}, S_p cannot exceed $\frac{\alpha M}{m_{sub} \cdot DW}$, where DW denotes the data width.

The memory bandwidth to off-chip DRAM. Both of kernel weights and feature maps are initially stored in off-chip memory and streamed into PEs during computation. The achievable T_{comp} should not be less than T_{IO}, otherwise implementing more PEs will incur low computation effiency. According to the performance analysis in Section 4.3, S_p should be set within $\frac{N_{comp} \cdot B}{S_c \cdot N_{IO}}$.

These constraints from hardware platform shall be taken into consideration during the matrix partitioning. A reasonable number of PEs therefore is determined by

$$S_p = \left\lfloor min(\frac{R}{r_{mac} \cdot S_c}, \frac{\alpha M}{m_{sub} \cdot DW}, \frac{N_{comp} \cdot B}{S_c \cdot N_{IO}}) \right\rfloor. \tag{4.3}$$

4.7 EVALUATION

In this section, we present the experimental results of the proposed design framework for CNN sparsification and acceleration. A layer-by-layer evaluation is set to validate the performance speedup. We also compare our design to the well-tuned CPU/GPU implementations and state-of-the-art FPGA-based CNN accelerators.

4.7.1 Experimental Setup

Benchmarks. To demonstrate the generalization of the proposed design framework, we implement ConvNet [74] on Cifar10 [75], AlexNet [58] and VGG-16 [76] on ImageNet classification [74]. The CNN models are trained under the performance-oriented optimization goal by setting $\theta = 10$. Rather than conventional 32-bit floating-point format, 16-bit fixed-point format is used to represent weight data, the effectiveness of which has been validated by previous study [61].

Table 15: Configuration of different platforms

Platform	**NVIDIA Tesla K40c**	**Intel Xeon E5-2630 v3**	**Xilinx ZC706**	**Xilinx VC707**
Technology	28nm	22nm	28nm	28nm
Optimizations	cuSPARSE	MKL	HLS	HLS
# of Cores	2880	8	24$\times$16	24$\times$32
Mem. Bandwidth	288GB/s	59GB/s	4.2GB/s	12.8GB/s
Feq. (Hz)	745M	2.4G	150M	150M
Power (W)	235	85	8.9	13.5
Peak Perf. (GOPS)	4290	307.2	115.2	230.4
Pratical GOPS	536.1	68.7	71.2	131.2

FPGA setup. The CNN accelerator is designed with Vivado HLS 2016.4. This tool initializes the implementation with C language and then exports the RTL as an IP core. Fast C/RTL co-simulation is used for design space exploration and performance estimation. After the placement and routing with Vivado 2016.4, the chip operates at 150 MHz. Table 16 summarizes the resource utilization of our implementation on Xilinx ZC706 (Zynq XC7Z045 w/ 900 DSPs) and VC707 (Virtex 485T w/ 2800 DSPs).

CPU/GPU setup. The software implementation runs with Caffe framework [74]. To adapt to sparse CNNs, the evaluation is optimized with the off-the-shelf libraries, i.e., MKL/cuSPARSE on CPU/GPU, respectively. The configuration details are summarized in Table 15.

Table 16: Resource utilization on FPGA

	FF	**LUT**	**DSP48E**	**BRAM**
ZC706	116,902 (26%)	68,446 (83%)	774 (86%)	498 (91%)
VC707	289,940 (47%)	104,274(34%)	1572 (56%)	795 (75%)

4.7.2 Layer-by-Layer Performance

Accelerating Conv layers is the key to an efficient sparse CNN implementation. So we first evaluate the performance of Conv1 $\sim$ Conv5 in the representative AlexNet and report the results in Table 17. The sparse kernel is compressed according to shape-wise and filter-wise sparsity. Based on the performance and resource utilization model in Section 4.6.2, we implement 16 and 32 PEs on the two FPGA platforms, respectively. Each PE executes 24 16-bit fixed-point MAC operations per cycle. The *practical GOPS performance* denotes the ratio of the total sparse operations over the entire runtime. The *computation efficiency* represents the ratio of the practical performance over the peak performance.

Across all the sparse Conv layers, our accelerator achieves an average performance of 148.7 GOPS with a computation efficiency of 64.5% on VC707. Slight performance decline on Conv3 and Conv4 is observed. This is because these two layers have higher sparsity and the compressed kernel matrix is too small to be distributed into multiple PEs. Averagely, compared to the well-tuned CPU and GPU implementations, our FPGA implementation on VC707 improves computation efficiency 1.8$\times$ and 4.7$\times$, respectively.

We also quantitatively analyze the performance gains of the proposed optimizations and report the results in Figure 33. The kernel compression leads to the most significant speedup—an average

Table 17: Performance evaluation on sparse Conv and FC Layers of AlexNet on ImageNet

Layer	Shape-wise Sparsity	Filter-wise Sparsity	# Ops - Dense $\times 10^7$	# Ops- Sparse $\times 10^7$	Pratical GOPS				Computation Efficiency			
					CPU	GPU	ZC706	VC707	CPU	GPU	ZC706	VC707
Conv1	0%	9.4%	21.1	19.1 (1.1×)	54.6	495.9	90.7	172.3	17.7%	11.5%	78.7%	74.7%
Conv2	63.2%	12.9%	44.7	14.3 (3.1×)	143.3	1070.4	88.6	168.3	46.6%	24.9%	76.9%	73.0%
Conv3	76.9%	40.6%	29.9	4.1 (7.2×)	114.8	400.1	68.2	129.5	37.3%	9.3%	59.0%	56.2%
Conv4	84.7%	46.9%	22.4	1.8 (12.3×)	146	583.6	69.4	131.8	47.5%	13.6%	60.2%	57.2%
Conv5	80.7%	0%	14.9	2.8 (5.1×)	97.3	389.1	74.8	142.1	61.3%	9.1%	64.9%	61.6%
Conv Total	**61.1%**	**21.9%**	**133.1**	**42.2 (3.2×)**	**111.2**	**587.8**	**78.3**	**148.7**	**36.2%**	**13.7%**	**67.9%**	**64.5%**
FC1	47.5%	9.7%	7.5	3.6 (2.1×)	13.8	45.4	20.1	33.8	4.4%	1.1%	17.4%	14.6%
FC2	43.5%	4.3%	3.3	1.8 (1.8×)	14.6	44.9	15.3	30.4	4.7%	1.0%	13.2%	13.1%
FC3	29.6%	3.3%	0.8	0.5 (1.4×)	11.4	29.2	16.5	28.8	3.7%	0.6%	14.3%	12.5%
FC Total	**40.2%**	**7.5%**	**11.7**	**5.9 (1.9×)**	**13.3**	**39.8**	**17.3**	**31.0**	**4.3%**	**0.9%**	**15.01%**	**13.4%**

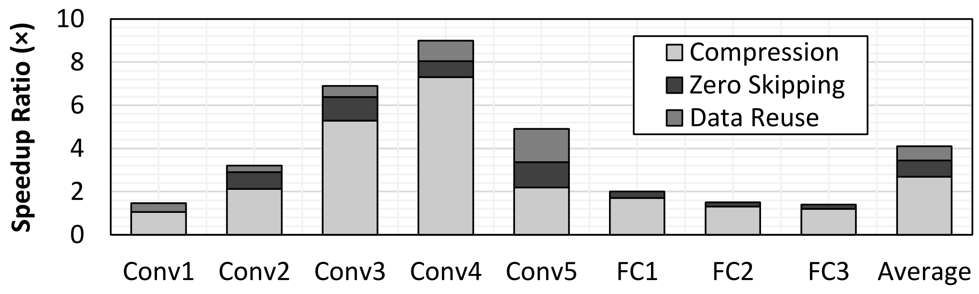


Figure 33: The performance is evaluated by applying the proposed optimizations and compared with the dense model. The sparse model is compressed first, then adds zero skipping and data fetcher, respectively.

2.7× over the dense model. After considering the dynamic sparsity of feature maps, the zero skipping technique improves the speedup to 3.4×. When the data replication rate of feature map is high, such as Conv1 and Conv5, the performance bottleneck shift from computation to memory bandwidth. *Data Fetcher* helps to relieve the bandwidth pressure.

4.7.3 End-to-End System Integration

Table 15 presents the overall performance of sparse CNNs on various platforms. The throughput on our FPGA design is compared with Caffe running on CPU and GPU. The results show that the CPU implementation is inferior in both performance (68.7 GOPS) and energy efficiency (0.81 GOPS/W). Averagely, our accelerator design on VC707 achieves 1.9× speedup on the sparse model compared with the CPU implementation. Among all the platforms, the GPU implementation provides the best performance for its high clock frequency and large memory bandwidth. Our design provides the highest energy efficiency of 9.7 GOPS/W, 12.1× and 5.1× over CPU and GPU implementations, respectively.

To validate the effectiveness and scalability of our design framework, we test three CNNs models—ConvNet, AlexNet and VGG-16, the layer numbers of which increase from 4 to 16. The detailed layer-by-layer analysis in Figure 34 shows that the CNN models optimized by our design framework can always deliver performance gain over its dense counterpart. Our design framework

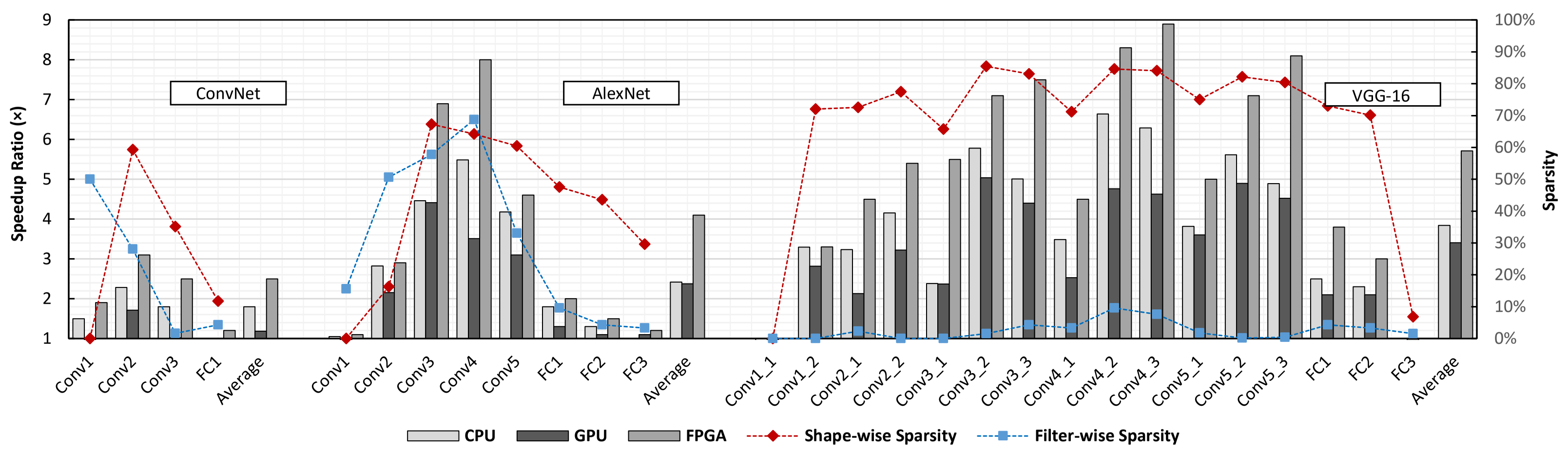


Figure 34: Accelerator performance on different network structures. The speedup ratio is evaluated by the runtime of sparse CNN on different platforms over its dense counterpart.

Table 18: Comparison to previous FPGA works

	FPGA-15 [60]	FPGA-16 [62]	This Work	FPGA-16 [61]	ICCAD-16 [70]	This Work
Model	**AlexNet**			**VGG-16**		
FPGA	Virtex7	Stratix V	Zynq	Zynq	Virtex7	Virtex7
Chip	485T	GSD8	XC7Z045	XC7Z045	690T	485T
Precision	32 float	16 fix	16 fix	16 fix	16 fix	16 fix
Top-1 Acc.	-	55.41%	54.84%	64.64%	-	64.82%
CNN Size	1.33	1.45	0.48	30.94	30.94	6.03
ms/Image	21.7	20.1	6.7	224.6	65.1	45.9
Practical GOPS	61.6	72.4	71.2 (215.1[§])	136.9	354	131.2 (673.1[§])

[§]The projected GOPS to the corresponding dense model.

preforms better on deeper networks (e.g., VGG-16) because small models tend to preserver more weights to maintain recognition accuracy. Compared to CPU/GPU implementations, our FPGA design obtains much higher speedups, benefiting from the proposed hardware innovation.

Table 18 compares our design with prior FPGA-based CNN accelerators for AlexNet and VGG-16. Our design takes advantages of the sparse structure of CNN models and achieves 2.6$\times$ speedup in classification runtime over [62] on AlexNet. Compared to [70], our implementation of VGG-16 runs 1.9$\times$ faster at a practical performance of 131.2 GOPS, which corresponds to 673.1 GOPS of a dense model.

4.8 CONCLUSIONS

In this work, we present an FPGA-based design framework for CNN sparsification and acceleration. The optimization is realized across the software-hardware boundary: first, the CNN model is sparsfied by taking consideration of the data-locality and then compressed to save the memory footprint. The hardware architecture is organized to well handle the compressed data format from the software level. The cross-layer optimization strategy is proposed to adapt the framework to different FPGA platforms. Working directly on the sparse model makes our design achieve efficient acceleration with minimal power dissipation.

5.0 RELATED WORK

Hardware accelerators with high parallelism and scalability are critical for deploying deep learning models with large data sets. As aforementioned in previous chapter, the basic operations of a DNN can be generally divided into two types: training and inferencing, which program the weights of the NN and perform the functions of the DNN, respectively.

GPU is the most popular computing platform for DNN applications. On GPU platforms, computations of connected layers are mapped to matrix operations; General Matrix-Matrix multiplication (GEMM) is serving as the core of DNN training and inferencing, in which are large matrices of input data and weights are multiplied. GPU processes data in SIMT (single-instruction multiple-threads) fashion by using centralized control of a large number of paralleled ALUs. The ALUs, however, fetch data from memory hierarchy but do not directly communicate with each other [77]. Very recently, the latest NVIDIA Volta GV100 architecture is equipped with Tensor Cores for deep learning matrix arithmetic. Each Tensor Core performs 64 floating point mixed-precision operations per clock (FP16 input multiply with full-precision product and FP32 accumulate, as shown in Figure 35) and 8 Tensor Cores in a streaming multiprocessor perform a total of 1024 floating point operations per clock. The introduction of Tensor Cores in Tesla V100 GPU boosts the performance of GEMM by more than $9\times$ compared to the previous Pascal-based GP100 GPU.

A common practice nowadays to speed up the DNN training on a single node GPU is using mini-batch stochastic gradient technique [78], which is known to be difficult to parallelize over multiple nodes. Hence, asynchronous stochastic gradient descent learning is proposed for large-scale GPU clusters [79], where multiple replicas of the gradients on different subsets of the training data are processed in parallel. Although each replica computes the gradients using the parameters that may not be immediately updated, such a scheme demonstrates a good tolerance to the errors generated in the asynchronous computations [80].

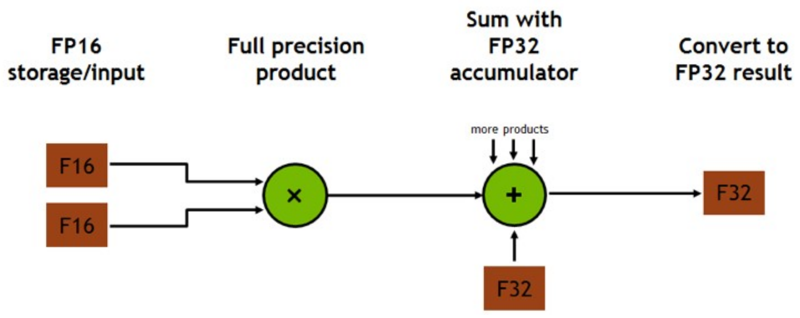


Figure 35: Volta GV100 Tensor Core operation.

Further improvements of computation and power efficiencies of DNN applications require the use of specialized circuits like ASIC (application-specific integrated circuit) accelerators [81] by paying the price of high non-recurring engineering cost, long design period, and less design flexibility. Most of such implementations focus on accelerating the operations of matrices and vectors: CNAPS [82] was designed in SIMD fashion with an array of 16×8 multipliers for matrix multiplications. Synapse-1 system [83] used systolic multiply-accumulators (MACs) with custom hardware to perform activation functions. The recent DianNao series focuses on optimizing memory access pattern in DNN applications and minimizing memory accesses to both on-chip memory and external DRAM with architectural support [84]: The original DianNao [2] implements an array of 64 16-bit integer MACs to map large DNN for computation acceleration (see Figure 36). However, due to the limited on-chip memory capacity, DRAM traffic of accessing weight parameters dominates the system energy consumption. DaDianNao [85] and ShiDianNao [86] eliminate the DRAM access by storing all weights on-chip on either eDRAM or SRAM.

Googles datacenters recently deployed Tensor Processing Unit (TPU) that accelerates the inference of neural networks [3]. The kernel of the TPU is a 65,536 8-bit MAC matrix multiply unit that provides a peak throughput of 92 TOPS and 28 MB software-managed on-chip memory. Figure 37 shows the block diagram of the TPU. Because the data access to memory consumes much more power than arithmetic units, the matrix unit uses systolic execution to save energy by reducing reads and writes of the Unified Buffer.

Enforcing sparsity through network pruning or regularization can extract the knowledge from

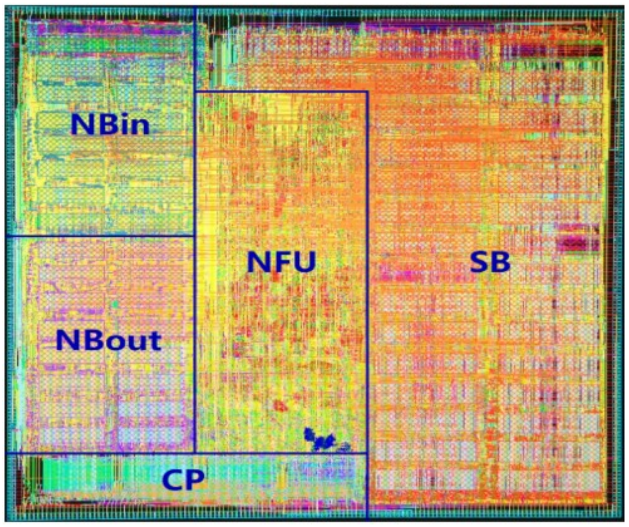


Figure 36: The layout of DianNao [2].

big and cumbersome networks meanwhile effectively shrinking the density and/or scale of the networks, reducing computational cost, and alleviating bandwidth limit. The EIE architecture [57] performs inference on compressed network model and accelerates the resulting sparse matrix-vector multiplication by weight sharing. With only 600mW power consumption, EIE can achieve 102 GOPS processing power on a compressed network that is equivalent to 3 TOPS/s on an uncompressed network. It translates to $24000\times$ and $3400\times$ energy efficiency of CPU and GPU, respectively [87].

A good balance between the high efficiency of ASICs and the generality of general-purpose microprocessors is programmable logic [88]. FPGA (field programmable gate array), for example, is a good candidate for DNN acceleration with necessary tailoring.

FPGA-based DNN accelerators exploit the computational concurrency with strong adaptability to the changes of weights and network topologies. Using the popular CNN model as an example, its FPGA-based accelerators can be categorized into two groups: the first group [63][89][64] focuses on optimizing computing engines. An early design uses systolic architecture to realize filtering convolution and has been used in some embedded systems for automotive robots. Two later designs explore the parallelism within feature maps and convolution kernel [63][89]. The latter one

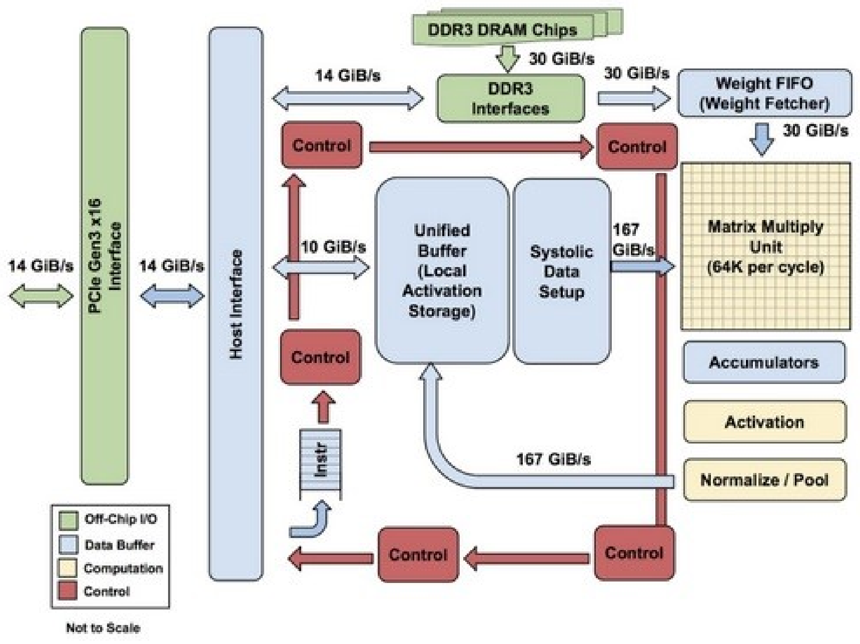


Figure 37: Block diagram of Googles Tensor Processing Unit [3].

[64] also leverages inter-output and intra-output parallelism with high bandwidth and dynamical configurations to improve the performance. The second group focuses on data communication limit and choose to maximize date reuse and minimize bandwidth requirement. Some designs need considerably long time (e.g., tens seconds) to prepare the FPGA for the computation of the next layer while other designs [60] only take less than a microsecond to configure a few registers. Computation cost and memory bandwidth consumption need to be balanced in practical FPGA implementations.

6.0 CONCLUSION AND FUTURE WORK

When addressing the hardware needs of deep neural networks, FPGAs provide an attractive alternative to GPUs and GPPs. In particular, the ability to exploit pipeline parallelism and achieve an efficient rate of power consumption give FPGAs a unique advantage over conventional hardware accelerators for common deep learning practices. As well, design tools have matured to a point where integrating FPGAs into popular deep learning frameworks is now possible. In this dissertation, we also demonstrate, by applying the proposed software-hardware co-design, FPGAs can effectively accommodate the trends of deep neural networks and provide architectural freedom for exploration and research.

The future of deep learning on FPGAs, and in general, is largely dependant on scalability. For these techniques to succeed on the problems of tomorrow, they must scale to accommodate data sizes and architectures that continue to grow and the deployment on hardware with different configurations. Another avenue for improving DNN efficiency is to use more compact data types. Many researchers have shown [61][90] that it is possible to represent data in much less than 32-bits, demonstrating the use of 8-4 bits (depending on the network) leads to only a small reduction in accuracy compared to full precision. Data types which are more compact than 32-bit single precision floating point are becoming the new norm. As an evidence of this, the latest GPUs are providing native support for FP16 and Int8 data types. Moreover, popular DNN frameworks, such as TensorFlow, provide support for such data types as well. Interestingly, very recently, research on binarized neural networks (BNNs) [91][92] investigates the use of 1-bit data types, by constraining values to +1 or -1. The most efficient variant of BNNs proposes using 1-bit for both neurons as well as weights. The brilliance of doing this is that not only is the storage size and bandwidth demand dramatically lower ($32\times$ smaller than FP32), but the computation of 1-bit multiply-accumulate can be done without multiplications or additions. BNNs have comparable accuracies to state-of-the-art full precision networks for small datasets (e.g., CIFAR10). However, the BNN accuracy on larger datasets (e.g. ImageNet) has not yet been realized. Nevertheless, BNN research is very active and rapidly advancing. Ternary neural networks (TNNs) [93] are another class of network that proposes extremely low bit-width. TNNs constrained weight values to 0, +1, or -1, which can be represented in 2 bits. Recently [94], TNNs have been shown to provide comparable accuracy on ImageNet, within 1% of full-precision ResNet-152, which is the latest ILSVRC winner. However, such TNNs still rely on FP32 neuron values. Thus, the multiply-accumulate computations are done between FP32 neurons and 2-bit weights.

The other trend is in optimizations using mathematical transforms. In particular, Winograd transformation [95] has been shown to be amenable to small DNN filters (e.g., 3×3) that are common in state-of-the-art DNNs. Fast Fourier Transforms (FFTs) have also been shown to be amenable for larger filters (5×5 and above), which are still used in some DNNs. FPGAs have been known to be an efficient platform for FFTs [96], and one could expect that they would be well-suited for Winograd transformations as well. These transforms are often computable in a streaming data fashion and involve an arbitrary set of mathematical operators. And, there are many possible transformation parameters that lead to different compositions of mathematical operators. Such computation properties (arbitrary composition of operations on streaming data) are likely to be amenable to FPGAs.

BIBLIOGRAPHY

[1] B. Li, E. Zhou, B. Huang, J. Duan, and Y. Wang, "Large scale recurrent neural network on gpu," in *IJCNN*, 2014, pp. 4062–4069.

[2] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," in *ACM Sigplan Notices*, vol. 49, no. 4. ACM, 2014, pp. 269–284.

[3] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *et al.*, "In-datacenter performance analysis of a tensor processing unit," *arXiv preprint arXiv:1704.04760*, 2017.

[4] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[5] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2014, pp. 580–587.

[6] R. Collobert and J. Weston, "A unified architecture for natural language processing: Deep neural networks with multitask learning," in *Proceedings of the 25th international conference on Machine learning*. ACM, 2008, pp. 160–167.

[7] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A.-r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath *et al.*, "Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 82–97, 2012.

[8] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[9] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2015, pp. 1–9.

[10] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *Proceedings of the 22nd ACM international conference on Multimedia*. ACM, 2014, pp. 675–678.

[11] C. Farabet, Y. LeCun, K. Kavukcuoglu, E. Culurciello, B. Martini, P. Akselrod, and S. Talay, "Large-scale fpga-based convolutional networks," *Scaling up Machine Learning: Parallel and Distributed Approaches*, pp. 399–419, 2011.

[12] S. K. Esser, A. Andreopoulos, R. Appuswamy, P. Datta, D. Barch, A. Amir, J. Arthur, A. Cassidy, M. Flickner, P. Merolla *et al.*, "Cognitive computing systems: Algorithms and applications for networks of neurosynaptic cores," in *Neural Networks (IJCNN), The 2013 International Joint Conference on*. IEEE, 2013, pp. 1–10.

[13] A. Amir, P. Datta, W. P. Risk, A. S. Cassidy, J. A. Kusnitz, S. K. Esser, A. Andreopoulos, T. M. Wong, M. Flickner, R. Alvarez-Icaza *et al.*, "Cognitive computing programming paradigm: a corelet language for composing networks of neurosynaptic cores," in *Neural Networks (IJCNN), The 2013 International Joint Conference on*. IEEE, 2013, pp. 1–10.

[14] S. A. McKee, "Reflections on the memory wall," in *Proceedings of the 1st conference on Computing frontiers*. ACM, 2004, p. 162.

[15] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cudnn: Efficient primitives for deep learning," *arXiv preprint arXiv:1410.0759*, 2014.

[16] N. Suda, V. Chandra, G. Dasika, A. Mohanty, Y. Ma, S. Vrudhula, J.-s. Seo, and Y. Cao, "Throughput-optimized opencl-based fpga accelerator for large-scale convolutional neural networks," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2016, pp. 16–25.

[17] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song *et al.*, "Going deeper with embedded fpga platform for convolutional neural network," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2016, pp. 26–35.

[18] P. A. Boncz, M. L. Kersten, and S. Manegold, "Breaking the memory wall in monetdb," *Communications of the ACM*, vol. 51, no. 12, pp. 77–85, 2008.

[19] M. Naylor, P. J. Fox, A. T. Markettos, and S. W. Moore, "Managing the fpga memory wall: Custom computing or vector processing?" in *2013 23rd International Conference on Field programmable Logic and Applications*. IEEE, 2013, pp. 1–6.

[20] A. Pauls and D. Klein, "Faster and smaller n-gram language models," in *Proc. ACL*, 2011, pp. 258–267.

[21] T. Mikolov, S. Kombrink, L. Burget, J. Cernocky, and S. Khudanpur, "Extensions of recurrent neural network language model," in *IEEE Conf. on ICASSP*, 2011, pp. 5528–5531.

[22] T. Mikolov, A. Deoras, D. Povey, L. Burget, and J. Cernocky, "Strategies for training large scale neural network language models," in *IEEE Workshop ASRU*, Dec 2011, pp. 196–201.

[23] H. Amin, K. Curtis, and B. Hayes-Gill, "Piecewise linear approximation applied to nonlinear function of a neural network," *Circuits, Devices and Systems, IEEE Proc.*, vol. 144, no. 6, pp. 313–317, 1997.

[24] D. L. Ly and P. Chow, "A high-performance fpga architecture for restricted boltzmann machines," in *Proc. FPGA*, 2009, pp. 73–82.

[25] *Convey Reference Manual*, Convey computer, 2012, http://www.conveycomputer.com/.

[26] G. Zweig and C. J. Burges, "A challenge set for advancing language modeling," in *Proc. NAACL-HLT 2012*, pp. 29–36.

[27] T. Mikolov, "Statistical language models based on neural networks," Ph.D. dissertation, Brno Uni. of Technology, 2012.

[28] A. Mnih and K. Kavukcuoglu, "Learning word embeddings efficiently with noise-contrastive estimation," in *Proc. NIPS*, 2013, pp. 2265–2273.

[29] M. Pietras, "Hardware conversion of neural networks simulation models for neural processing accelerator implemented as fpga-based soc," in *Proc. FPL*, 2014, pp. 1–4.

[30] P. Sonneveld and M. B. van Gijzen, "Idr (s): A family of simple and fast algorithms for solving large nonsymmetric systems of linear equations," *SIAM Journal on Scientific Computing*, vol. 31, no. 2, pp. 1035–1062, 2008.

[31] Ü. V. Çatalyürek, M. Deveci, K. Kaya, and B. Ucar, "Multithreaded clustering for multi-level hypergraph partitioning," in *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*. IEEE, 2012, pp. 848–859.

[32] J. Fowers, K. Ovtcharov, K. Strauss, E. S. Chung, and G. Stitt, "A high memory bandwidth fpga accelerator for sparse matrix-vector multiplication," in *Field-Programmable Custom Computing Machines (FCCM), 2014 IEEE 22nd Annual International Symposium on*. IEEE, 2014, pp. 36–43.

[33] Y. Shan, T. Wu, Y. Wang, B. Wang, Z. Wang, N. Xu, and H. Yang, "Fpga and gpu implementation of large scale spmv," in *Application Specific Processors (SASP), 2010 IEEE 8th Symposium on*. IEEE, 2010, pp. 64–70.

[34] M. Wolf and et al., "Sparse matrix partitioning for parallel eigenanalysis of large static and dynamic graphs," in *IEEE, High Performance Extreme Computing Conference*, 2014, pp. 1–6.

[35] P. Grigoras and et al., "Accelerating spmv on fpgas by compressing nonzero values," in *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2015, pp. 64–67.

[36] *Intel math kernel library*, Intel, http://software.intel.com/en-us/intel-mkl/.

[37] *Nvidia cuSPARSE*, Nvidia, http://developer.nvidia.com/cusparse.

[38] S. Jain and et al., "Implications of memory-efficiency on sparse matrix-vector multiplication," in *Symposium on Application Accelerators in High-Performance Computing*. IEEE, 2014, pp. 80–83.

[39] R. Dorrance and et al., "A scalable sparse matrix-vector multiplication kernel for energy-efficient sparse-blas on fpgas," in *Proceedings of the ACM/SIGDA international symposium on Field-programmable gate arrays*. ACM, 2014, pp. 161–170.

[40] A. Rafique and et al., "Communication optimization of iterative sparse matrix-vector multiply on gpus and fpgas," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 1, pp. 24–34, 2015.

[41] E. G. Boman and et al., "Scalable matrix computations on large scale-free graphs using 2d graph partitioning," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. ACM, 2013, p. 50.

[42] U. Catalyurek and et al., "Multithreaded clustering for multi-level hypergraph partitioning," in *IEEE Parallel Distributed Processing Symposium,*, 2012, pp. 848–859.

[43] A. Yzelman and et al., "Cache-oblivious sparse matrix-vector multiplication by using sparse matrix partitioning methods," *SIAM Journal on Scientific Computing*, vol. 31, no. 4, pp. 3128–3154, 2009.

[44] ——, "Two-dimensional cache-oblivious sparse matrix–vector multiplication," *Parallel Computing*, vol. 37, no. 12, pp. 806–819, 2011.

[45] J. Fowers and et al., "A high memory bandwidth fpga accelerator for sparse matrix-vector multiplication," in *Field-Programmable Custom Computing Machines, IEEE Symposium on*, 2014, pp. 36–43.

[46] K. K. Nagar and et al., "A sparse matrix personality for the convey hc-1," in *IEEE Symposium on Field-Programmable Custom Computing Machines*. IEEE, 2011, pp. 1–8.

[47] T. A. Davis and et al., "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, 2011.

[48] L. He, Y. Luo, and Y. Cao, "Accelerator of stacked convolutional independent subspace analysis for deep learning-based action recognition," in *Field-Programmable Custom Computing Machines (FCCM)*, 2014, pp. 104–104.

[49] S. Kestur and et al., "Towards a universal fpga matrix-vector multiplication architecture," in *Field-Programmable Custom Computing Machines, IEEE Symposium on*, 2012, pp. 9–16.

[50] L. Zhuo and V. K. Prasanna, "Sparse matrix-vector multiplication on fpgas," in *Proceedings of the ACM/SIGDA International Symposium on Field-programmable Gate Arrays*. ACM, 2005, pp. 63–74.

[51] Y. Taigman et al., "Deepface: Closing the gap to human-level performance in face verification," in *Proc. of Conf. on Computer Vision and Pattern Recognition*, 2014, pp. 1701–1708.

[52] K. He et al., "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," in *Proc. of the conf. on computer vision*, 2015, pp. 1026–1034.

[53] R. Girshick et al., "Rich feature hierarchies for accurate object detection and semantic segmentation," in *Proc. of conf. on computer vision and pattern recognition*, 2014, pp. 580–587.

[54] S. Han et al., "Learning both weights and connections for efficient neural network," in *Advances in Neural Information Processing Systems*, 2015, pp. 1135–1143.

[55] B. Liu et al., "Sparse convolutional neural networks," in *Proc. of Conf. on Comp. Vision & Pattern Recognition*, 2015, pp. 806–814.

[56] W. Wen et al., "Learning structured sparsity in deep neural networks," in *Advances in Neural Information Processing Systems*, 2016, pp. 2074–2082.

[57] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "Eie: efficient inference engine on compressed deep neural network," in *Proceedings of the 43rd International Symposium on Computer Architecture*. IEEE Press, 2016, pp. 243–254.

[58] Y. Jia et al., "Caffe: Convolutional architecture for fast feature embedding," in *Proc. of conf. on Multimedia*, 2014, pp. 675–678.

[59] G. Lacey et al., "Deep learning on fpgas: Past, present, and future," *arXiv preprint arXiv:1602.04283*, 2016.

[60] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2015, pp. 161–170.

[61] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song *et al.*, "Going deeper with embedded fpga platform for convolutional neural network," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2016, pp. 26–35.

[62] N. Suda et al., "Throughput-optimized opencl-based fpga accelerator for large-scale convolutional neural networks," in *Proc. of Symp. on Field-Programmable Gate Arrays*, 2016, pp. 16–25.

[63] C. Farabet, C. Poulet, J. Y. Han, and Y. LeCun, "Cnp: An fpga-based processor for convolutional networks," in *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*. IEEE, 2009, pp. 32–37.

[64] S. Chakradhar, M. Sankaradas, V. Jakkula, and S. Cadambi, "A dynamically configurable coprocessor for convolutional neural networks," in *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3. ACM, 2010, pp. 247–257.

[65] M. Courbariaux et al., "Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1," *arXiv preprint arXiv:1602.02830*, 2016.

[66] Y. Umuroglu et al., "Finn: A framework for fast, scalable binarized neural network inference," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '17, 2017, pp. 65–74.

[67] B. Reagen et al., "Minerva: Enabling low-power, highly-accurate deep neural network accelerators," in *Proc. Symp. on Computer Architecture*, 2016, pp. 267–278.

[68] R. Dorrance et al., "A scalable sparse matrix-vector multiplication kernel for energy-efficient sparse-blas on fpgas," in *Proc. of symp. on Field-programmable gate arrays*, pp. 161–170.

[69] D. Mahajan et al., "Tabla: A unified template-based framework for accelerating statistical machine learning," in *Symp. on High Performance Computer Architecture*, 2016, pp. 14–26.

[70] C. Zhang et al., "Caffeine: towards uniformed representation and acceleration for deep convolutional neural networks," in *Proc. of Conf. on Computer-Aided Design*, 2016, p. 12.

[71] S. Kim et al., "Tree-guided group lasso for multi-task regression with structured sparsity," in *Proc. of Conf. on Machine Learning*, 2010, pp. 543–550.

[72] O. Russakovsky et al., "Imagenet large scale visual recognition challenge," *Journal of Computer Vision*, vol. 115, no. 3, pp. 211–252, 2015.

[73] J. Albericio et al., "Cnvlutin: Ineffectual-neuron-free deep neural network computing," in *Symp. on Computer Architecture,*, 2016, pp. 1–13.

[74] A. Krizhevsky et al., "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[75] A. Krizhevsky, "Learning multiple layers of features from tiny images," 2009.

[76] K. Simonyan et al., "Very deep convolutional networks for large-scale image recognition," *arXiv preprint:1409.1556*, 2014.

[77] V. Sze, Y.-H. Chen, T.-J. Yang, and J. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *arXiv preprint arXiv:1703.09039*, 2017.

[78] N. L. Roux, M. Schmidt, and F. R. Bach, "A stochastic gradient method with an exponential convergence _rate for finite training sets," in *Advances in Neural Information Processing Systems*, 2012, pp. 2663–2671.

[79] B. Recht, C. Re, S. Wright, and F. Niu, “Hogwild: A lock-free approach to parallelizing stochastic gradient descent,” in *Advances in neural information processing systems*, 2011, pp. 693–701.

[80] L. Deng, D. Yu *et al.*, “Deep learning: methods and applications,” *Foundations and Trends® in Signal Processing*, vol. 7, no. 3–4, pp. 197–387, 2014.

[81] H. Esmaeilzadeh, E. Blem, R. St Amant, K. Sankaralingam, and D. Burger, “Dark silicon and the end of multicore scaling,” in *ACM SIGARCH Computer Architecture News*, vol. 39, no. 3. ACM, 2011, pp. 365–376.

[82] D. Hammerstrom, “A vlsi architecture for high-performance, low-cost, on-chip learning,” in *Neural Networks, 1990., 1990 IJCNN International Joint Conference on*. IEEE, 1990, pp. 537–544.

[83] U. Ramacher, J. Beichter, W. Raab, J. Anlauf, N. Bruels, U. Hachmann, and M. Wesseling, “Design of a 1st generation neurocomputer,” in *VLSI design of Neural Networks*. Springer, 1991, pp. 271–310.

[84] K. Keutzer, “If i could only design one circuit: technical perspective,” *Communications of the ACM*, vol. 59, no. 11, pp. 104–104, 2016.

[85] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun *et al.*, “Dadiannao: A machine-learning supercomputer,” in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2014, pp. 609–622.

[86] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam, “Shidiannao: Shifting vision processing closer to the sensor,” in *ACM SIGARCH Computer Architecture News*, vol. 43, no. 3. ACM, 2015, pp. 92–104.

[87] S. Han, J. Kang, H. Mao, Y. Hu, X. Li, Y. Li, D. Xie, H. Luo, S. Yao, Y. Wang *et al.*, “Ese: Efficient speech recognition engine with sparse lstm on fpga.” in *FPGA*, 2017, pp. 75–84.

[88] E. S. Chung, J. D. Davis, and J. Lee, “Linqits: Big data on little clients,” in *ACM SIGARCH Computer Architecture News*, vol. 41, no. 3. ACM, 2013, pp. 261–272.

[89] M. Sankaradas, V. Jakkula, S. Cadambi, S. Chakradhar, I. Durdanovic, E. Cosatto, and H. P. Graf, “A massively parallel coprocessor for convolutional neural networks,” in *Application-specific Systems, Architectures and Processors, 2009. ASAP 2009. 20th IEEE International Conference on*. IEEE, 2009, pp. 53–60.

[90] P. Gysel, M. Motamedi, and S. Ghiasi, “Hardware-oriented approximation of convolutional neural networks,” *arXiv preprint arXiv:1604.03168*, 2016.

[91] M. Courbariaux, Y. Bengio, and J.-P. David, “Binaryconnect: Training deep neural networks with binary weights during propagations,” in *Advances in Neural Information Processing Systems*, 2015, pp. 3123–3131.

[92] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1," *arXiv preprint arXiv:1602.02830*, 2016.

[93] F. Li, B. Zhang, and B. Liu, "Ternary weight networks," *arXiv preprint arXiv:1605.04711*, 2016.

[94] G. Venkatesh, E. Nurvitadhi, and D. Marr, "Accelerating deep convolutional networks using low-precision and sparsity," in *Acoustics, Speech and Signal Processing (ICASSP), 2017 IEEE International Conference on*. IEEE, 2017, pp. 2861–2865.

[95] A. Lavin and S. Gray, "Fast algorithms for convolutional neural networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 4013–4021.

[96] P. D'Alberto, P. A. Milder, A. Sandryhaila, F. Franchetti, J. C. Hoe, J. M. Moura, M. Puschel, and J. R. Johnson, "Generating fpga-accelerated dft libraries," in *Field-Programmable Custom Computing Machines, 2007. FCCM 2007. 15th Annual IEEE Symposium on*. IEEE, 2007, pp. 173–184.