3D CNN Acceleration on FPGA using Hardware-Aware Pruning

Mengshu Sun*[1], Pu Zhao*[1], Mehmet Gungor[1], Massoud Pedram[2], Miriam Leeser[1], Xue Lin[1]
[1]Dept. of Electrical and Computer Engineering, Northeastern University, MA, United States
[2]Dept. of Electrical and Computer Engineering, University of Southearn California, LA, United States
sun.meng@husky.neu.edu, zhao.pu@husky.neu.edu, gungor.m@husky.neu.edu,
pedram@ceng.usc.edu, mel@ece.neu.edu, xue.lin@northeastern.edu

***Abstract*—There have been many recent attempts to extend the successes of convolutional neural networks (CNNs) from 2-dimensional (2D) image classification to 3-dimensional (3D) video recognition by exploring 3D CNNs. Considering the emerging growth of mobile or Internet of Things (IoT) market, it is essential to investigate the deployment of 3D CNNs on edge devices. Previous works have implemented standard 3D CNNs (C3D) on hardware platforms, however, they have not exploited model compression for acceleration of inference. This work proposes a hardware-aware pruning approach that can fully adapt to the loop tiling technique of FPGA design and is applied onto a novel 3D network called R(2+1)D. Leveraging the powerful ADMM, the proposed pruning method achieves simultaneous high accuracy and significant acceleration of computation on FPGA. With layer-wise pruning rates up to 10× and negligible accuracy loss, the pruned model is implemented on a Xilinx ZCU102 FPGA board, where the pruned model achieves 2.6× speedup compared with the unpruned version, and 2.3× speedup and 2.3× power efficiency improvement compared with state-of-the-art FPGA implementation of C3D.**

***Index Terms*—3D CNNs, video analysis, DNN weight pruning, ADMM, FPGA.**

I. Introduction

Deep learning has made great achievements in the field of image classification with a sequence of remarkable milestones driven by convolutional neural networks (CNNs). However, the achievement in the video domain is not as significant as that in the image domain. The image-based 2D CNNs are unable to model temporal information and motion patterns, which are the critical aspects of video analysis. To perform accurate action recognition, 3D CNNs have been proposed to investigate the temporal reasoning with 3D convolutions over the spatiotemporal video volume [1], [2]. The standard 3D CNNs (C3D) may be storage and computation intensive and do not differentiate between spatial and temporal information. To address this, some variant architectures of 3D CNNs have been proposed, among which R(2+1)D CNNs exhibit superior accuracy with significantly reduced number of parameters. R(2+1)D CNNs explicitly factorize 3D convolution into two separate and successive operations i.e., a 2D spatial convolution and then a 1D temporal convolution [3]. The decomposition extends a model's capability to represent more complex functions with a lower training and testing loss. R(2+1)D CNNs achieve state-of-the-art performance on video datasets including UCF101 [4], Kinetics [5] and Sports-1M [6].

It is desirable to move deep neural network (DNN) executions from the cloud servers to the edge devices for a variety of applications such as sensor nodes, smartphones, wearable devices, robotics, unmanned vehicles, smart health device, etc. However, the huge amounts of storage and computation in state-of-the-art DNNs are prohibitive for the resource limited edge devices. On the other hand, various DNN model compression techniques, such as weight quantization and weight pruning, have been proposed to reduce the storage and accelerate the computation of DNNs. In particular, DNN weight pruning can lead to a notable model size reduction. In early work of non-structured weight pruning [7], [8], the hardware performance (inference speed) improvement is undermined by the irregular weight representation, although high model pruning rates can be reached. The structured weight pruning [9]–[12] is more hardware-friendly due to the regular weight representation, but may induce accuracy loss.

In an attempt to accelerate execution of 3D CNNs for video analysis, this work proposes a novel DNN weight pruning approach that overcomes the shortcomings of the previous DNN non-structured/structured pruning schemes in that significant improvement in the end-to-end inference time can be achieved on the FPGA prototypes with *negligible accuracy loss*, leveraging the powerful ADMM (Alternating Direction Method of Multipliers). Our contributions are summarized:

- We propose a blockwise pruning scheme for 3D DNNs that can directly match the loop tiling technique of FPGA designs. To the best of our knowledge, we are the first to apply weight pruning to 3D CNNs for FPGA implementations.
- We provide an ADMM-based solution framework that can achieve the proposed blockwise pruning scheme with negligible accuracy loss.
- We test our hardware-aware DNN weight pruning method on a novel variant of 3D CNNs, R(2+1)D, and implement the pruned model into FPGA prototypes, achieving 2.6× speedup compared with the unpruned version, and 2.3× speedup and 2.3× power efficiency improvement compared with the state-of-the-art FPGA implementation of C3D [13].

*Equal contribution

II. Related Work on 3D CNNs and Hardware Implementations

The standard 3D CNNs (C3D) [2], [14] are usually computationally intensive, inspiring various methods to accelerate computation. By improving the computation process, the work [15] achieves speedup of a 3D CNN architecture based on the Winograd minimal filter algorithm with uniform tiling sizes in the three dimensions of feature maps. Besides, novel model architectures are explored to improve storage and computation efficiency. The "mixed convolutional" structure [3] utilizes 3D convolution in either the bottom or top layers and 2D in the rest layers, and Inception 3D (I3D) [1] inflates the 2D convolutions in Inception-V1 to 3D convolutions. Other architectures include Pseudo-3D ResNet (P3D) [16], separable 3D CNN (S3D) [17] and R(2+1)D [3], all of which contain replacement of 3D convolutions with separable convolutions, e.g., convolving first spatially in 2D and temporally in 1D.

For 3D CNN hardware acceleration, the work [13] firstly implements the C3D network on FPGA with K_c parallel 2D convolution blocks where K_c is the kernel depth of the 3D convolution, utilizing blocking in pixel level and parallelism in filter and pixel levels. The Winograd algorithm is adopted in [18] with tiling for both feature maps and filters to generally accelerate 2D and 3D CNNs. Another work [19] realizes the acceleration of 2D and 3D CNNs with a customized mapping module to generate the feature matrix tilings and eliminate the need for storing the entire enlarged feature matrix. In addition, attention has been paid to multiple-FPGA platforms to further improve the computation parallelism [20]. Besides the above FPGA implementations, an accelerator for 3D CNNs on ASIC is designed in [21] to adaptively support different spatial and temporal tiling strategies for different layers.

In previous works, the hardware acceleration of R(2+1)D CNNs [3] has not been fully explored. R(2+1)D is a superior variant of 3D CNNs in that it achieves high accuracy with fewer parameters. Different from the standard 3D architecture, R(2+1)D consists of different kinds of kernels to explore spatial and temporal information separately, and contains more types of layers such as batch normalization layers and shortcuts between adjacent blocks, which makes the hardware design space exploration of R(2+1)D is more challenging than that of C3D. This paper proposes a hardware-aware weight pruning algorithm, leveraging the powerful ADMM, to achieve simultaneous high accuracy and significant acceleration of computation on FPGAs.

III. Hardware-Aware Weight Pruning of 3D CNNs Leveraging ADMM

A. Preliminaries of 3D CNNs and Proposed Weight Sparsity Pattern

The weights $\mathbf{W}_i$ of the i-th convolutional (CONV) layer can be denoted as a 5D tensor, i.e., $\mathbf{W}_i \in \mathbb{R}^{M \times N \times K_d \times K_r \times K_c}$, where M and N represent the numbers of output and input channels, and K_d, K_r and K_c are kernel sizes corresponding to the three dimensions D, R and C of the feature maps. R and C denote the spatial height and width which are similar to 2D CNNs, and D indicates the temporal depth. For efficiently mapping to the FPGA hardware, a weight tensor $\mathbf{W}_i$ can be treated as multiple *blocks* of 3D kernels, as demonstrated in Fig. 1 (left). For a weight tensor $\mathbf{W}_i$ with M output channels and N input channels, the $M \times N$ 3D kernels are divided into blocks of size $T_m \times T_n$, thus the weight tensor contains $\lceil \frac{M}{T_m} \rceil \times \lceil \frac{N}{T_n} \rceil$ blocks. In the FPGA implementation, the buffer to store weights has the same size as one weight block due to limited amounts of memory and computation resources. The weight blocks of the whole weight tensor are one by one loaded into the buffer and then participate in the computation process, which will be elaborated in Section IV-A.

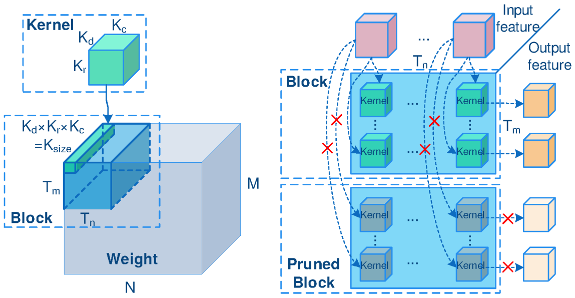

Fig. 1. A weight tensor can be divided into blocks and pruned blockwise.

Weight pruning sets a certain amount of weight values to zeros to simultaneously reduce the model size and the amount of computations. More specifically, for the i-th CONV layer, a pruning ratio $\eta_i \in [0\%, 100\%]$ is achieved if at least η_i of all values in $\mathbf{W}_i$ are zeros. In our blockwise weight pruning scheme, the basic pruning unit is one weight block as shown in Fig. 1 (right), and the values of a pruned block are all set to zeros. Hence, the sparsity requirement for the i-th layer is

$$\mathbf{S}_i = \{\hat{\mathbf{W}}_i | E_i \leq (1 - \eta_i) \times \lceil \frac{M}{T_m} \rceil \times \lceil \frac{N}{T_n} \rceil\} \tag{1}$$

where E_i denotes the number of non-zero blocks in the pruned weight tensor $\hat{\mathbf{W}}_i$. Please note that the proposed blockwise weight pruning scheme can be applied to different types of 3D CNNs including C3D and R(2+1)D.

B. Weight Pruning Problem Formulation

The loss function of a 3D CNN can be expressed as $f(\{\mathbf{W}_i\})$. As CONV layer weights account for the majority of the storage and computation,we mainly focus on pruning the CONV layer weights. So we omit the expression of inputs and fully-connected (FC) weights in the above loss function.

Our objective is to prune the model weights so that the parameters would satisfy the sparsity requirements, while the pruned model can still maintain high test accuracy. To achieve this, we mainly minimize the loss function subject to the sparsity constraints on the weights, as specified below,

$$\underset{\{\mathbf{W}_i\}}{\text{minimize}} \quad f(\{\mathbf{W}_i\}), \quad \text{s.t.} \quad \mathbf{W}_i \in \mathbf{S}_i, \quad \forall i \tag{2}$$

As the constraint in problem (2) is combinatorial, it cannot be directly solved through stochastic gradient descent (SGD) methods. Therefore, we propose a systematic framework based on ADMM. ADMM is shown to be effective when facing such clustering-like constraints [22], [23]. The ADMM-based solution framework is shown in the following.

C. ADMM Reformulation

We define the indicator function as

$$g_i(\mathbf{W}_i) = \begin{cases} 0 & \text{if } \mathbf{W}_i \in \mathbf{S}_i, \\ +\infty & \text{otherwise.} \end{cases} \tag{3}$$

Besides, new auxiliary variables $\mathbf{Z}_i$ for all i are introduced and the original problem (2) can be transformed to

$$\begin{aligned} \underset{\{\mathbf{W}_i\}}{\text{minimize}} \quad & f\big(\{\mathbf{W}_i\}\big) + \sum_{i=1}^{L} g_i(\mathbf{Z}_i), \\ \text{subject to} \quad & \mathbf{W}_i = \mathbf{Z}_i, \ \ \forall i. \end{aligned} \tag{4}$$

The augmented Lagrangian function [23] is expressed as

$$\begin{aligned} L_\rho(\{\mathbf{W}_i\}, \{\mathbf{Z}_i\}, \{\mathbf{V}_i\}) = f\big(\{\mathbf{W}_i\}\big) + \sum_{i=1}^{L} \mathbf{U}_i^T \left(\mathbf{W}_i - \mathbf{Z}_i\right) \\ + \sum_{i=1}^{L} g_i(\mathbf{Z}_i) + \sum_{i=1}^{L} \frac{\rho}{2} \left\| \mathbf{W}_i - \mathbf{Z}_i \right\|_F^2, \end{aligned} \tag{5}$$

where $\mathbf{U}_i$ is the dual variable or Lagrange multiplier with the same dimension as $\mathbf{W}_i$, ρ is the penalty parameter for the ADMM loss, L is the number of CONV layers in the model and $\|\cdot\|_F$ denotes the Frobenius norm. Inspired by the scaled form of ADMM through defining $\mathbf{U}_i = \rho \mathbf{V}_i$, the augmented Lagrangian function can be further expressed as

$$\begin{aligned} L_\rho(\{\mathbf{W}_i\}, \{\mathbf{Z}_i\}, \{\mathbf{V}_i\}) = f\big(\{\mathbf{W}_i\}\big) + \sum_{i=1}^{L} g_i(\mathbf{Z}_i) \\ + \sum_{i=1}^{L} \frac{\rho}{2} \left\| \mathbf{W}_i - \mathbf{Z}_i + \mathbf{V}_i \right\|_F^2 - \left\| \mathbf{V}_i \right\|_F^2. \end{aligned} \tag{6}$$

D. ADMM Iteration

The ADMM solves problem (4) in an iterative manner as

$$\{\mathbf{W}_i^{k+1}\} = \underset{\{\mathbf{W}_i\}}{\operatorname{argmin}}\, L_\rho(\{\mathbf{W}_i\}, \{\mathbf{Z}_i^k\}, \{\mathbf{V}_i^k\}) \tag{7}$$

$$\{\mathbf{Z}_i^{k+1}\} = \underset{\{\mathbf{Z}_i\}}{\operatorname{argmin}}\, L_\rho(\{\mathbf{W}_i^{k+1}\}, \{\mathbf{Z}_i\}, \{\mathbf{V}_i^k\}) \tag{8}$$

$$\mathbf{V}_i^{k+1} = \mathbf{V}_i^k + \mathbf{W}_i^{k+1} - \mathbf{Z}_i^{k+1}, \forall i. \tag{9}$$

Initially, we set $\mathbf{Z}_i^0 = \mathbf{W}_i^0$ and $\mathbf{V}_i^0 = \mathbf{0}, \forall i$. The iteration stops when the following conditions are satisfied,

$$\|\mathbf{W}_i^{k+1} - \mathbf{Z}_i^{k+1}\| \le \epsilon_i, \|\mathbf{Z}_i^{k+1} - \mathbf{Z}_i^k\| \le \epsilon_i, \tag{10}$$

where ϵ_i is a threshold. The ADMM steps are equivalent to the following Proposition 1.

Proposition 1: The ADMM subproblems (7) and (8) can be equivalently transformed into a) $\mathbf{W}$-minimization step and b) $\mathbf{Z}$-minimization step. More specifically,

$\mathbf{W}$-minimization step: $\{\mathbf{W}_i^{k+1}\}$ in (7) can be obtained through solving the following equivalent problem,

$$\underset{\{\mathbf{W}_i\}}{\text{minimize}} \quad f\big(\{\mathbf{W}_i\}\big) + \sum_{i=1}^{L} \frac{\rho}{2} \|\mathbf{W}_i - \mathbf{Z}_i^k + \mathbf{V}_i^k\|_F^2. \tag{11}$$

Both terms in problem (11) are differentiable such that gradient descent method can be applied to obtain its solution. It is similar to training a model with a loss function including an additional regularization term.

$\mathbf{Z}$-minimization step: $\{\mathbf{Z}_i^{k+1}\}$ is given by solving

$$\underset{\{\mathbf{Z}_i\}}{\text{minimize}} \quad \sum_{i=1}^{L} g_i(\mathbf{Z}_i) + \sum_{i=1}^{L} \frac{\rho}{2} \|\mathbf{W}_i^{k+1} - \mathbf{Z}_i + \mathbf{V}_i^k\|_F^2. \tag{12}$$

First we note that $\mathbf{Z}_i$ is not correlated with $\mathbf{Z}_j, j \neq i$. So the above problem can be solved layerwisely. For the i-th layer, since $g_i(\cdot)$ is the indicator function of $\mathbf{S}_i$, we can directly obtain its optimal solution [24], which is the Euclidean projection of $\mathbf{W}_i^{k+1} + \mathbf{V}_i^k$ onto $\mathbf{S}_i$, as follows,

$$\mathbf{Z}_i^{k+1} = \prod_{\mathbf{S}_i} \left(\mathbf{W}_i^{k+1} + \mathbf{V}_i^k \right), \quad \forall i. \tag{13}$$

where $\prod(\cdot)$ denotes Euclidean projection. More specifically, we initially set $\mathbf{Z}_i^{k+1} = \mathbf{W}_i^{k+1} + \mathbf{V}_i^k, \forall i$. As $\mathbf{S}_i$ requires η_i blocks to zero, we first sort all of the blocks in $\boldsymbol{Z}_i^{k+1}$ in ascending order according to the ℓ_2 norm of each block. Then we are able to obtain a threshold $\zeta_i = P(\eta_i, \boldsymbol{Z}_i^{k+1})$ which computes the $(\eta_i \times 100)$-th percentile of the sorted ℓ_2 norm of all block in $\mathbf{Z}_i^{k+1}$. Finally, we set the blocks with an ℓ_2 norm under ζ_i to zeros and obtain the solution $\{\boldsymbol{Z}_i^{k+1}\}$ for Problem 8.

Note that in problem (11), $\{\mathbf{Z}_i^k\}$ and $\{\mathbf{V}_i^k\}$ are given values from previous iterations and only $\{\mathbf{W}_i\}$ are variables to be updated. Similarly, in problem (12), $\{\mathbf{W}_i^{k+1}\}$ are given values from solving problem (11), and $\{\mathbf{Z}_i\}$ are variables to be updated. After $\{\mathbf{W}_i^{k+1}\}$ and $\{\mathbf{Z}_i^{k+1}\}$ are obtained, we are able to update $\{\mathbf{V}_i\}$ as shown in Equation (9).

E. Masking and Retraining

After the ADMM iterations, we obtain $\mathbf{W}_i$ and find that the accuracy may suffer from a relatively significant degradation. To improve the pruning accuracy, we perform a masking and retraining step afterwards. In detail, we retrain the pruned model and only the non-zero weights are updated during retraining. The zero weights are masked out and not updated. With the retraining step, we can further improve the accuracy. Algorithm 1 summarizes our ADMM solution framework.

IV. FPGA Implementations of 3D CNNs

A. Implementation of 3D CNNs on FPGA

This section discusses about the FPGA implementations of 3D CNNs, with the support of unpruned and pruned models

Algorithm 1: ADMM-based blockwise pruning framework

```
ρ = Penalty parameter of ADMM loss; r = Multi-ρ number;
W = Model weights; Z, V = ADMM sub-problem variables;
L = Number of CONV layers; B_i = Number of blocks in layer i;
epoch_ρ = Number of epochs in one round of ADMM training;
epoch_admm = Number of epochs between updates of Z, V;
ζ_i = Pruning threshold for layer i;

Initialize ρ, Z = W, V = 0;
for $j_1 \leftarrow 1$ to $r$ do
    for $j_2 \leftarrow 1$ to $epoch_\rho$ do
        Update W by solving Eq. (6);  //ADMM training
        if $epoch_\rho \bmod epoch_{admm} = 0$ then
            for $i \leftarrow 1$ to $L, b \leftarrow 1$ to $B_i$ do
                $\mathbf{Z}_b^i[:] = 0$ if $\|\mathbf{W}_b^i[:]\|^2 < \zeta_i$;  //Update Z
            $\mathbf{V} = \mathbf{V} + \mathbf{W} - \mathbf{Z}$;  //Update V
    Expand ρ;

Load W, hard prune and retrain with masks.
```

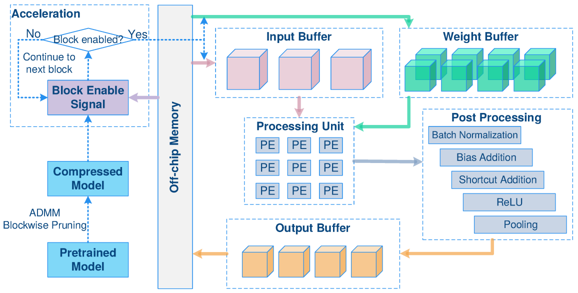

Fig. 2. Hardware architecture of tiled convolution of 3D CNNs with support of pruned/unpruned models by the block enable signal.

by the block enable signal, as shown in Fig. 2. For pruned 3D CNNs, the block enable signal is fetched from a pre-stored array generated for the pruned model to decide whether to load the corresponding input feature and weight tiles into on-chip memory. If the block enable signal is active, the corresponding weight block is retained and would participate in the computation, while an inactive block enable signal indicates that the weight block has been pruned and therefore the loading and computing processes are skipped one time. To save the computation resources and speedup the computations, the results from the convolution computations in the processing unit are further handled by the post processing unit, when there is a subsequent batch normalization, bias addition, a shortcut layer from the last residual block, an activation (ReLU) operation, or a pooling layer.

Algorithm 2 displays the tiled convolution procedure in details. In addition to the output and input channels, the three dimensions D, R and C of the output feature maps are also tiled with tiling parameters T_d, T_r and T_c. For a convolution layer with weights $\boldsymbol{W}[M][N][K_d][K_r][K_c]$, input feature maps $I[N][(D-1)\times S_d+K_d][(R-1)\times S_r+K_r][(C-1)\times S_c+K_c]$, and output feature maps $O[M][D][R][C]$, the weight tile, the input tile, and the output tile on the buffer are denoted as $W_{buf}[T_m][T_n][K_{size}]$, $I_{buf}[T_n][I_{size}]$, and $O_{buf}[T_m][T_d][T_r][T_c]$, respectively. The kernel and stride sizes of the convolutional layer are represented by K_x and S_x where $x = d, r, c$. The computations are conducted in a pipelined scheme in the processing unit with multiple processing elements (PE). The unrolling of loops L2 and L3 in the computation module in Algorithm 2 enables $T_m \times T_n$ Multiply-and-Accumulate (MAC) operations in parallel in one cycle. The accumulation of T_n multiplication results is implemented via an adder tree with a depth of $\log_2(T_n)$. Array partition is performed in corresponding dimensions of the buffers to increase bandwidth. Additionally, the double buffering technique is utilized to reduce the latency through overlapping data transfer with computation.

B. Design Space Exploration

The tiling technique is required for FPGA computation because of the limited resources on FPGA, and the tiling size parameters need to be chosen delicately for efficient resource utilization. The most critical resources are related to memory and computation, corresponding to BRAMs and DSPs on the FPGA board. The tiling method is leveraged in 5 dimensions, i.e., T_m, T_n, T_d, T_r, and T_c corresponding to the 5 dimensions of the tensor blocks.

Algorithm 2: Tiled Convolution for 3D CNNs

```
for ($d = 0 : T_d : D, r = 0 : T_r : R, c = 0 : T_c : C$) {
  $j_x = x \cdot S_x : (x + T_x - 1) \cdot S_x + K_x$ for $x = d, r, c$
  for ($m = 0 : T_m : M$) {
    for ($n = 0 : T_n : N$) {
      if (block is not enabled)
        continue;
      Load $W_{buf} = W[m : m + T_m][n : n + T_n]$;
      Load $I_{buf} = I[n : n + T_n][j_d][j_r][j_c]$;
      Compute($I_{buf}, W_{buf}, O_{buf}$);
    }
    Post processing;
    Store $O[m : m + T_m]$
      $[d : d + T_d][r : r + T_r][c : c + T_c] = O_{buf}$;
}}

Compute($I_{buf}, W_{buf}, O_{buf}$) {
  $T'_x = (T_x - 1) \times S_x + K_x$ for $x = d, r, c$;
  for ($k_d = 0 : K_d, k_r = 0 : K_r, k_c = 0 : K_c$)
    L1: for ($t_d = 0 : T_d, t_r = 0 : T_r, t_c = 0 : T_c$) {
      $j_x = t_x \times S_x + k_x$ for $x = d, r, c$;
#PIPELINE
      L2: for ($t_m = 0 : T_m$) {
#UNROLL
        L3: for ($t_n = 0 : T_n$) {
#UNROLL
          $j_w = W_{buf}[t_m][t_n][(k_d \times K_h + k_h) \times K_w + k_w]$;
          $j_i = I_{buf}[t_n][(j_d \times T'_r + j_r) \times T'_c + j_c]$;
          $O_{buf}[t_m][t_d][t_r][t_c] += j_w \times j_i$;
}}}}
```

1) Resource Utilization: Considering double buffering, the memory utilization of output, input and weight buffers can be calculated as

$$B_{out} = 2 \times T_m \times T_d \times T_r \times T_c, \tag{14}$$

$$B_{in} = 2 \times T_n \times I_{size}, \tag{15}$$

$$B_{wgt} = 2 \times T_m \times T_n \times K_{size} \tag{16}$$

where I_{size} and K_{size} considered integrally in three dimensions to save the BRAM utilization and adapt to distinct kernel and input feature shapes of different convolutional layers, satisfying

$$\begin{aligned} K_{size} &= \max_i \{K_d^i \times K_r^i \times K_c^i\}, \\ I_{size} &= \max_i \{((T_d - 1) \times S_d^i + K_d^i) \times \\ &\quad ((T_r - 1) \times S_r^i + K_r^i) \times ((T_c - 1) \times S_c^i + K_c^i)\}. \end{aligned} \tag{17}$$

K_x^i and S_x^i for $x = d, r, c$ are kernel and stride sizes of the i-th convolutional layer. The overall utilization of BRAMs is constrained by

$$\lceil (B_{out} + B_{in} + B_{wgt}) \times N_{bit}/36\text{K} \rceil \leq \mathcal{B}, \tag{18}$$

where N_{bit} is 16 for 16-bit fixed-point data and $\mathcal{B}$ is the number of available BRAMs on the board each with space of 36K bits.

For the DSP utilization, as $T_m \times T_n$ MAC operations are performed in parallel for convolution computations and each MAC utilizes one DSP block, the value $T_m \times T_n$ should be no more than the number of available DSPs.

TABLE I
R(2+1)D MODEL ARCHITECTURE

Layer (Residual Block)	Output Size	Kernel/Filter Size
conv1	$16 \times 56 \times 56$	$\begin{bmatrix} 1 \times 7 \times 7, 45 \\ 3 \times 1 \times 1, 64 \end{bmatrix}$
conv2_x	$16 \times 56 \times 56$	$\begin{bmatrix} 1 \times 3 \times 3, 144 \\ 3 \times 1 \times 1, 64 \\ 1 \times 3 \times 3, 144 \\ 3 \times 1 \times 1, 64 \end{bmatrix} \times 2$
conv3_x	$8 \times 28 \times 28$	$\begin{bmatrix} 1 \times 3 \times 3, 288(230) \\ 3 \times 1 \times 1, 128 \\ 1 \times 3 \times 3, 288 \\ 3 \times 1 \times 1, 128 \end{bmatrix} \times 2$
conv4_x	$4 \times 14 \times 14$	$\begin{bmatrix} 1 \times 3 \times 3, 576(460) \\ 3 \times 1 \times 1, 256 \\ 1 \times 3 \times 3, 576 \\ 3 \times 1 \times 1, 256 \end{bmatrix} \times 2$
conv5_x	$2 \times 7 \times 7$	$\begin{bmatrix} 1 \times 3 \times 3, 1152(921) \\ 3 \times 1 \times 1, 512 \\ 1 \times 3 \times 3, 1152 \\ 3 \times 1 \times 1, 512 \end{bmatrix} \times 2$
	$1 \times 1 \times 1$	spatio-temporal average pooling, FC layer with softmax

2) Performance Analysis: The running time of the design depends on two aspects, i.e., the latency of data transmissions between off-chip and on-chip memory, and the latency of computations. Here the latency is discussed as indicated by the number of cycles needed. Given the numbers of ports p_{wgt}, p_{in} and p_{out} to transfer weights, input features and output features, the latency required to load the data can be described by

$$t_{wgt} = T_m \times T_n \times K_d \times K_r \times K_c / p_{wgt}, \tag{19}$$

$$t_{in} = T_n \times T'_d \times T'_r \times T'_c / p_{in}, \tag{20}$$

$$t_{out} = T_m \times T_d \times T_r \times T_c / p_{out}, \tag{21}$$

where $T'_x = (T_x - 1) \times S_x + K_x$ for $x = d, r, c$. The data in the input and weight buffers would generate the output data with $K_d \times K_r \times K_c \times T_d \times T_r \times T_c \times T_m \times T_n$ MAC operations. Since the processing element could execute $T_m \times T_n$ MACs in parallel in one cycle, the computation latency is given by

$$t_{comp} = K_d \times K_r \times K_c \times T_d \times T_r \times T_c. \tag{22}$$

Benefiting from double buffering, the loading of input feature and weight data and the computations of convolutions can be processed simultaneously. Thus the latency in loop L3 is determined by

$$t_{L3} = \max\{t_{wgt}, t_{in}, t_{comp}\}, \tag{23}$$

and latency in loop L2 is

$$t_{L2} = \max\{t_{L3} \times \lceil \frac{N}{T_n} \rceil + t_{comp}, t_{out}\}. \tag{24}$$

Therefore, the overall latency is given by

$$t_{tot} = \lceil \frac{D}{T_d} \rceil \times \lceil \frac{R}{T_r} \rceil \times \lceil \frac{C}{T_c} \rceil \times \lceil \frac{M}{T_m} \rceil \times t_{L2} + t_{out}. \tag{25}$$

As the storing latency t_{out} is overlapped by $\lceil N/T_n \rceil$ times of the latency t_{L3}, t_{L3} becomes crucial to the overall performance, and the loading latency t_{wgt}, t_{in} and the computation latency t_{comp} need to be well balanced.

V. EXPERIMENTAL RESULTS

Although our proposed ADMM-based blockwise pruning algorithm and FPGA implementation are general for different types of 3D CNNs, we focus on the R(2+1)D CNN because of its superior accuracy and significantly reduced number of parameters. The R(2+1)D CNN consists of 40 convolutional (CONV) layers in 5 blocks and 1 additional fully-connected (FC) layer. The first block contains 2 layers and the others are residual blocks each containing 8 primary layers. In addition, the last 3 blocks each have a shortcut with 2 layers. Therefore, it makes a total of 2+4×8+3×2 CONV layers. The only FC layer contributes little to the number of parameters, so our weight pruning focuses on the CONV layers. More details are in Table I. Specifically, different from C3D with actual 3D kernels, the R(2+1)D network decomposes each 3D convolution into a 2D convolution in spatial dimensions and a 1D convolution in the temporal dimension, i.e., the feature maps are convoluted alternately by kernels with shape $1 \times K \times K$ and kernels with shape $K \times 1 \times 1$.

TABLE II
RESULTS OF ADMM PRUNING ALGORITHM

Layer	Number of Parameters (M)		Operations (giga)	
(Residual Block)	Before/After Pruning	Pruning Rate	Before/After Pruning	Pruning Rate
conv1	0.015	N/A	1.53	N/A
conv2_x	0.444/0.045	9.85×	44.39/4.35	10.19×
conv3_x	1.56/0.322	4.85×	21.21/4.33	4.89×
conv4_x	6.23	N/A	10.61	N/A
conv5_x	24.92	N/A	5.31	N/A
Total	33.22/31.53	1.05×	83.05/26.13	3.18×

TABLE III
FGPA RESOURCE UTILIZATION

Design	Resource	DSP	BRAM	LUT	FF
	Available	2520	912	274K	548K
$(T_m, T_n) = (64, 8)$	Used	695	710.5	74K	51K
	Utilization	28%	78%	27%	9%
$(T_m, T_n) = (64, 16)$	Used	1215	912	148K	76K
	Utilization	48%	100%	54%	14%

The original R(2+1)D model is pretrained on the Kinetics dataset and transferred onto the UCF101 dataset as the unpruned model. The pruned R(2+1)D model is obtained by the proposed ADMM pruning and masked retraining on UCF101. The initial learning rate is 5×10^{-3} when training the unpruned model, and is reduced to 5×10^{-4} in both ADMM training and masked retraining for stability. The batch size is fixed to 32, and the video clip length is 16 frames. ADMM pruning is conducted four rounds each with 50 epochs and the penalty factor ρ is set to 0.0001, 0.001, 0.01, and 0.1 corresponding to the four rounds, respectively. In each round, the pruning framework has 5 ADMM iterations with 10 epochs for the W-minimization step in each iteration. Masked retraining is performed for 100 epochs after ADMM pruning. η_i is 90% for the second residual block and 80% for the third residual block, corresponding to pruning rates of 10× and 5×. In addition, several tricks [25] are utilized to assist the training, such as label smoothing in ADMM training, as well as warmup and cosine scheduling of the learning rate in masked retraining.

We use two block size configurations $(T_m, T_n) = (64, 8)$ and $(T_m, T_n) = (64, 16)$ in the pruning algorithm with the same pruning ratios η_i, i.e., we mainly prune the second and third residual-blocks as they are the most computation intensive. The unpruned model has an accuracy of 89.0%, while the accuracy after pruning is 88.66% for $(T_m, T_n) = (64, 8)$ and 88.40% for $(T_m, T_n) = (64, 16)$. Table II demonstrates the pruning results of $(T_m, T_n) = (64, 8)$, with significant reduction of the number of operations (giga) by 3.18× for the whole model.

TABLE IV
PERFORMANCE COMPARISON WITH PREVIOUS IMPLEMENTATIONS, CPU AND GPU

Network	C3D					R(2+1)D			
Device	ZC706 [13]	VC709 [18]	VUS440 [18]	Ours ($T_n = 8$)	Ours ($T_n = 16$)	GPU (GTX 1080 Ti)	CPU (E5-1650 v4)	Ours ($T_n = 8$)	Ours ($T_n = 16$)
Frequency (MHz)	176	150	200	150	150	1481	3600	150	150
Precision	16-bit fixed					32-bit float		16-bit fixed	
Technology	28nm	28nm	20nm	16nm		16nm	14nm	16nm	
Power (W)	9.7	25	26	5.4	6.7	230	-	5.4	6.7
Throughput (GOPS)	71.0	430.7	784.7	46.6	79.1	3256.9	68.1	67.7	111.7
Power Efficiency (GOPS/W)	7.3	17.1	30.2	8.6	11.8	14.2	-	12.5	16.7
DSP Utilization	810(90%)	1536(42%)	1536(53%)	695(28%)	1215(48%)	-	-	695(28%)	1215(48%)
DSP Efficiency (GOPS/DSPs)	0.088	0.41	0.60	0.067	0.065	-	-	0.097	0.092
Latency (ms)	542.5	89.4	49.1	826	487	25.5	1220	**386** (1044)	**234** (609)

The hardware design is implemented on the Xilinx ZCU102 FPGA board through Xilinx Vivado 2019.1 with high-level synthesis. Two tiling settings are explored, respectively, with tiling factors $(T_m, T_n) = (64, 8)$ and $(T_m, T_n) = (64, 16)$. The other tiling factors are fixed as $(T_d, T_r, T_c) = (4, 14, 14)$. The data precision is 16-bit fixed-point with 1 sign bit, 7 integer bits and 8 fractional bits. Table III shows the resource utilization of the proposed design, indicating that the case with larger tiling block size $(T_m, T_n) = (64, 16)$ requires more resources than those for $(T_m, T_n) = (64, 8)$. Table IV displays the performance comparison of the proposed framework with previous implementations on FPGA platforms as well as an Intel(R) Xoen(R) E5-1650 v4 CPU and a GeForce GTX 1080 Ti GPU. In order to compare with previous FPGA implementations we also implement unpruned C3D on our FPGA board. For R(2+1)D, we report on the pruned model. In terms of latency, the value in brackets is for unpruned R(2+1)D models. Comparing with [13], we can achieve comparable latency on C3D with less power consumption, and our pruned R(2+1)D can have much lower latency (with around 2.3× speedup) and much higher power efficiency (by 2.3×) than [13]. Comparing with unpruned version, our pruned R(2+1)D achieves around 2.6× speedup in terms of latency. Please note that our work focuses on a hardware-aware pruning approach, our FPGA implementations are very basic. Therefore, there is still space for improvements in FPGA design. In addition, the design space exploration of R(2+1)D is more challenging due to irregular kernel sizes, shortcuts, and more types of operations. But we achieve significant improvements in terms of latency and power efficiency comparing between pruned version and unpruned version. Therefore, our novel pruning approach can complement more advanced FPGA design.

VI. CONCLUSION

In this work, we propose a hardware-aware DNN weight pruning algorithm for 3D CNNs leveraging ADMM. We target for actual acceleration of the inference speed on FPGA with negligible accuracy loss. We test our pruning approach on R(2+1)D CNN i.e., a superior variant of C3D, and hardware implementation on a Xilinx ZCU102 FPGA board.

ACKNOWLEDGMENT

This work is partly supported by the National Science Foundation CCF-1901378 and CCF-1901440.

REFERENCES

[1] J. Carreira and A. Zisserman, “Quo vadis, action recognition? a new model and the kinetics dataset,” in *CVPR*, 2017, pp. 6299–6308.

[2] S. Ji, W. Xu, M. Yang, and K. Yu, “3d convolutional neural networks for human action recognition,” *TPAMI*, vol. 35, no. 1, pp. 221–231, 2012.

[3] D. Tran, H. Wang, L. Torresani *et al.*, “A closer look at spatiotemporal convolutions for action recognition,” in *CVPR*, 2018.

[4] K. Soomro, A. R. Zamir, and M. Shah, “Ucf101: A dataset of 101 human actions classes from videos in the wild,” *arXiv:1212.0402*, 2012.

[5] W. Kay, J. Carreira, K. Simonyan *et al.*, “The kinetics human action video dataset,” *arXiv preprint arXiv:1705.06950*, 2017.

[6] A. Karpathy, G. Toderici, S. Shetty *et al.*, “Large-scale video classification with convolutional neural networks,” in *CVPR*, 2014.

[7] T. Zhang, S. Ye *et al.*, “A systematic dnn weight pruning framework using alternating direction method of multipliers,” in *ECCV*, 2018.

[8] A. Ren, T. Zhang, S. Ye *et al.*, “Admm-nn: An algorithm-hardware co-design framework of dnns using alternating direction method of multipliers,” in *ASPLOS*, 2019.

[9] W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li, “Learning sparsity in deep neural networks,” in *NIPS*, 2016.

[10] Y. He, X. Zhang, and J. Sun, “Channel pruning for accelerating very deep neural networks,” in *ICCV*, 2017.

[11] H. Mao, S. Han, J. Pool *et al.*, “Exploring the regularity of sparse structure in convolutional neural networks,” *arXiv:1705.08922*, 2017.

[12] C. Ding, S. Liao *et al.*, “Circnn: accelerating and compressing dnns using block-circulant weight matrices,” in *MICRO*, 2017.

[13] H. Fan, X. Niu, Q. Liu, and W. Luk, “F-c3d: Fpga-based 3-dimensional convolutional neural network,” in *FPL*, Sep. 2017.

[14] D. Tran, L. Bourdev, R. Fergus *et al.*, “Learning spatiotemporal features with 3d convolutional networks,” in *ICCV*, 2015.

[15] Z. Wang, Q. Lan, H. He, and C. Zhang, “Winograd algorithm for 3d convolution neural networks,” in *International Conference on Artificial Neural Networks*. Springer, 2017, pp. 609–616.

[16] Z. Qiu, T. Yao, and T. Mei, “Learning spatio-temporal representation with pseudo-3d residual networks,” in *ICCV*, 2017.

[17] S. Xie, C. Sun *et al.*, “Rethinking spatiotemporal feature learning: Speed-accuracy trade-offs in video classification,” in *ECCV*, 2018.

[18] J. Shen, Y. Huang, Z. Wang *et al.*, “Towards a uniform template-based architecture for accelerating 2d and 3d cnns on fpga,” in *FPGA*, 2018.

[19] Z. Liu, P. Chow, J. Xu *et al.*, “A uniform architecture design for accelerating 2d and 3d cnns on fpgas,” *Electronics*, 2019.

[20] W. Jiang, E. H.-M. Sha, X. Zhang, L. Yang, Q. Zhuge, Y. Shi, and J. Hu, “Achieving super-linear speedup across multi-fpga for real-time dnn inference,” *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 18, no. 5s, pp. 1–23, 2019.

[21] K. Hegde, R. Agrawal, Y. Yao, and C. W. Fletcher, “Morph: Flexible acceleration for 3d cnn-based video understanding,” in *MICRO*, 2018.

[22] M. Hong and Z.-Q. Luo, “On the linear convergence of the alternating direction method of multipliers,” *Mathematical Programming*, 2017.

[23] S. Boyd, N. Parikh, E. Chu, B. Peleato, J. Eckstein *et al.*, “Distributed optimization and statistical learning via the alternating direction method of multipliers,” *Foundations and Trends® in Machine Learning*, 2011.

[24] N. Parikh, S. Boyd *et al.*, “Proximal algorithms,” *Foundations and Trends® in Optimization*, vol. 1, no. 3, pp. 127–239, 2014.

[25] T. He, Z. Zhang, H. Zhang *et al.*, “Bag of tricks for image classification with convolutional neural networks,” in *CVPR*, 2019.