DiMO-Sparse: Differentiable Modeling and Optimization of Sparse CNN Dataflow and Hardware Architecture

Jianfeng Song
ECE, Texas A&M
College Station, USA
jsong26@tamu.edu

Rongjian Liang
Nvidia
Austin, USA
rliang@nvidia.com

Yu Gong
ECE, Rutgers University
Piscataway, USA
yg430@soe.rutgers.edu

Bo Yuan
ECE, Rutgers University
Piscataway, USA
bo.yuan@soe.rutgers.edu

Jiang Hu
ECE&CSE, Texas A&M
College Station, USA
jianghu@tamu.edu

***Abstract*—Many real-world CNNs exhibit sparsity, a characteristic that has primarily been utilized in manual design processes and has received little attention in existing automatic optimization techniques. To the best of our knowledge, this paper presents the first systematic investigation of automatic dataflow and hardware optimization for sparse CNN computation. A differentiable PPA (Power Performance Area) model incorporating stochastic modeling of sparse CNN workloads is developed to enable fast nonlinear optimization solving and massively parallel local search-based discretization. Experimental results on public domain testcases demonstrate the efficacy of the proposed approach, achieving an average of 5× and 10× better PPA than the previous work for two different sparsity patterns.**

I. Introduction

The growing complexity of CNN (Convolutional Neural Networks) has made it challenging to accelerate large CNNs on resource-constrained devices. One solution to this challenge is to leverage the sparsity in compact CNN weight arrays and sparse input arrays brought by widely-used ReLU activation layers [1]. In practice, sparse CNNs can reduce the model size by over 10× without significant loss of accuracy [2], [3].

To leverage the advantage of sparsity, previous sparse CNN accelerators [4]–[8] propose various sparsity-aware acceleration techniques, which are categorized into three SAFs (sparse acceleration features) in [9]: representation format, gating, and skipping.

Different SAFs achieve various trade-offs between the hardware and computation overhead to leverage sparsity vs. the reduction of unnecessary data movement and computation enabled by the sparsity. Therefore, it is crucial to choose appropriate SAFs, as deploying unsuitable ones can significantly degrade the efficiency of CNN computations. Moreover, the selection of SAFs is closely intertwined with dataflow mapping [10] and hardware resource allocation [11]. The vast joint search space presents a substantial challenge to efficient manual designs. Thus, there is a compelling need for an automatic hardware architecture search algorithm specifically tailored for sparse CNN computation.

SparseLoop [9] proposes an analytical PPA (Power Performance Area) model for a sparse tensor accelerator and achieves high accuracy in evaluating real design. However, SparseLoop contains many "if-else" operations that prevent its usage in analytical optimization. As a result, only exhaustive and random search are developed in SparseLoop using the models. Despite its primitive nature, this work is perhaps the only attempt at automating the optimization of dataflow and hardware architecture for sparse CNN computation. Prior works on dense CNN [10]–[13] are not applicable for sparse CNNs, since they do not consider the effect of sparse workload characteristics and SAFs.

In this study, we introduce DiMO-Sparse, a differentiable modeling and optimization tool for enhancing PPA in sparse CNN computation. It optimizes SAFs, dataflow, and hardware resources, accounting for sparse workload characteristics and hardware constraints. In contrast to prior differentiable models [14] that approximate gradients using Gumbel softmax estimators and require expensive sampling on non-differentiable PPA simulators. Our model is inherently differentiable. This allows for precise gradient calculations, resulting in an efficient search algorithm for finding good solutions. Our main contributions are summarized as follows:

- This is the first systematic investigation on automatic hardware architecture optimization for sparse CNN computation, to the best of our knowledge.
- The first differentiable PPA model that incorporates stochastic modeling of sparse CNN workloads. Achieved a maximum of **3**% error in model accuracy with 20K speedup, compared with SparseLoop, the state-of-the-art sparse PPA model.
- A differentiable optimization algorithm to efficiently search through the joint space of SAFs, dataflow, and hardware resource assignment. We develop a network-wise optimization algorithm rather than only targeting one CNN layer [12].
- Demonstration of the effectiveness of our method via three usage scenarios. When only optimizing dataflow, we achieve an average of **5**× improvement on PPA compared with random search in SparseLoop. By co-optimization of dataflow and hardware resource allocation, we achieve an average of **10**× improvement on PPA. By further enabling the determination of SAFs, we achieve an additional **3**× improvement on PPA.

II. Preliminaries

A. CNN Hardware Architecture and Resource Allocation

A typical CNN hardware consists of a 2D (or 1D) array of Processing Elements (PEs), each with a MAC unit and register file. Data storage involves four hierarchical levels: DRAM, global buffer, PE array, and register files. A key hardware resource allocation decision is **buffering level** [15], which is the loop level for certain data to be stored and reused at a specific memory level. Taking the buffering level for weight

979-8-3503-4859-0/DATE24/© 2024 EDAA

data in the global buffer as an example, if it's set at loop L_{13} as depicted in Listing 2, all weight data necessary for loop L_{13} are stored at global buffer for reuse.

B. Dataflow in CNN Hardware Computation

A typical CNN computation kernel is a set of nested loops as shown in Listing 1. Dataflow means how input, weight, and output data are navigated through the memory hierarchy and the PE array. It includes three operations [10].

1) **Spatial unrolling**: This is almost the same as conventional loop unrolling, except that the unrolling factor must equal the number of PEs.
2) **Loop tiling**: This is the same as conventional loop tiling in compiler designs. Note that loop tiling also affects hardware resource allocation since the **loop tile boundaries**, together with **buffering level**, decide the sizes of the global buffer, PE array, and register files.
3) **Loop ordering**: Different loop order changes computation efficiency without affecting computing results.

```
for m in range(M):          <- Filter iterations
  for c in range(C):        <- Channel iterations
    for i in range(I):      <- I,J: In/Out dimensions
      for j in range(J):
        for y in range(Y): <- Y,X: Filter dimensions
          for x in range(X):
            Output[m][i][j] +=
              Weight[m][c][y][x] *
              Input[c][i*S+y][j*S+x] <- S: stride
```

Listing 1. CNN computation kernel.

The work of [16] states that the impact of dataflow is small. However, the dataflow in [16] mostly refers to different stationary options (corresponding to spatial unrolling in this work) and has not considered other dataflow operations or hardware decisions.

C. Sparsity Patterns

CNN sparsity occurs in network weights through pruning [17], [18], and in input features due to activation functions like ReLU. This work considers two sparsity patterns from SparseLoop [9].

1) Fixed-density: Data groups have a fixed number of non-zero elements with the same density. For example, groups of size 32 in a row of the input array contain 8 non-zero elements each, randomly positioned. The number of non-zero elements in loop tiles may vary due to tile boundaries, but as the tile size increases, the variation reduces gradually.

2) Binomial distribution: Each data entry can be treated as an independent Bernoulli trial for being non-zero with probability p. If there are total n data entries, all of them share the same p. This follows binomial distribution $B(n, p)$. SparseLoop [9] used hypergeometric distribution, which is similar. Please note that the p here is different from the density of non-zero elements in the fixed-density pattern, where the densities of the two loop tiles are usually similar to each other. By contrast, there is a systematic and significant variance for non-zero densities in a binomial distribution pattern.

D. Sparse Acceleration Features (SAFs)

1) Sparse data representation format: Sparse data representation keeps only non-zero elements and their locations in a matrix, reducing storage footprint and data traffic. Our work supports commonly used sparse formats, e.g., compressed sparse row (CSR) [19], coordinate payload [9], run length encoding [9] and bitmask [9].

2) Gating and Skipping: Gating and skipping are methods to reduce computations in sparse CNNs. Gating gates PEs when encountering zero multiplications, saving energy without affecting latency. Skipping avoids sending zero multiplications to the PE array, reducing both energy consumption and computation latency but introduces overhead for tracing non-zero elements.

Taking the first layer of VGG16 as an example, the magnitude of the joint space of loop ordering (10^8), loop tiling (10^7), spatial unrolling (10^1), buffering level (10^4), and SAFs (10^1) is on the order of $10^{8+7+1+4+1} = 10^{21}$. Consequently, an efficient search algorithm is imperative.

III. Problem Formulation and Overview

```
      #DRAM level
L19:  for m3 in mD
          ...
L14:  for i3 in iD
      #Global buffer level
L13:  for i2 in iG     <--Weight buffering level
          ...
L8:   for m2 in mG
      #Spatial unrolling
L7:   for m1 in mS
L6:   for i1 in iS
      #Register file level
L5:   for m0 in mR     <--Weight buffering Level
          ...
L1:   for i0 in iR
      Output [m0] [i0] [j0]+=
             Weight [m0] [c0] [y0] [x0]*
                 Input [c0] [i0*S+y0] [j0*S+x0]
```

Listing 2. Example of loops tiling, ordering and spatial unrolling.

We study the co-optimization of hardware architecture and dataflow for sparse CNN computation in a 2D (or 1D) PE array and three memory levels: DRAM (D), global buffer (G), and register file (R). The decision variables are defined as follows.

- *Loop order & buffering level vector* $\boldsymbol{o}_D = [\pi_1^D, \pi_2^D, \ldots, \pi_{k_D}^D]$, $\boldsymbol{o}_G = [\pi_1^G, \pi_2^G, \ldots, \pi_{k_G}^G]$ and $\boldsymbol{o}_R = [\pi_1^R, \pi_2^R, \ldots, \pi_{k_R}^R]$. Each is a one-hot vector indicating which combination of loop order and buffering level is selected for a memory level among D, G, and R. We use $\boldsymbol{O}$ to denote the vector combining $\boldsymbol{o}_D$, $\boldsymbol{o}_G$ and $\boldsymbol{o}_R$.
- *Tile boundary vector* $\boldsymbol{m} = [m_D, m_G, m_R, m_S]$, where m_D, m_G, m_R, and m_S are the loop tile boundaries at memory levels D, G and R, and spatial unrolling, respectively. Tile boundary vectors for dimensions C, I, J, Y, and X are defined similarly.
- *Sparse acceleration features vector* $\boldsymbol{f} = [\alpha_1, \alpha_2, \ldots,]$, which is a one-hot vector indicating the choice of SAFs.

Given a sparse CNN layer instance λ, our method decides the buffering levels, dataflow, and SAFs, which are collectively denoted as $\Theta \triangleq \{\boldsymbol{O}, \boldsymbol{f}, \boldsymbol{m}, \boldsymbol{c}, \boldsymbol{i}, \boldsymbol{j}, \boldsymbol{y}, \boldsymbol{x}\}$, to minimize an integrated objective function Ψ of inference latency T, power P and area A. The corresponding formulation is given below.

$$\min_{\Theta} \Psi(\lambda; \Theta) = [T(\lambda; \Theta), P(\lambda; \Theta), A(\lambda; \Theta)], \quad (1)$$

subject to

$$b_D b_G b_R b_S = B, \text{for } B/b = M/m, C/c, I/i, J/j, Y/y, X/x, \quad (2)$$

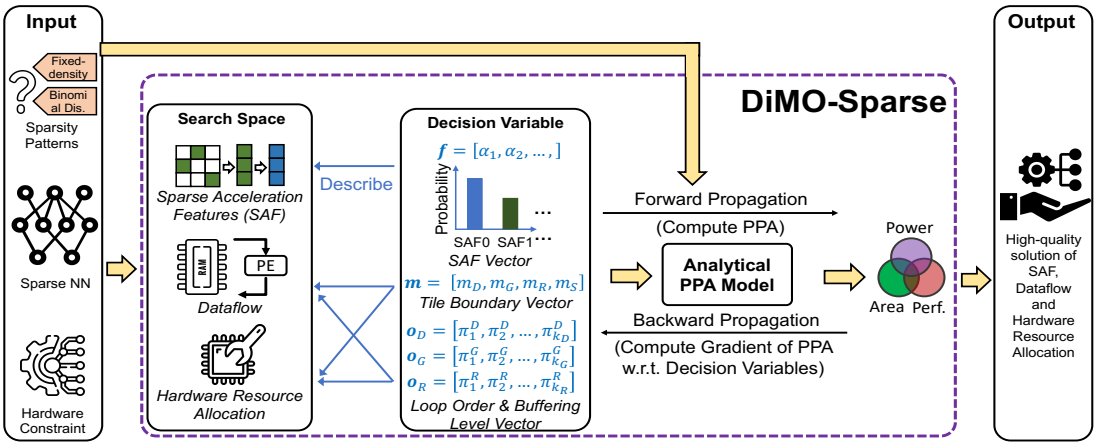

Figure 1. Overview of DiMO-Sparse. DiMO-Sparse automatically produces a high-quality solution of SAF, dataflow, and hardware resource allocation for a given sparse CNN and hardware constraints. Central to DiMO-Sparse is a differentiable PPA model, allowing for precise gradient calculations with respect to decision variables. These variables are relaxed to continuous, enabling an efficient gradient-based solver.

$$m_D, m_G, m_R, m_S, c_D, c_G, \dots, x_D, x_G, x_R, x_S \in \mathbb{N}_+, \quad (3)$$

$$\boldsymbol{O} \text{ and } \boldsymbol{f} \text{ are one-hot vectors,} \quad (4)$$

At most two in $[m_S, c_S, i_S, j_S, x_S, y_S]$ greater than 1. (5)

Here $T(*), P(*), A(*)$ are the mapping from design parameters and sparse workloads to inference latency, power, and area, respectively. Constraints (2) ensure that the tiling solution is legal. Constraint (4) means that only one loop order, buffering level, and sparse acceleration feature option can be chosen. Constraint (5) enforces that no more than two CNN dimensions are selected for spatial unrolling. Note that selecting only one dimension for spatial unrolling (e.g., setting $m_s = 1$ in Listing 2) results in a 1D PE array accelerator. Hence, our problem formulation incorporates 1D and 2D PE arrays. It is a multi-objective optimization problem, and our overall objective is to find a Pareto optimal solution.

Notably, our formulation differs from [12] by (1) incorporating SAF selection and (2) considering the impact of sparse workloads in the PPA model $T(*), P(*)$ and $A(*)$.

IV. DiMO-Sparse PPA Model

A. Overview of the Model Structure

DiMO-Sparse centers around a differentiable PPA model, illustrated in Figure 1. The key of PPA modeling is the estimate of global buffer size B_Δ^G, register file sizes B_Δ^R, PE array size Π and data traffic $\Phi_\Delta^{\eta \leftrightarrow \eta'}$ between two adjacent memory levels η and η', for data type $\Delta \in \{Input, Weight, Output\}$. This estimate is obtained through three steps: (1) A baseline estimate based on dense CNN design is obtained; (2) An estimate for sparse payload related to non-zero data is derived based on stochastic techniques; (3) Adjusting the estimate by accounting for the overhead related to sparse data representation. Once the 3-step estimate is completed, the calculation of PPA is straightforward [12]. The baseline estimate is based on sparse CNN hardware designs and reuses the results in [12]. We use the notation $\tilde{*}$ in the following sections to indicate values from baseline estimation.

B. Estimate for Sparse Payload

The PE array size Π of sparse CNN hardware is the same as that of dense CNN. In practice, sparsity does not affect spatial unrolling; each PE still gets the same number of tiles, but it is very unlikely all of those tiles are empty tiles. Therefore, the baseline estimate can be used without change. The buffer size and data traffic estimate for sparse payload (non-zero data) are obtained for two different sparsity patterns: fixed-density and binominal distribution.

1) Sparsity with fixed-density: Suppose the density of non-zero elements is $\gamma \in (0, 1)$. The baseline buffer size and traffic estimates are scaled by γ to obtain the estimate for a sparse payload of fixed-density pattern.

2) Sparsity with binomial distribution: Since the number of non-zero elements in a loop tile is a random variable for sparsity with a binomial distribution, the estimate is significantly more complicated than the case of fixed-density. The probability of the binomial distribution is denoted by β.

For a buffer $\eta \in \{G, R\}$, where G indicates global buffer and R represents register files, of data type Δ, we derive the minimum size that can accommodate the loop tile with the maximum number of non-zero elements. Assume there are τ tiles and each tile has n elements including zeros and non-zeros. The number of non-zero elements in the tile is a random variable z, and we define two lemmas as below to calculate the buffer size $\hat{B}$.

Lemma 1: If z is a random variable following binomial distribution $B(n, \beta)$, then for a sufficiently large n, z has a normal distribution with mean $n\beta$ and variance $n\beta(1-\beta)$ [20].

Lemma 2: If z is a random variable with normal distribution with mean μ and variance VAR, then the solution for $F_{z,norm}(Z) \geq c$ can be written as $Z = \mu + g \cdot \sqrt{VAR}$, where g only depends on c [21], where c is the confidence level, and we set $c = 99\%$ in this work.

From lemma 1, the cumulative distribution function of the binomial distribution for τ tries can be approximated by

$$F_{z,norm}(\hat{B})^\tau \geq c, \quad (6)$$

The Equation 6 can be transformed to

$$F_{z,norm}(\hat{B}) \geq e^{\frac{\ln c}{\tau}} \quad (7)$$

From Lemma 2, the solution of Equation (7) is given by

$$\hat{B} = \mu(z) + g \cdot \sqrt{Var(z)}, \quad (8)$$

where g is a coefficient depending on τ. We empirically find that g is almost a linear function of $\ln \tau$ and calculate g as

$$g = a \cdot \ln(\tau) + b, \quad (9)$$

where a and b are constants obtained by curve fitting for a given confidence level c. Overall, the required buffer size $\hat{B}$

can be analytically calculated as

$$\hat{B} = n\beta + (a \cdot \ln(\tau) + b)\sqrt{n\beta(1-\beta)} \quad (10)$$

The data traffic $\hat{\Phi}$ estimate includes the average case and the worst case. The average case estimate is the same as the fixed-density pattern, where the baseline traffic is scaled by β and applied in the power model. The worst case estimate is similar to the buffer size estimate and applied in the latency model.

C. Overall Buffer Size and Traffic Estimate with Overhead

Besides the payload of non-zero data, sparse data representations have the overhead of storing the locations of non-zero elements. The overhead is estimated by scaling the baseline estimate. Hence, the buffer overhead is estimated as

$$\delta B_{\Delta}^{G}(\alpha) = \rho(\alpha) \cdot \delta \tilde{B}_{\Delta}^{G} \quad (11)$$

where α indicates a sparse data representation format and ρ is an empirical parameter. Please note that register files only store decoded data and thus do not have such overhead. Likewise, the data traffic overhead between DRAM and the global buffer is estimated by

$$\delta \Phi_{\Delta}^{D \leftrightarrow G}(\alpha) = \omega(\alpha) \cdot \delta \tilde{\Phi}_{\Delta}^{D \leftrightarrow G} \quad (12)$$

where ω is an empirical parameter depending on the data format. The overall estimate are:

$$B_{\Delta}^{G} = \hat{B}_{\Delta}^{G} + \delta B_{\Delta}^{G}, \quad B_{\Delta}^{R} = \hat{B}_{\Delta}^{R} \quad (13)$$

$$\Phi_{\Delta}^{D \leftrightarrow G} = \hat{\Phi}_{\Delta}^{D \leftrightarrow G} + \delta \Phi_{\Delta}^{D \leftrightarrow G}, \quad \Phi_{\Delta}^{G \leftrightarrow R} = \hat{\Phi}_{\Delta}^{G \leftrightarrow R} \quad (14)$$

D. The PPA Model

For PPA estimation, the area is the sum of the global buffer area, register file areas, and PE areas. The power estimate consists of those associated with PEs and memory (DRAM, global buffer, and register file) access. The PE power is the product between the total number of MAC (Multiply and Accumulate) operations and the power per MAC operation. For the sparsity pattern of the binomial distribution, the average case traffic estimate is employed in the power model, and the worst case traffic estimate is used in the inference latency model. The memory access power is the product between data traffic Φ_{Δ} from our estimate and the power per access from lookup tables. The power estimate of memory access at different levels and MAC operations can be found in [12], [16]. The total latency is estimated by the total number of MAC operations divided by the number of PEs. The global buffer size is sufficiently large, so the PE array can be continuously fed with data without waiting for DRAM access. As such, the inference latency is dominated by the latency of PE computations.

In addition, we perform three adjustments. (1) Small constant PPA overhead is added for the circuits decoding/encoding the sparse data representation. (2) If gating is applied, the power associated with PEs in the baseline estimate is scaled down by an empirical factor. (3) If skipping is employed, PE power and inference latency are scaled down similarly.

V. DiMO-Sparse: Optimization

The differentiable PPA model described in Section IV is integrated into the optimization framework [12] with two remarkable enhancements. First, our DiMO-Sparse optimization supports sparse CNN dataflow and hardware architecture, while the work of [12] is restricted to dense CNN computations. Second, the optimization of [12] is for individual CNN layers, while our DiMO-Sparse can optimize for an entire CNN. Our optimization includes three parts: (1) continuous optimization, (2) massively parallel discretization, and (3) network-wise optimization.

A. Continuous Optimization

A key ingredient in this part is to separate the decision variables and describe the PPA models in a matrix-vector product form using the *Ln-Exp* trick [12]. For example, a buffer size can be expressed as

$$B_{\Delta}^{\eta} = \boldsymbol{s}^{T} \cdot \boldsymbol{Q} \cdot \boldsymbol{u} \quad (15)$$

where $\boldsymbol{s}$ is the selection variable vector, $\boldsymbol{Q}$ is the query matrix and $\boldsymbol{u}$ is the vector of tile boundary variables. $\boldsymbol{Q}$ is a 0-1 constant matrix with each row corresponding to one choice of buffering level, loop order, etc. The rows of $\boldsymbol{Q}$ are pre-generated offline through enumeration. Since the number of loop levels is limited and a divide-and-conquer strategy [12] is applied, the number of rows in $\boldsymbol{Q}$ does not explode. $\boldsymbol{s}$ is a 0-1 variable vector for selecting a row in $\boldsymbol{Q}$.

The problem in Section III involves relaxed decision variables in selection vectors $\boldsymbol{s}$ and loop tile boundary vectors $\boldsymbol{u}$, which allows fractional values. This non-linear programming problem can be solved using off-the-shelf solvers. Please note our differentiable models play a critical role in allowing gradient-based search to efficiently find high-quality solutions. Using the matrix-vector product form, we leverage deep learning toolkits like PyTorch for quick solutions. In PyTorch, the objective function from Section III becomes the loss function and solves it by CNN training. The tradeoff among latency, power, and area is managed through multi-task learning [22].

B. Massively Parallel Discretization

The fractional solution obtained from the non-linear programming is discretized into integer solutions. For the selection vector $\boldsymbol{s}$ solutions, we find the entries with top k maximum values and keep only the rows in $\boldsymbol{Q}$ corresponding to these k entries to have a reduced query matrix $\boldsymbol{Q}^{*}$. For a fractional tile boundary solution $\boldsymbol{u}$, we enumerate its nearby integer solutions, which form a constant solution matrix $\boldsymbol{U}$. Then, numerous buffer size solutions can be simultaneously searched in the form

$$\boldsymbol{B}_{\Delta}^{\eta} = \boldsymbol{Q}^{*} \cdot \boldsymbol{U} \quad (16)$$

where $\boldsymbol{B}_{\Delta}^{\eta}$ is a vector of integer solutions, from which we can pick the best solution.

C. Network-wise Optimization

The network-wise optimization consists of two phases. In Phase 1, each individual layer of a given CNN is optimized separately using the optimization method described in Sections V-A and V-B. Then, the maximum hardware resource values, including global buffer size, the PE array size, and register file size, among all layers and the corresponding SAFs, are identified and will be used as constraints in Phase 2. In Phase 2, dataflow optimization is performed for all layers under the hardware and SAF constraints.

VI. Evaluation

We evaluate DiMO-Sparse on three well-known CNN workloads: VGG16, ResNet18, and AlexNet. It is implemented in Python and can run on CPUs or GPUs. For a fair comparison with SparseLoop, all experiments are conducted on one core of an AMD Ryzen Threadripper 1920X 12-Core Processor.

SparseLoop is a widely recognized model for sparse CNN hardware but is not differentiable. It includes a primitive optimizer based on random search, the only previous work on automatic optimization of dataflow and hardware architecture for sparse CNN computation. SparseLoop also has an optimizer based on exhaustive search, but it is impractical due to its long runtime (more than three days per CNN layer). Sections VI-A, VI-B , and VI-C use sparsity ratio of 10% of non-zero values for both weight and input arrays.

A. DiMO-Sparse Model Accuracy

DiMO-Sparse is compared with SparseLoop on over 1000 legal solutions of dataflow and hardware architectures in two sparsity patterns with various SAFs. Results in Tables I and II show average relative error and RMSE. The RF and GB sizes are measured by the size required to store non-zero values. The overhead is extra storage space required for sparse data representation. The power and latency are measured as total power and maximum latency per solution. DiMO-Sparse closely matches SparseLoop estimates on RF size, GB size, and inference latency. Errors on power and sparse data representation overhead estimates are at most 3%. And DiMO-Sparse achieves a 20K speedup over SparseLoop.

Table I
DIMO-SPARSE MODEL ERROR VS. SPARSELOOP FOR FIXED-DENSITY PATTERN.

	RF Size	GB Size	Overhead	Power	Latency
Ave Err	0%	0%	0.1%	0.1%	0%
RMSE	0%	0%	1%	1%	0%

Table II
DIMO-SPARSE MODEL ERROR VS. SPARSELOOP FOR BINOMIAL DISTRIBUTION PATTERN.

	RF Size	GB Size	Overhead	Power	Latency
Ave Err	0%	0%	3%	0.3%	0%
RMSE	0%	0%	1.6%	0.6%	0%

B. Layer-wise Dataflow Optimization with Fixed Hardware

In this experiment, we evaluate the effectiveness of our DiMO-Sparse for per-layer dataflow optimization with fixed hardware architectures. The hardware architecture designs include global buffer size, PE array size, register file size, RLE for sparse data representation, and gating for the SAF. Dataflow optimization focuses on loop order and tiling, benefiting CNN accelerators' compiler designs.

Figures 2 and 3 show the normalized results for the fixed-density pattern and binomial distribution pattern, respectively. The results represent the average among all layers of each network. It's important to note that all latency and power results are based on the SparseLoop model despite our DiMO-Sparse optimization being guided by our DiMO-Sparse model. As the hardware resources are fixed in this experiment, both the SparseLoop optimizer and our DiMO-Sparse optimizer utilize the same hardware area. The comparison clearly demonstrates that our DiMO-Sparse optimization outperforms SparseLoop optimization across all workloads in terms of inference latency, power consumption, and optimization runtime. For example, for VGG16 with the fixed-density pattern, DiMO-Sparse achieves **77**% reduction in power consumption, **57**% decrease in inference latency, and **6.3**× speedup in optimization runtime compared to SparseLoop optimization. For ResNet18 with the binomial distribution pattern, DiMO-Sparse obtains **77**% reduction in power consumption, a **41**% decrease in inference latency, and a notable **6**× reduction in optimization runtime.

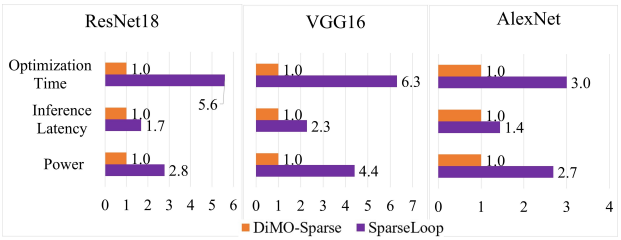

Figure 2. Dataflow optimization with fixed-density pattern.

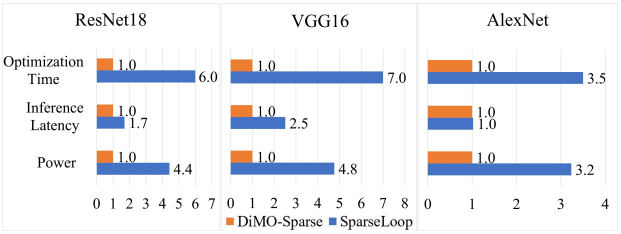

Figure 3. Dataflow optimization with binomial distribution pattern.

C. Network-wise Optimization of Hardware Architecture and Dataflow

In this experiment, we jointly optimize hardware architecture and dataflow for entire CNNs with fixed SAFs (RLE and gating) for application-specific CNN hardware design with consideration of optimized dataflow mapping.

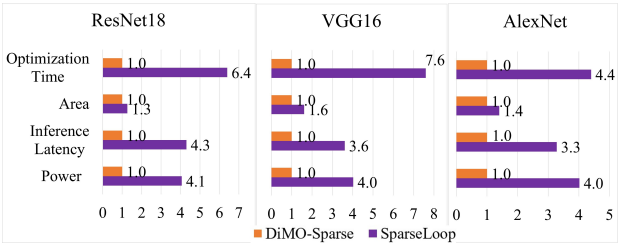

Figure 4. Network-wise optimization of hardware architecture and dataflow with fixed-density pattern.

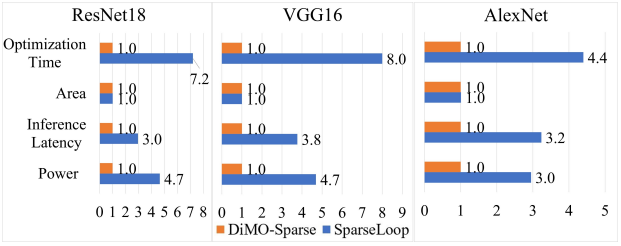

Figure 5. Network-wise optimization of hardware architecture and dataflow with binomial distribution pattern.

Figure 4 shows the normalized results for the fixed-density pattern. PPA estimations of optimization results are based on the SparseLoop model. DiMO-Sparse outperforms SparseLoop optimization with **75**% lower power consumption, **77**% reduced inference latency, **23**% area reduction, and **6**× faster optimization runtime for ResNet18.

Similarly, Figure 5 illustrates the normalized results for the binomial distribution pattern. DiMO-Sparse attains remarkable advantages over SparseLoop. For VGG16, it achieves an impressive **78**% reduction in power consumption, **74**% decrease

in inference latency, and a remarkable **8**× reduction in optimization runtime with the same area.

D. Dataflow, Hardware Architecture, and SAF Co-optimization

In this part of the experiment, we evaluate network-wise DiMO-Sparse for simultaneous dataflow, hardware architecture, and SAF optimization. The sparse ratio is the same for weight and input arrays. SparseLoop optimizer does not support SAF optimization, and its model has limited support for evaluating a variety of SAFs. Therefore, we compare the following variants of DiMO-Sparse applications, and the PPA results are calculated using our DiMO-Sparse model, whose accuracy has been validated in Section VI-A.

- SAF_1: DiMO-Sparse optimization of dataflow and hardware architecture with CSR data format and skipping.
- SAF_2: DiMO-Sparse optimization of dataflow and hardware architecture with CSR data format and gating.
- SAF_1_2: DiMO-Sparse optimization of dataflow and hardware architecture, with simultaneous selection between SAF_1 and SAF_2.
- SAF_All: DiMO-Sparse optimization of dataflow and hardware architecture, with simultaneous SAF selection among all SAF options.

For the SAF_1_2 and SAF_All results, all layers of a network share the same SAF.

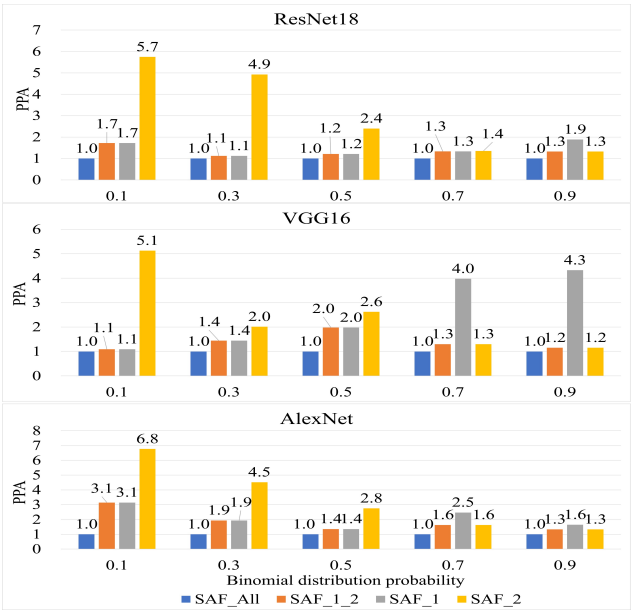

Figure 6. Network-wise DiMO-Sparse optimization of dataflow, hardware architecture, and SAFs. The X-axis represents the ratio of non-zero values.

Figure 6 plots the normalized PPA results of these variants for different binomial distribution probabilities. The PPA here is the product of power, inference latency, and area. SAF_1 produces inferior solutions when the probability of non-zero elements is high. This is because the high density of non-zero elements allows very little latency/power reduction while the overhead of skipping becomes conspicuous. SAF_2 is not good at handling the case with low probability because gating does not reduce latency, which can be largely decreased by skipping. SAF_1_2 achieves an average of **1.3**× and **1.9**× PPA improvement compared to SAF_1 and SAF_2, respectively. SAF_All consistently outperforms all the other variants in all cases. SAF_All achieves an average of **2**× and **3**× PPA improvement compared to SAF_1 and SAF_2, respectively.

VII. Conclusions

Sparse CNN hardware computation offers a big boost in performance, power, and area (PPA), but current methods to harness this potential are manual. We introduce DiMO-Sparse, the first differentiable PPA model for sparse CNN hardware and dataflow design. It also includes an efficient technique for optimizing dataflow, architecture, and sparsity acceleration features. Our experiments show that DiMO-Sparse outperforms previous work by $10\times$ times in PPA with faster runtime.

Acknowledgments

This work is supported by National Science Foundation (NSF) award CCF-1955909, CMMI-2038625, CCF-2106725.

References

[1] J. O. Neill, "An overview of neural network compression," *arXiv preprint arXiv:2006.03669*, 2020.

[2] S. Han and *et al.*, "Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding," in *ICLR*, 2016.

[3] F. N. Iandola and *et al.*, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and $<$ 0.5 mb model size," *arXiv preprint arXiv:1602.07360*, 2016.

[4] A. Parashar and *et al.*, "SCNN: An accelerator for compressed-sparse convolutional neural networks," in *ISCA*, 2017, pp. 27–40.

[5] Y.-H. Chen and *et al.*, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *ISCA*, 2016, pp. 367–379.

[6] K. Hegde and *et al.*, "Extensor: An accelerator for sparse tensor algebra," in *MICRO*, 2019, p. 319–333.

[7] A. Gondimalla and *et al.*, "SparTen: A sparse tensor accelerator for convolutional neural networks," in *MICRO*, 2019, p. 151–165.

[8] C. Deng and *et al.*, "GoSPA: An energy-efficient high-performance globally optimized sparse convolutional neural network accelerator," in *ISCA*, 2021, pp. 1110–1123.

[9] Y. N. Wu and *et al.*, "Sparseloop: An analytical, energy-focused design space exploration methodology for sparse tensor accelerators," in *ISPASS*, 2021, pp. 232–234.

[10] S.-C. Kao and T. Krishna, "GAMMA: Automating the hw mapping of dnn models on accelerators via genetic algorithm," in *ICCAD*, 2020, pp. 1–9.

[11] S. Kao and *et al.*, "ConfuciuX: Autonomous hardware resource assignment for DNN accelerators using reinforcement learning," in *MICRO*, 2020, pp. 622–636.

[12] R. Liang and *et al.*, "Deep learning toolkit-accelerated analytical co-optimization of cnn hardware and dataflow," in *ICCAD*, 2022, pp. 1–12.

[13] A. Parashar and *et al.*, "Timeloop: A systematic approach to dnn accelerator evaluation," in *ISPASS*, 2019, pp. 304–315.

[14] Y. Fu, Y. Zhang, Y. Zhang, D. Cox, and Y. Lin, "Auto-NBA: Efficient and *et al.*" in *ICML*, 2021, pp. 3505–3517.

[15] A. Stoutchinin and *et al.*, "Optimally scheduling cnn convolutions for efficient memory access," *arXiv preprint arXiv:1902.01492*, 2019.

[16] X. Yang and *et al.*, "Interstellar: Using halide's scheduling language to analyze dnn accelerators," in *ASPLOS*, 2020, p. 369–383.

[17] J. Luo and *et al.*, "Thinet: A filter level pruning method for deep neural network compression," in *ICCV*, 2017, pp. 5068–5076.

[18] M. A. Carreira-Perpinan and Y. Idelbayev, ""learning-compression" algorithms for neural net pruning," in *CVPR*, 2018, pp. 8532–8541.

[19] S. Chou and *et al.*, "Format abstraction for sparse tensor algebra compilers," *The ACM on Programming Languages*, pp. 1–30, 2018.

[20] W. Feller, "On the Normal Approximation to the Binomial Distribution," *The Annals of Mathematical Statistics*, pp. 319–329, 1945.

[21] M. J. P. Selby, "Handbook of the Normal Distribution," *Royal Statistical Society. Journal. Series A: General*, pp. 95–95, 2018.

[22] O. Sener and V. Koltun, "Multi-task learning as multi-objective optimization," in *NIPS*, 2018, p. 525–536.