Deep Learning Toolkit-Accelerated Analytical Co-optimization of CNN Hardware and Dataflow

Rongjian Liang*
Nvdia
rliang@nvidia.com

Jianfeng Song
Department of ECE
Texas A&M University
jsong26@tamu.edu

Yuan Bo
Department of ECE
Rutgers University
bo.yuan@soe.rutgers.edu

Jiang Hu
Department of ECE & CSE
Texas A&M University
jianghu@tamu.edu

ABSTRACT

The continuous growth of CNN complexity not only intensifies the need for hardware acceleration but also presents a huge challenge. That is, the solution space for CNN hardware design and dataflow mapping becomes enormously large besides the fact that it is discrete and lacks a well behaved structure. Most previous works either are stochastic metaheuristics, such as genetic algorithm, which are typically very slow for solving large problems, or rely on expensive sampling, e.g., Gumbel Softmax-based differentiable optimization and Bayesian optimization. We propose an analytical model for evaluating power and performance of CNN hardware design and dataflow solutions. Based on this model, we introduce a co-optimization method consisting of nonlinear programming and parallel local search. A key innovation in this model is its matrix form, which enables the use of deep learning toolkit for highly efficient computations of power/performance values and gradients in the optimization. In handling power-performance tradeoff, our method can lead to better solutions than minimizing a weighted sum of power and latency.

The average relative error of our model compared with Timeloop is as small as 1%. Compared to state-of-the-art methods, our approach achieves solutions with up to **1.7 × shorter inference latency**, **37.5% less power consumption**, and **3 × less area** on ResNet 18. Moreover, it provides a **6.2 × speedup of optimization runtime**.

CCS CONCEPTS

• **Hardware → Hardware accelerators**.

KEYWORDS

CNN accelerator, CNN dataflow, CNN compiler optimization

ACM Reference Format:
Rongjian Liang, Jianfeng Song, Yuan Bo, and Jiang Hu. 2022. Deep Learning Toolkit-Accelerated Analytical Co-optimization of CNN Hardware and Dataflow. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD '22), October 30-November 3, 2022, San Diego, CA, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3508352.3549402

*Work performed while studying at the Texas A&M University.

ICCAD '22, October 30-November 3, 2022, San Diego, CA, USA

ACM ISBN 978-1-4503-9217-4/22/10…$15.00
https://doi.org/10.1145/3508352.3549402

1 INTRODUCTION

Driven by market demands, CNN complexity has reached the order of billions of parameters [10] and continues to grow. This trend not only increases the need for customized CNN hardware acceleration but also makes the accelerator design a huge challenge. In addition, the performance and power of CNN computing heavily depend on how to map a dataflow onto a given hardware, thereby making the combined solution space for the hardware design and dataflow mapping become enormously large. Moreover, because the solution space is discrete as well as lacking a well-behaved structure, it is quite challenging to find efficient solution search schemes. A generic combinatorial approach like genetic algorithm [15] may entail hours of computation time for a large network. In system-level development, where a variety of high-level decisions need to be carefully assessed through optimization, such computation cost is painfully expensive.

Existing Works and Limitations. Due to the importance of CNN accelerator design and its challenge, there have been numerous related studies published in recent literature [2, 6, 8, 9, 13, 15, 17, 20, 27, 28, 30, 32, 33]. Genetic algorithm [15] works well for dataflow mapping but is difficult to scale when the hardware is simultaneously optimized. Gumbel Softmax-based differentiable optimization [6] relies on expensive sampling and is vulnerable to large solution variations. Neural network-based CNN accelerator performance model [3] is also differentiable. However, its model training cost is very expensive. An analytical model is proposed in [32], but it comes with hidden "if-else" operations, which degrade the efficiency of its gradient computation. NAS (Neural Architecture Search) and accelerator co-design [1, 3, 6, 12, 18, 30, 33] usually simplify the formulation of its hardware design or dataflow component in order to restrict solution space.

Our goal is to develop a CNN hardware and dataflow co-optimization method that can attain significantly better solutions and much faster computation speed than existing approaches. We achieve this goal through a new modeling technique and a new optimization approach that dovetail with each other.

Our first key contribution is an analytical performance model for CNN hardware and dataflow under a given workload. Its novelty lies in two advantages compared to the previous analytical model [32]:

(1) We propose a one-time offline computation technique that can help completely avoid any "if-else" operations, which are present in the model of [32].
(2) We design the model to be a matrix form that enables the use of deep learning toolkit, such as PyTorch, for highly efficient computation of power/performance values and gradients.

The second main contribution is a two-stage optimization scheme. In stage I, the problem is relaxed to continuous non-linear optimization. It is then cast as neural network training and solved through

a deep learning toolkit with parallel computing and automatic gradient propagation. Stage II is to obtain an integer solution through parallel local search. Our model facilitates not only efficient computing in stage I but also a large search region in stage II for better solution quality. Our method is a multi-objective optimization that leads to Pareto front solutions for the power-performance tradeoff.

To evaluate the accuracy of our model, we compared the HW performance value from our model with Timeloop and achieved an average 1% relative error. Experiments are performed on three widely used CNN designs. Compared to state-of-the-art methods, our approach achieves solutions with 1.7× shorter inference latency, 37.5% less power consumption, and 3× less area on ResNet 18. It also provides a maximum of 6.2× speedup of optimization runtime. Moreover, it can obtain a better power-performance tradeoff than minimizing a weighted sum of power and latency.

2 PREVIOUS METHODS

Genetic algorithm is a general framework for combinatorial optimization and is applied for CNN dataflow optimization in GAMMA [15]. It is well known that genetic algorithm tends to be very slow in solving large problems. Although it works well for dataflow mapping, its runtime would increase remarkably for CNN hardware-dataflow co-optimization. Bayesian optimization (BO) is another general framework and is applied in DNN accelerator co-design [24]. BO requires numerous sampling of a raw model for the objective function. Usually, such a raw model is complicated, and its sampling becomes a bottleneck for runtime. As such, the work of [24] optimizes only a small subset of hardware and dataflow options. Recently, there is a trend of NAS (Neural Architecture Search) and accelerator co-design [3, 6, 30, 33]. Arguably, NAS-accelerator co-design is a superset of CNN hardware-dataflow co-optimization. However, the co-design space is so huge that existing approaches have to simplify their hardware/dataflow components. For instance, the co-design in [33] skips dataflow optimization, and the work of [30] is restricted to a limited number of hardware templates.

In AutoNBA [6], the objective function is made differentiable through Gumbel Softmax [11] so that gradient-based search is enabled. However, Gumbel Softmax requires sampling of a raw model, which can be expensive. Moreover, this approach is vulnerable to large solution variations. The work of DANCE [3] employs a neural network-based model to facilitate differentiability. However, its training data generation is very expensive and thus offsets its advantage in optimization. Timeloop [20] and Maestro [16] provide models for evaluating DNN hardware and dataflow, such as latency and power, but they are neither differentiable nor analytical. The optimizations in Timeloop [20] are exhaustive search and random sampling. An analytical performance model for DNN accelerators is introduced in [32].

Other related works include [1, 4] for FPGAs, Sparseloop [5, 7, 21, 29] for sparse tensors, and [22] for memristor-based accelerators. Also, our optimization method is inspired by Dreamplace [19], which accelerates VLSI placement with deep learning toolkits.

3 PRELIMINARIES

3.1 CNN Convolution Layer

A generic CNN convolution layer can be expressed as a 6-level loop nest, as shown in Listing 1. Related symbols and their meaning are listed in Table 1. Note that the fully-connected layer can be viewed as a special type of CNN layer.

```
for m in range(M):
 for c in range(C):
  for i in range(I):
   for j in range(J):
    for y in range(Y):
     for x in range(X):
      Output[m][i][j] +=
        Weight[m][c][y][x]*Input[c][i*S+y][j*S+x]
```

Listing 1: CNN example

Table 1: Symbols used in the paper

Symbol	Meaning
IN	input feature maps
OUT	output feature maps
W	convolution kernels
M	number of output feature maps
C	number of input feature maps
I, J	height and width of output feature map
Y, X	height and width of convolution kernel
S	convolution stride
$ln(\star)$	logarithmic operation
$exp(\star)$	exponential operation

3.2 Hardware Resource Allocation and Dataflow

The architecture of CNN accelerators depends on two key components: hardware resource allocation and dataflow strategy.

3.2.1 Hardware resource allocation. Spatial CNN accelerators contain an array of Processing Elements (PE), each of which has a MAC to compute the partial sum and local buffers, called Register File (RF), to store IN, OUT, and W data. The IN, OUT, and W data can share one RF or they can have an individual RF. The accelerators typically also house a global shared buffer (GB) to prefetch data from external DRAM. Again, the IN, OUT, and W data might also have their individual global buffer. Networks-on-chip are used to transfer data among PEs and between PEs and the global buffer. The hardware resource allocation between on-chip compute (MAC) and on-chip buffers, as well as the allocation among different buffer components, have a significant impact on the accelerators' performance [14].

3.2.2 Dataflow strategy. It is mainly determined by three components. **1).** Loop order. Different data reuse opportunities can be utilized by swapping the order of loops in Listing 1. **2).** Spatial unrolling, which consists of the number of levels of parallelism and the specific loops selected for spatial unroll. **3).** Tiling, which refers to determining the tiling size for each CNN dimension.

3.2.3 Relationship between hardware resource allocation, dataflow strategy, and hardware performance. The latency of CNN inference is determined by the computation time and the data transfer time. The computation time is affected by the spatial unrolling and constrained by the size of the PE array. The data transfer time is affected by all three dataflow components and limited by the data transfer bandwidth. The total power comprises of the computation power and the data transfer power, which mainly depends on the dataflow

strategy and hardware resource allocation. The area is decided by the hardware resource allocation.

3.3 Footprint, Buffering Level, Buffer Requirement and Traffic

The concept of *footprint* is introduced in [26] to denote the number of distinct elements in *IN*, *OUT* or *W* used inside a specific loop. The footprint at loop level l can be computed as follows.

$$F_{IN}(L_l) = c(l)[(i(l)-1)*S+y(l)][(j(l)-1)*S+x(l)], \quad (1)$$

$$F_{OUT}(L_l) = m(l)i(l)j(l), \quad (2)$$

$$F_{W}(L_l) = m(l)c(l)y(l)x(l), \quad (3)$$

where $F_{IN}(L_l)$, $F_{OUT}(L_l)$ and $F_W(L_l)$ are the footprint at loop level l for *IN*, *OUT* and *W*, respectively. And $m(l)$ is the product of all tiled loop boundaries associated with CNN layer dimension M from loop level 0 to loop level l. In the example in Listing 2, the $m(5) = m_R$ and $m(10) = m_R m_S$. $c(l), i(l), j(l), y(l)$ and $x(l)$ are defined in a similar way.

The concept of *buffering level* is also proposed in [26] to associate on-chip buffer requirements with footprint. Buffering level denotes a nested loop level at which data are buffered for reuse. Note that *IN*, *OUT*, and *W* data can have different buffering levels at each memory hierarchy. In the example in Listing 2, loop L_3 is selected as the buffering level for *IN* array at register file. Then the buffer requirement for *IN* at register file is the size of distinct elements in *IN* used in one iteration of loop L_3. The buffer size requirement when buffering at loop level l must be sufficient enough to store the total number of distinct elements that are used in one iteration of level l, and it can be computed as:

$$BS_X(L_l) = \begin{cases} P_X * F_X(L_{l-1}), & \text{if } l \geq 1; \\ P_X, & l = 0, \end{cases} \quad (4)$$

where $X = IN, OUT, W$, P_X[1] is the numerical precision (in bytes) used for its storage. Note that spatial unrolling loops (L_6 and L_7 in Listing 2) would not be selected as buffering levels.

The data transfer traffic between DRAM and global buffer when buffering at loop level l ($l \geq 8$) can be computed as:

$$TF_D^X(L_l) = P_X * F_X(L_l) * \prod_{j>l} b_j, \quad (5)$$

where b_j is the loop boundary at loop L_j.

The traffic between global buffer and register file when buffering at loop level L_l ($0 \leq l \leq 5$) can be computed as:

$$TF_G^X(L_l) = P_X * F_X(L_l) * \frac{b_6 b_7}{S_X} \prod_{j \geq 8} b_j, \quad (6)$$

where S_X is the number of PEs that share the same data.

3.4 Multi-Objective Optimization

We consider the latency, power, and area in the co-optimization of CNN hardware and dataflow. It is a multi-objective optimization problem. A common compromise is to optimize a linear combination of per-objective losses, called the weighted sum loss method in this paper. However, it typically does not perform well when the objectives conflict with each other (like in our case), and it is hard to find the proper weights for the objectives. A gradient-based neural network training technique that can optimize a collection of possibly conflicting objectives is proposed in [25]. Its network parameter update scheme is as follows. Firstly, the gradients of each objective w.r.t. the shared neural network parameters are computed. Then a minimum-norm vector in the convex hull of the set of gradient vectors is found. Finally, gradients are updated in the direction of the minimum-norm vector. It has been shown in [25] that such gradient update approach essentially optimizes an upper bound for the multi-objective loss.

[1]For simplicity but without loss of generality, we set P_X to 1 in this work.

```
      #DRAM Level
L19: for m3 in mD
L18: for c3 in cD
L17: for i3 in iD
L16: for j3 in jD
L15: for y3 in yD
L14: for x3 in xD
      #Global Buffer Level
L13: for m2 in mG
L12: for c2 in cG
L11: for i2 in iG     <--Weight and Output Buffering Level
L10: for j2 in jG
L9:  for y2 in yG     <--Input Buffering Level
L8:  for x2 in xG
      #Spatial Unrolling
L7:  for m1 in mS
L6:  for i1 in iS
      #Register File Level
L5:  for m0 in mR     <--Weight and Output Buffering Level
L4:  for c0 in cR
L3:  for i0 in iR     <--Input Buffering Level
L2:  for j0 in jR
L1:  for y0 in yR
L0:  for x0 in xR
      Output [m0] [i0] [j0]+=
             Weight [m0] [c0] [y0] [x0]*
                 Input [c0] [i0*S+y0] [j0*S+x0]
```

Listing 2: Example of design solution

4 PROBLEM FORMULATION

We study the co-optimization of hardware and dataflow for CNN accelerators with a 2D PE-array and three-level memory hierarchy, i.e., DRAM, GB, and RF.

The design parameters are defined as follows:

- *LoopOrder & BufferingLevel vectors* $\boldsymbol{o}_D = [\pi_1^D, \pi_2^D, \ldots, \pi_{k_D}^D]$, $\boldsymbol{o}_G = [\pi_1^G, \pi_2^G, \ldots, \pi_{k_G}^G]$ and $\boldsymbol{o}_R = [\pi_1^R, \pi_2^R, \ldots, \pi_{k_R}^R]$. They are one-hot vectors that indicate which loop order and buffering level are chosen in DRAM, GB, and RF, respectively. For instance, $\boldsymbol{o}_R = [0, 1, 0, \ldots, 0]$ means that the second combination of loop order and buffering level is selected for the RF loops, i.e., L_0-L_5 in Listing 2. k_D, k_G, k_R are the total number of combinations of loop order and buffering level at DRAM, GB and RF, respectively.
- Boundary vector $\boldsymbol{m} = [m_D, m_G, m_R, m_S]$, where m_D, m_G, m_R, and m_S are the tiled loop boundaries for M at DRAM, GB, RF and spatial unrolling, respectively, as shown in Listing 2. Similarly, we define loop boundaries for C, I, J, Y and X as $\boldsymbol{c} = [c_D, c_G, c_R, c_S]$, $\boldsymbol{i} = [i_D, i_G, i_R, i_S]$, $\boldsymbol{j} = [j_D, j_G, j_R, j_S]$, $\boldsymbol{x} = [x_D, x_G, x_R, x_S]$, and $\boldsymbol{y} = [y_D, y_G, y_R, y_S]$, respectively. CNN dimensions that are not selected for spatial unrolling have their spatial unrolling loop boundaries to be 1. In the

example of Listing 2, $c_S = j_S = y_S = x_S = 1$. In this way, the spatial unrolling is also encoded into the boundary vectors.

We let the GB size and RF size match their buffer requirements computed according to Equation (4). And the PE array size is equal to the product of the boundaries of spatial unrolling loops (e.g., $m_S * i_S$ in Listing 2. In this way, hardware resource allocation is also determined by our design parameters.

Given a CNN layer instance x with dimensions $[M, C, I, J, Y, X]$, the objective is to minimize the inference latency, power, and area by finding the best design parameters. The mathematical problem formulation is defined as follows:

$$\min_{\Theta \triangleq \{\boldsymbol{o}_D, \boldsymbol{o}_G, \boldsymbol{o}_R, \boldsymbol{m}, \boldsymbol{c}, \boldsymbol{i}, \boldsymbol{j}, \boldsymbol{r}, \boldsymbol{q}\}} HW(x; \Theta) = [T(\Theta), P(\Theta), A(\Theta)], \quad (7)$$

subject to

$$m_D m_G m_R m_S = M, \quad (8)$$

$$c_D c_G c_R c_S = C, \quad (9)$$

$$i_D i_G i_R i_S = I, \quad (10)$$

$$j_D j_G j_R j_S = J, \quad (11)$$

$$y_D y_G y_R y_S = Y, \quad (12)$$

$$x_D x_G x_R x_S = X, \quad (13)$$

$$m_D, m_G, m_R, m_S, c_D, c_G, \ldots, x_D, x_G, x_R, x_S \in \mathbb{N}_+, \quad (14)$$

$$\boldsymbol{o}_D, \boldsymbol{o}_G \text{ and } \boldsymbol{o}_R \text{ are one-hot vectors}, \quad (15)$$

No more than two from $[m_S, c_S, i_S, j_S, x_S, y_S]$are greater than 1. (16)

Here $T(*), P(*), A(*)$ are the mapping from design parameters to inference latency, power, and area. Constraints (8-14) ensure that the tiling solution is legal. Constraint (15) means that only one loop order and buffering level option can be chosen for each memory level. And constraint (16) enforces that no more than two CNN dimensions are selected for spatial unroll. It is a multi-objective optimization problem, and our overall objective is to find a Pareto optimal solution.

5 METHODOLOGY

5.1 Overview

Figure 1 depicts our CNN hardware dataflow co-optimization framework. The core is an analytical hardware performance model in a matrix form, which facilitates our two-stage optimization scheme. In stage I, the original problem (Equation 7) is relaxed into a non-linear programming problem, which is then been cast as neural network training and solved through a deep-learning toolkit with parallel computing and automatic gradient propagation. In stage II, an integer solution is found by parallel local search around the continuous solution output by non-linear optimization.

5.2 Analytical Hardware Performance Model in a Matrix Form

5.2.1 Ln-Exp Trick. We propose a *Ln-Exp* trick which is the key to our matrix-form model. Its main idea is illustrated via a few examples.

The multiplication of a set of scalars can be expressed as the inner product of two vectors. For example,

$$a_1 b_1 c_1 = \exp\langle \boldsymbol{q}, \boldsymbol{b} \rangle, \quad (17)$$

where $\boldsymbol{q} = [1, 1, 1]$, $\boldsymbol{b} = \ln[a_1, b_1, c_1]$ and $\langle *, * \rangle$ denotes inner product.

The *Ln-Exp* trick can transform a series of multiplication of scalars into a matrix multiplication operation. For instance,

$$\begin{bmatrix} a_1 b_1 c_1 & a_2 b_2 c_2 & a_3 b_3 c_3 \\ a_1 c_1 & a_2 c_2 & a_3 c_3 \\ b_1 c_1 & b_2 c_2 & b_3 c_3 \end{bmatrix} = \exp\{\boldsymbol{Q}\mathbf{B}\}, \quad (18)$$

where

$$\boldsymbol{Q} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix}, \boldsymbol{B} = \ln \begin{bmatrix} a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \\ c_1 & c_2 & c_3 \end{bmatrix}. \quad (19)$$

5.2.2 Computation of Buffer Size Requirement and Data Traffic. As discussed in previous works [32], the key to hardware performance evaluation is to calculate the buffer size requirements and the data traffics between neighboring memory levels, based on which the latency, power, and area can be easily computed.

One common property of previous analytical memory performance models [26, 32] is that the buffer requirements and the data traffics can be expressed as the product of a selected set of (tiled) loop boundaries, and a limited number of their transformations. Equations (1-6) are good examples. Via the *Ln-Exp* trick, they can be described as the inner product of a binary vector, called *query vector*, and a vector of the logarithm of loop boundaries and a few of their transformations called *ln-bound vector*. The query vector is mainly determined by the selection of loop order and buffering level, i.e., by $\boldsymbol{o}_D, \boldsymbol{o}_G, \boldsymbol{o}_R$; while the ln-bound vector is decided by the tiling and spatial unrolling, i.e., by $\boldsymbol{m}, \boldsymbol{c}, \boldsymbol{i}, \boldsymbol{j}, \boldsymbol{r}, \boldsymbol{q}$. In the example depicted in Listing 2, the buffer size requirement for kernel weights at GB can be derived from Equation (4) and computed as follows:

$$BS_W(L_{11}) = y_G * x_G * m_S * m_R * c_R * x_R * y_R = \exp\langle \boldsymbol{q}, \boldsymbol{b} \rangle, \quad (20)$$

where $\boldsymbol{q} = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 0, 1, 1]$,
$\boldsymbol{b} = \ln([m_D, c_D, i_D, j_R, x_D, y_D, m_G, c_G, i_G, j_G, x_G, y_G, m_S, i_s, m_R, c_R, i_R, j_R, x_R, y_R])$.

According to Equation (5), the traffic of weight data from DRAM to GB can be computed as follows:

$$TF_D^W(L_{11}) = m_D * c_D * i_D * j_D * x_D * y_D * m_G * c_G * BS_W(L_{11}) = \exp\langle \boldsymbol{q}', \boldsymbol{b} \rangle, \quad (21)$$

where $\boldsymbol{q}' = [1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 0, 1, 1, 0, 0, 1, 1]$. Note that a few transformations of the loop boundaries, e.g., $(i_R - 1) * S + y_R$, might be required for discussing the buffer requirement and data transfer traffic of IN array, as implied by Equation (1).

Different loop orders and buffering levels can lead to the product of different sets of loop boundaries. We can simply let the ln-bound vector involves the union of loop boundaries and transformations required by various loop orders and buffering levels. And the query vectors corresponding to different loop orders and buffering levels can be stacked into a matrix, denoted as Query Matrix $\mathbf{Q}$. For different memory levels of IN, OUT, and W arrays, we generate a set of individual Query Matrices to compute the buffer size and data transfer traffic. To further enable parallel evaluation of design solutions, different ln-bound vectors, corresponding to different tiling and parallelism strategies, are stacked into a matrix, denoted as $Ln - Bound$ Matrix $\boldsymbol{B}$. In this way, various loop order and buffering level solutions, as well as tiling and parallelism solutions, can be evaluated in parallel by a few matrix multiplications. The Buffer Size

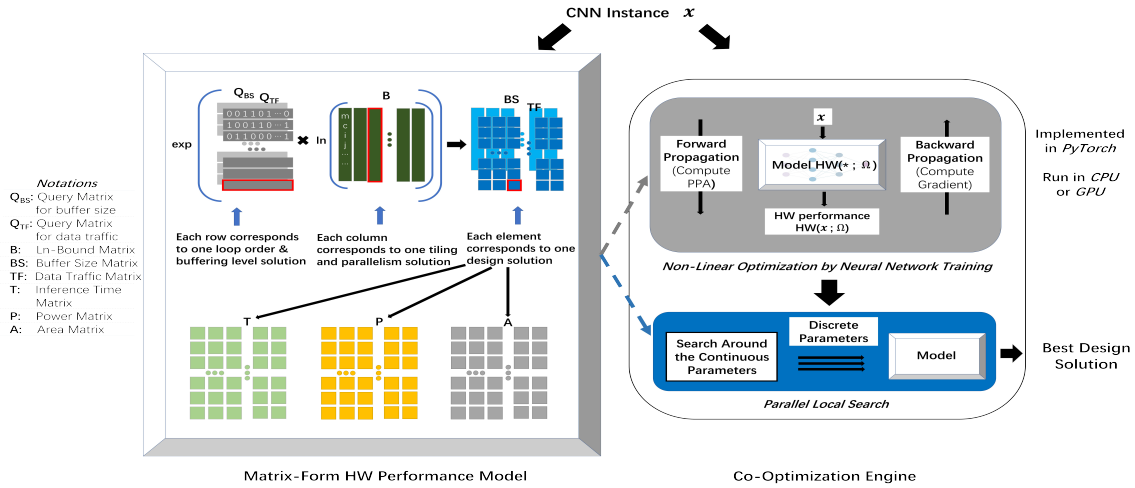

Figure 1: CNN hardware and dataflow co-optimization framework.

Matrices $\mathbf{BS}_X^Y$ ($X = IN, OUT, W$ and $Y =$ GB, RF) can be computed as follows:

$$\mathbf{BS}_X^Y = \mathbf{Q}_{BS}^{X,Y}\mathbf{B}, \tag{22}$$

where $\mathbf{Q}_{BS}^{X,Y}$ is the Query Matrix of buffer size for X array at Y. $\mathbf{BS}_X^Y(i,j)$ corresponds to the buffer size requirement of X array at memory level Y with the i-th combination of loop order and buffering level and the j-th tiling and spatial unrolling.

Similarly, the Traffic Matrices $\mathbf{TF}_X^Y$ ($X = IN, OUT, W$ and $Y =$ GR, RF) are computed as follows:

$$\mathbf{TF}_X^Y = \mathbf{Q}_{TF}^{X,Y}\mathbf{B}, \tag{23}$$

where $\mathbf{Q}_{TF}^{X,Y}$ is the Query Matrix of data traffic for X array at memory level Y. Here, data traffic at GB means the data transfer between DRAM and GB. And data traffic at RF means the the data transfer between GB and RF. With the Traffic Matrices, we can further obtain the Buffer Access Matrices $\boldsymbol{AC}_X^Y$ ($X = IN, OUT, W$ and $Y =$ DRAM, global buffer, register file) by conducting a few linear transformations on the Traffic Matrices. $\boldsymbol{AC}_X^Y$ represents the buffer access counts of X array at memory level Y.

However, the curse of dimensionality prevents naïve applications of the aforementioned matrix-form model to systems with multi-level memory hierarchy. Since the height of the Query Matrices $\mathbf{Q}_{BS/TF}^{X,Y}$ equals the number of combinations of loop order and buffering level equals which grows exponentially to the number of memory levels. We address this problem by breaking the coupling between different memory levels. For example, Equation (20) can be rewritten as

$$\begin{aligned} BS_W(L_{11}) &= y_G x_G * (m_S m_R c_R y_R x_R) \\ &= y_G x_G * \text{Downstream Footprint} \\ &= \exp\langle \boldsymbol{p}, \boldsymbol{b}' \rangle * \text{Downstream Footprint}, \end{aligned} \tag{24}$$

where $\boldsymbol{p} = [0,0,0,0,1,1]$ and $\boldsymbol{b}' = \ln[m_G, c_G, i_G, j_G, y_G, x_G]$. In this way, we only need to consider the loop order and buffering level options at the current memory level when computing the buffer requirement. The result is then multiplied by the downstream footprint. In this example, the downstream footprint means the footprint when buffering at loop L_8.

Similarly, the traffic computation Equation (21) can be rewritten as

$$\begin{aligned} TF_D^W(L_{11}) &= (m_D c_D i_D j_D y_D x_D) * m_G c_G y_G x_G * (m_S m_R c_R y_R x_R) \\ &= \text{Upstream Bounds} * m_G c_G y_G x_G * \text{Downstream Footprint} \\ &= \text{Upstream Bounds} * \exp\langle \boldsymbol{p}', \boldsymbol{b} \rangle * \text{Downstream Footprint}, \end{aligned} \tag{25}$$

where $\boldsymbol{p}' = [1,1,0,0,1,1]$. It means that the effects of higher memory levels on the traffic of lower levels can be described by the product of upstream loop boundaries, and the effects of lower memory levels on a higher level can be described by the downstream footprint.

It is also noteworthy that our method can incorporate various memory performance models, as long as the buffer requirements and the data transfer traffics can be expressed as the product of loop boundaries and their transformations. In this paper, we mainly deploy the memory performance model in [26]. It is an analytical model for loop-nest optimization targeting architectures with application managed buffer, which is commonly used in CNN accelerators. We derive the matrix-form and extend it to consider multi-level memory hierarchy by our aforementioned techniques.

5.2.3 One-Time Offline Computation. As discussed in Section 5.2.2, different loop orders and buffering levels can lead to the product of different sets of loop boundaries when computing the buffer size requirements and data traffic. Previous analytical models [26, 32] contain a large number of hidden "if-else" operations to find the right set of loop boundaries for a given loop order and buffering level, this greatly degrades the efficiency of gradient computation.

We propose a one-time offline computation technique that can completely avoid any "if-else" operations during the online evaluation of design solutions. To be specific, we generate the Query Matrices $\mathbf{Q}_{BS/TF}^{X,Y}$ offline and reuse them across different CNN layers. As introduced in Section 5.2.2, each row in the Query Matrix is a query vector that corresponds to one combination of loop order and buffering level. It is a binary vector, and the locations of its "1" elements indicate the set of loop boundaries that need to be multiplied when computing buffer size or data traffic. In this way,

the results of "if-else" operations are baked into the Query Matrices and reused across CNN layers.

Table 2: Per Access Power

RF size	Energy (pJ)	GB size	Energy (pJ)
32 B	0.06	32 KB	5.82
64 B	0.12	64 KB	8.1
128 B	0.24	128 KB	11.66
256 B	0.48	256 KB	15.6
512B	0.96	512 KB	23.27
1 KB	1.2	1 MB	36.32

5.2.4 Latency, Power and Area Computation. We define E^{DRAM} as a constant scalar representing the per DRAM access power. But the access power of GB and RF depends on their sizes, thus depending on the hardware resource allocation. Table 2 is obtained by interpolating the data reported in [31]. We can observe that the per access power grows with the buffer size. Hence, we define Per-Access-Power Matrices $\boldsymbol{E}_X^Y$ ($X = IN, OUT, W$ and Y = GB, RF) as follows:

$$\boldsymbol{E}_X^Y(i,j) = tab(\boldsymbol{BS}_X^Y(i,j)), \tag{26}$$

where $tab(x)$ means rounding up x to the closest buffer size in Table 2 and fetching the corresponding per access power value.

The Memory Access Power Matrices $\boldsymbol{Pm}_X^Y$ ($X = IN, OUT, W$ and Y = DRAM, GB, RF) can be calculated as:

$$\boldsymbol{Pm}_X^Y(i,j) = \begin{cases} \boldsymbol{E}_X^Y(i,j) * \boldsymbol{AC}_X^Y(i,j), & \text{if } Y = \text{GB, RF;} \\ \boldsymbol{E}^{DRAM} * \boldsymbol{AC}_X^Y(i,j), & \text{if } Y = \text{DRAM.} \end{cases} \tag{27}$$

The MAC computation power is

$$P_{MAC} = N_{MAC} * E_{MAC}, \tag{28}$$

where N_{MAC} is the count of MAC computations and E_{MAC} is the per MAC computation power.

The Total Power Matrix $\boldsymbol{P}$ can be obtained as:

$$\boldsymbol{P} = \sum_{\substack{X=IN,OUT,W \\ Y=\text{DRAM, GB, RF}}} \boldsymbol{Pm}_X^Y + P_{MAC}\boldsymbol{J}, \tag{29}$$

where $\boldsymbol{J}$ is an all-one matrix with the same shape as $\boldsymbol{Pm}_X^Y$.

The Inference Latency Matrix $\boldsymbol{T}$ can be calculated as:

$$\boldsymbol{T}(i,j) = \frac{N_{MAC}}{\boldsymbol{N}_{PE}(i,j)}, \tag{30}$$

where the $\boldsymbol{N}_{PE}(i,j)$ is the number of PEs, i.e., the product of boundaries of spatial unrolling loops, for the i-th combination of loop order and buffering level and the j-th tiling strategy. Such inference time can be achieved by providing a large enough data transfer bandwidth, and we use the bandwidth as a constraint in our experiments. The Bandwidth Requirement Matrices $\boldsymbol{BW}_X^Y$ can be computed as:

$$\boldsymbol{BW}_X^Y(i,j) = \boldsymbol{TF}_X^Y(i,j)/\boldsymbol{T}_X^Y(i,j). \tag{31}$$

The Area Matrix $\boldsymbol{A}$ can be calculated as:

$$\boldsymbol{A} = \sum_{\substack{X=IN,OUT,W \\ Y=\text{GB, RF}}} \boldsymbol{BS}_X^Y A_{SRAM} + A_{MAC}\boldsymbol{N}_{PE}, \tag{32}$$

where the A_{SRAM} is the area per buffer size unit and A_{MAC} is the area per MAC unit.

5.3 Non-Linear Optimization by Neural Network Training

It is difficult to directly solve the original problem in Equation (7), due to its enormous and discrete solution space. Hence, we relax the original problem into a non-linear optimization problem and then solve it with a gradient-based approach.

5.3.1 Problem Relaxation. Originally, the tiled loop boundaries $m_D, m_G, m_R, m_S, c_D, c_G, \ldots, x_D, x_G, x_R, x_S$ are required to be integers. Now we relax them to continuous. Also, $\boldsymbol{o}_D, \boldsymbol{o}_G$ and $\boldsymbol{o}_R$ originally are one-hot vectors, each of which indicating the selection of one combination of loop order and buffering level at the corresponding memory level. Now they are relaxed to represent the probabilities of sampling over the combinations of loop orders and buffering levels. Constraint (15) is relaxed as follows.

$$0 \le \pi_1^D, \pi_2^D, \ldots, \pi_1^G, \pi_2^G, \ldots, \pi_1^R, \pi_2^R, \ldots, \le 1, \tag{33}$$

$$\sum_{i=1}^{i=k_D} \pi_i^D = 1, \sum_{i=1}^{i=k_G} \pi_i^G = 1, \sum_{i=1}^{i=k_R} \pi_i^R = 1. \tag{34}$$

We denote the relaxed version of $\boldsymbol{o}_D, \boldsymbol{o}_G, \boldsymbol{o}_R$ as $\boldsymbol{\pi}_D, \boldsymbol{\pi}_G, \boldsymbol{\pi}_R$. One-hot vectors $\boldsymbol{z}_D, \boldsymbol{z}_G, \boldsymbol{z}_R$ can be generated according to $\boldsymbol{\pi}_D, \boldsymbol{\pi}_G$ and $\boldsymbol{\pi}_R$ as follows. For $X = D, G, R$,

$$p_{\boldsymbol{\pi}_X}(u_X = i) = \pi_i^X, i = 1, 2, \ldots, k_X, \tag{35}$$

$$\boldsymbol{z}_X = one\_hot(u_X), \tag{36}$$

where $p_{\boldsymbol{\pi}_X}$ denotes a probability distribution function with parameter $\boldsymbol{\pi}_X$.

.

The relaxed problem is defined as:

$$\min_{\Omega \triangleq \{\boldsymbol{\pi}_D, \boldsymbol{\pi}_G, \boldsymbol{\pi}_R, \boldsymbol{m}, \boldsymbol{c}, \boldsymbol{i}, \boldsymbol{j}, \boldsymbol{r}, \boldsymbol{q}\}} \mathbb{E}_{\substack{\boldsymbol{z}_D \sim p_{\boldsymbol{\pi}_D} \\ \boldsymbol{z}_G \sim p_{\boldsymbol{\pi}_G} \\ \boldsymbol{z}_R \sim p_{\boldsymbol{\pi}_R}}} [T(\Theta), P(\Theta), A(\Theta)], \tag{37}$$

$$\Theta = [\boldsymbol{z}_D, \boldsymbol{z}_G, \boldsymbol{z}_R, \boldsymbol{m}, \boldsymbol{c}, \boldsymbol{i}, \boldsymbol{j}, \boldsymbol{r}, \boldsymbol{q}]. \tag{38}$$

Here, $\mathbb{E}[x]$ means the expectation of x.

5.3.2 Neural Network Training. We notice the analogy between neural network training and our problem Equation (37) as follows.

(1) Both problems are non-linear optimization problems. In neural network training, neural network weights are updated to reduce the misprediction error; while our task is to optimize hardware performance by tuning design parameters.
(2) In the forward propagation of neural network training, a feature vector goes through a series of network layers, which essentially comprise of matrix operations, to predict a label. In our task, a CNN layer instance is fed to our matrix-form model to output hardware performance.
(3) In the backward propagation of neural network training, the gradient of the misprediction error w.r.t. network weights are computed and used to update weights. In our task, the gradient of hardware performance w.r.t. design parameters are computed to guide the update of design parameters.

We implement our hardware performance model with the deep learning toolkit *PyTorch*[23], which offers mature and efficient implementation of matrix operations, automatic gradient derivation, and optimization engines with compatibility to both CPU and GPU

acceleration. We set the learning rate to be 0.1043 and epoch count to be 500, and use the *Adam* optimizer for network training.

Our problem has multiple objectives, i.e., latency, power, and area. We deploy the multi-objective optimization technique proposed in [25], whose main idea is introduced in Section 3.4.

5.4 Parallel Local Search

The outputs from our non-linear optimization stage are continuous design parameters. Hence, we conduct a local search around the continuous parameters to find the best integer parameters, as shown in Figure 1. It is a discrete optimization step and also implemented with deep learning toolkits. It is noteworthy that our parallel local search does not necessarily require the usage of multiple CPU cores or GPU. Instead, thanks to our matrix-form model, millions of design solutions can be evaluated in parallel via a few matrix operations, which are efficiently accelerated by deep learning toolkits even on a single CPU core.

6 EVALUATION

Experiments are conducted on VGG16, ResNet18, and AlexNet. All algorithms are implemented in Python and run on one core in AMD Ryzen Threadriper 1920X 12-Core Processor, except for section 6.3 where an additional Nvidia GeForce RTX 2080 Ti GPU is used.

6.1 Model Accuracy

We evaluate the accuracy of our model with respect to Timeloop [20], which is a state-of-the-art DNN hardware performance model. We generate thousands of different valid dataflows for three different CNN layers. Each dataflow is fed to both Timeloop and our model to compute the hardware performance.

Table 3: Relative errors of our model vs. Timeloop

	RF Size	GB Size	Power	Latency
Average	0%	0%	1%	0%
RMSE	0%	0%	3%	0%

Table 3 shows the relative errors of our model compared to Timeloop. We can see a perfect match on register file level buffer size, global buffer size, and latency estimation. The relative errors for power estimation are 1% on average. In general, our model matches Timeloop very well. Since many solutions generated in the rest of the experiments cannot be handled by Timeloop, they are evaluated by our model.

6.2 Comparison with Previous Works

Our hardware and dataflow co-optimization method is compared with the following previous works:

(1) **GAMMA**[15]: It is a generic algorithm-based two-stage approach. Latency is optimized at stage I, while stage II focuses on power optimization. The original GAMMA optimizes only dataflow, and we extended it to optimize CNN hardware as well.
(2) **Gumbel Softmax based approach**[11]: Similar work is proposed for a differentiable co-optimization framework for network structure and accelerator co-design [6]. In our experiments, we implement a differentiable accelerator search engine for comparison using Gumbel Softmax.
(3) **Bayesian Optimization** introduced in [24].

6.2.1 Power-Latency Optimization Results. We first report optimization results with power and latency as the objective. At the same time, all the methods share the same maximum chip area limit $\leq$ 5e6 μm^2. Figure 2 shows layer-by-layer comparison for *VGG*16. Our approach dominates all the other methods in latency and power for all layers. The CNN dimensions for layers 11, 12, and 13 are the same, and so are their solutions for each method.

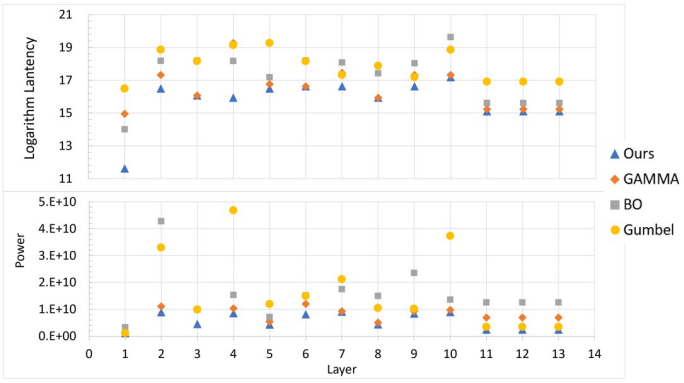

Figure 2: Per layer results for VGG16 when optimizing power and latency.

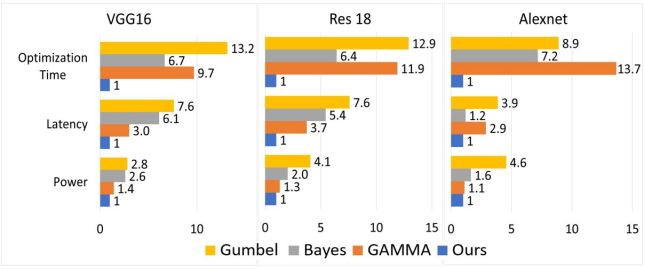

Figure 3: Overall results on power and latency optimization running on a single CPU core.

Figures 3 shows the overall results for *VGG*16, *Alexnet*, and *Res*18 on power, inference latency and optimization runtime. The overall result for a CNN network is obtained by summing up the results of all layers. Our approach dominates all the other methods in all three cases. Compared to Bayesian optimization, our method achieves **1.2-6× less inference latency** with 38% to 62% less power consumption. Compared to GAMMA and Gumbel Softmax approach, our approach reduces latency by about 2.9× or more and power reduction from 10% to 78%. Due to our matrix form performance model, the non-linear optimization of the proposed method can be performed through neural network training in *PyTorch*. Therefore, the proposed method provides **9-13×** and **8-13×speedup of optimization runtime** compared with GAMMA and Gumbel Softmax approach, respectively.

6.2.2 PPA Optimization. We also evaluate our method on PPA (Power Performance Area) optimization, where performance is indicated by inference latency. The comparison is made with GAMMA, which achieves the best power-latency results among the three previous works. Figure 4 shows the PPA comparison results. The

proposed method outperforms GAMMA in all cases. Compared with GAMMA, our method achieves up to **1.7 × shorter inference latency, 37.5% less power consumption**, **3× less area** and **6.2 less optimization time** on ResNet 18.

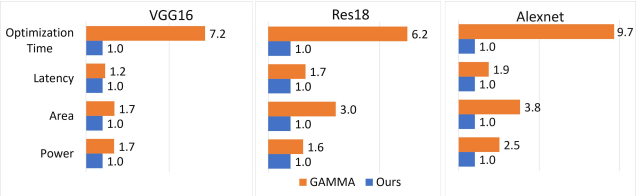

Figure 4: PPA results. CNN performance is indicated by inference latency.

6.3 Runtime Analysis

6.3.1 Why Does the Matrix Form Matter? One of our key contributions is the analytical model in a matrix form, which allows fast solution evaluation through a deep learning toolkit such as PyTorch. Experiments are conducted on VGG16 layer 1. During the local search, our matrix form model can evaluate 36 million solutions (which is still a very tiny portion of the design solution space) in 35 secs. The evaluation time per design solution of matrix form is **2300× faster** than non-matrix form analytical model. At the same time, it is also **8300× faster** than Timeloop solution evaluation. Therefore, the matrix form matters much more than merely an analytical form model.

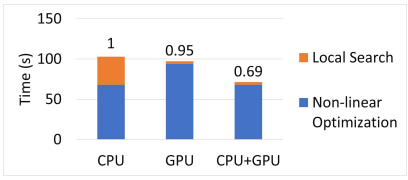

Figure 5: Runtime breakdown on different platforms.

6.3.2 Effect of Computing Platforms. Figure 5 shows the runtime breakdown for optimizing PPA of VGG16 layer 1 by our method on different platforms. On CPU, the non-linear optimization takes 68 secs, and local search takes 34 secs. Where on GPU, the non-linear optimization takes 94 secs and local search takes 3.3 secs. One can tell that the local search can be easily parallelized on GPU while the non-linear optimization part runs faster on the CPU due to the small size of our model.

6.4 Ablation Study

6.4.1 Effect of Local Search. The local search evaluates a large number of integer solutions in the neighborhood of the fractional non-linear optimization solution and finds the optimal integer one. An alternative and naïve method is to round the fractional solution to its integer neighbor with the minimum Euclidean distance, a.k.a. nearest rounding. Figure 6 compares the integer solutions from our local search and nearest rounding on 4 CNN layers of VGG16. One can see that our local search leads to solutions superior to the nearest rounding. Since the results are shown in a logarithmic scale to cover a wide range, the actual difference is greater than it appears to be. The runtime cost of our local search is labeled in the figure.

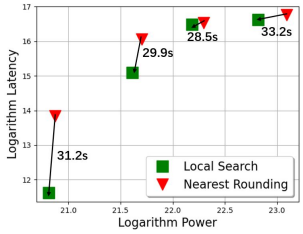

Figure 6: Comparison between local search and nearest rounding.

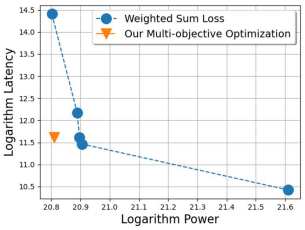

Figure 7: Power and latency tradeoff of multi-objective optimization.

6.4.2 Multi-objective Optimization. Figure 7 shows the effect of our multi-objective optimization in comparison with a common approach of minimizing a weighted sum between power and latency on VGG16 layer 1. By varying the weighting factors, the weighted sum method can obtain the blue dot solutions with different latency-power tradeoffs. Our multi-objective optimization can lead to the orange triangle solution that dominates all the Pareto front solutions found by the weighted sum method.

6.5 Parallelism Between CNN Layers

So far our method focuses on the parallelism inside a single CNN layer. We, therefore, devise a two-stage heuristic for applying our method to accelerators where the entire CNN model is executed in a layer-wise pipeline manner [15]. Stage 1: Run our algorithm to optimize PPA for each layer. Stage 2: Identify the latency bottleneck over layers and then re-optimizing power and area for each layer given the latency bottleneck as a constraint. Future extensions of our work will comprehend efficient handling of parallelism between CNN layers.

7 CONCLUSION

Hardware and dataflow co-optimization is critical for the performance, power, and area of CNN accelerator designs. However, the co-optimization solution space is enormously large and presents a huge challenge. We develop a matrix form analytical model for evaluating co-optimization solutions, and it is 2300× faster than a non-matrix form analytical model. Based on this model, we propose a co-optimization approach that significantly outperforms several state-of-the-art methods. Our approach can also be incorporated as an important component in NAS-accelerator co-design.

ACKNOWLEDGMENTS

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

REFERENCES

[1] M. S. Abdelfattah, Ł. Dudziak, T. Chau, R. Lee, H. Kim, and N. D. Lane. 2020. Best of Both Worlds: AutoML Codesign of a CNN and its Hardware Accelerator. In Design Automation Conference. 1–6.
[2] Y-H. Chen, T. Krishna, J. Emer, and V. Sze. 2017. Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks. IEEE Journal of Solid-State Circuits 52, 1 (2017), 127–138.
[3] K. Choi, D. Hong, H. Yoon, J. Yu, Y. Kim, and J. Lee. 2021. DANCE: Differentiable Accelerator/Network Co-Exploration. In Design Automation Conference. 337–342.
[4] J. Cong, P. Wei, C. H. Yu, and P. Zhang. 2018. Automated Accelerator Generation and Optimization with Composable, Parallel and Pipeline Architecture. In Design Automation Conference. 1–6.

[5] C. Deng, Y. Sui, S. Liao, X. Qian, and B. Yuan. 2021. GoSPA: An Energy-efficient High-performance Globally Optimized SParse Convolutional Neural Network Accelerator. In International Symposium on Computer Architecture. 1110–1123.
[6] Y. Fu, Y. A. Zhang, Y. Zhang, D. Cox, and Y. Lin. 2021. Auto-NBA: Efficient and Effective Search Over the Joint Space of Networks, Bitwidths, and Accelerators. arXiv:2106.06575
[7] K. Hegde, H. Asghari-Moghaddam, M. Pellauer, N. Crago, A. Jaleel, E. Solomonik, J. Emer, and C. Fletcher. 2019. ExTensor: An Accelerator for Sparse Tensor Algebra. In International Symposium on Microarchitecture. 319–333.
[8] K. Hegde, P. Tsai, S. Huang, V. Chandra, A. Parashar, and C. Fletcher. 2021. Mind Mappings: Enabling Efficient Algorithm-Accelerator Mapping Space Search. In International Conference on Architectural Support for Programming Languages and Operating Systems. 943–958.
[9] Q. Huang, M. Kang, G. Dinh, T. Norell, A. Kalaiah, J. Demmel, J. Wawrzynek, and Y. Shao. 2021. CoSA: Scheduling by Constrained Optimization for Spatial Accelerators. In International Symposium on Computer Architecture. 554–566.
[10] Y. Huang, Y. Cheng, A. Bapna, O. Firat, M. X. Chen, D. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu, and Z. Chen. 2019. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. In Conference on Neural Information Processing Systems. 103–112.
[11] E. Jiang, S. Gu, and B. Poole. 2017. Categorical Reparameterization with Gumbel-Softmax. arXiv:1611.01144
[12] W. Jiang, L. Yang, E. Sha, Q. Zhuge, S. Gu, S. Dasgupta, Y. Shi, and J. Hu. 2020. Hardware/Software Co-Exploration of Neural Architectures. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (2020), 1–6.
[13] Norman P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, et al. 2017. In-datacenter performance analysis of a tensor processing unit. In Annual International Symposium on Computer Architecture. 1–12.
[14] S. Kao, G. Jeong, and T. Krishna. 2020. ConfuciuX: Autonomous Hardware Resource Assignment for DNN Accelerators using Reinforcement Learning. In International Symposium on Microarchitecture. 622–636.
[15] S.-C. Kao and T. Krishna. 2020. GAMMA: Automating the HW Mapping of DNN Models on Accelerators via Genetic Algorithm. In International Conference On Computer Aided Design. 1–9.
[16] H. Kwon, P. Chatarasi, V. Sarkar, T. Krishna, M. Pellauer, and A. Parashar. 2020. MAESTRO: A Data-Centric Approach to Understand Reuse, Performance, and Hardware Cost of DNN Mappings. International Symposium on Microarchitecture 40, 3 (2020), 20–29.
[17] H. Kwon, A. Samajdar, and T. Krishna. 2018. Maeri: Enabling flexible dataflow mapping over dnn accelerators via reconfigurable interconnects. ACM SIGPLAN Notices 53, 2 (2018), 461–475.
[18] Y. Li, C. Hao, X. Zhang, X. Liu, Y. Chen, J. Xiong, W. Hwu, and D. Chen. 2020. EDD: Efficient Differentiable DNN Architecture and Implementation Co-search for Embedded AI Solutions. In Design Automation Conference. 1–6.
[19] Y. Lin, Z. Jiang, J. Gu, W. Li, S. Dhar, H. Ren, B. Khailany, and D. Pan. 2020. Dreamplace: Deep learning toolkit-enabled gpu acceleration for modern vlsi placement. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 40, 4 (2020), 748–761.
[20] A. Parashar, P. Raina, Y.-S. Shao, Y.-H. Chen, V. A. Ying, A. Mukkara, R. Venkatesan, B. Khailany, S. W. Keckler, and J. Emer. 2019. Timeloop: A Systematic Approach to DNN Accelerator Evaluation. In International Symposium on Performance Analysis of Systems and Software. 304–315.
[21] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. Keckler, and S. Dally. 2017. SCNN: An accelerator for compressed-sparse convolutional neural networks. In International Symposium on Computer Architecture. 27–40.
[22] M. Parsa, J. P. Mitchell, C. D. Schuman, R. M. Patton, T. E. Potok, and K. Roy. 2020. Bayesian Multi-objective Hyperparameter Optimization for Accurate, Fast, and Efficient Neural Network Accelerator Design. Frontiers in Neuroscience 14 (2020).
[23] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. Advances in neural information processing systems 32 (2019).
[24] B. Reagen, J. M. Hernandez-Lobato, R. Adolf, M. Gelbart, P. Whatmough, G.-Y. Wei, and D. Brooks. 2017. A Case for Efficient Accelerator Design Space Exploration via Bayesian Optimization. In International Symposium on Low Power Electronics and Design. 1–6.
[25] O. Sener and V. Koltun. 2018. Multi-task learning as multi-objective optimization. Advances in neural information processing systems (2018), 525–536.
[26] A. Stoutchinin, F. Conti, and L. Benini. 2019. Optimally Scheduling CNN Convolutions for Efficient Memory Access. arXiv:1902.01492
[27] S. Venkataramani, J. Choi, et al. 2019. DeepTools: Compiler and Execution Runtime Extensions for RaPiD AI Accelerator. International Symposium on Microarchitecture 39, 5 (2019), 102–111.
[28] L. Waeijen, S. Sioutas, M. Peemen, M. Lindwer, and H. Corporaal. 2021. ConvFusion: A Model for Layer Fusion in Convolutional Neural Networks. IEEE Access 9 (2021), 168245–168267.
[29] Y. N. Wu, P. A. Tsai, A. Parashar, V. Sze, and J. S. Emer. 2021. Sparseloop: An Analytical, Energy-Focused Design Space Exploration Methodology for Sparse Tensor Accelerators. In International Symposium on Performance Analysis of Systems and Software. 232–234.
[30] L. Yang, Z. Yan, M. Li, H. Kwon, L. Lai, T. Krishna, V. Chandra, W. Jiang, and Y. Shi. 2020. Co-Exploration of Neural Architectures and Heterogeneous ASIC Accelerator Designs Targeting Multiple Tasks. In Design Automation Conference. 1–6.
[31] X. Yang, M. Gao, Q. Liu, J. Setter, J. Pu, A. Nayak, S. Bell, K. Cao, H. Ha, P. Raina, et al. 2020. Interstellar: Using halide's scheduling language to analyze dnn accelerators. In International Conference on Architectural Support for Programming Languages and Operating Systems. 369–383.
[32] Y. Zhao, C. Li, Y. Wang, P. Xu, Y. Zhang, and Y. Lin. 2020. DNN-Chip Predictor: An Analytical Performance Predictor for DNN Accelerators with Various Dataflows and Hardware Architectures. In International Conference on Acoustics, Speech and Signal Processing.
[33] Y. Zhou, X. Dong, B. Akin, M. Tan, D. Peng, T. Meng, A. Yazdanbakhsh, D. Huang, and R. Narayanaswami. 2021. Rethinking Co-design of Neural Architectures and Hardware Accelerators. arXiv:2102.08619