Comprehensive Accelerator-Dataflow Co-design Optimization for Convolutional Neural Networks

Miheer Vaidya
University of Utah
Salt Lake City, Utah, USA
m.vaidya@utah.edu

Aravind Sukumaran-Rajam
Washington State University
Pullman, WA, USA
a.sukumaranrajam@wsu.edu

Atanas Rountev
Ohio State University
Columbus, Ohio, USA
rountev@cse.ohio-state.edu

P. Sadayappan
University of Utah
Salt Lake City, Utah, USA
saday@cs.utah.edu

***Abstract*—The design space of possible schedules for mapping a Convolutional Neural Network layer onto a spatial accelerator array, referred as the *dataflow*, is enormous. The co-design of key architectural parameters (such as number of processing elements, sizes of register files and scratchpad memories) along with the dataflow to optimize the implementation of one or more CNN stages makes the design space explosively larger. Several recent efforts have addressed the design-space exploration problem for CNN accelerators via heuristics or limited search strategies. In this paper we develop the first optimization approach that uses analytical modeling and the solution of constrained nonlinear optimization problems for comprehensive algorithm-architecture co-design optimization. Using the Timeloop accelerator modeling framework, we demonstrate that the new optimization methodology can enable significant improvements over prior accelerator designs for both energy minimization and performance maximization.**

I. Introduction

Convolutional Neural Networks (CNNs) are key components of deep neural network pipelines. Due to its importance and its very compute-intensive nature, several hardware accelerator designs have been proposed for CNNs [5], [6], [7], [9], [17], as well as frameworks/methodologies for optimizing CNN on accelerators [8], [11], [12], [13], [14], [18], [22]. The overall design space is prohibitively large, when we consider both the choice of architectural parameters (such as number of processing elements (PE), sizes of register files, and sizes of scratchpad memories) and the possible schedule of execution of the arithmetic operations and movement of data within the accelerator (referred as the *dataflow*). Thus, prior efforts have been forced to make ad-hoc design choices or use heuristics. In this work, we present the first comprehensive model-driven co-design optimization approach that evaluates the full design space of accelerator parameters and dataflows.

Several recent efforts (COSA [11], DMazeRuner [8], Interstellar [22], MAESTRO/GAMMA [13], [14], Timeloop [18]) have sought to develop infrastructure for searching the space of dataflows for a given set of accelerator parameters. A few efforts (e.g., Confuxius [12]) have addressed the co-design problem of optimizing across both the dataflow and architectural parameters, but use heuristics search. Our work takes a fundamentally different view of the problem and its solution. We consider the full space of possible designs and *automatically generate a collection of constrained nonlinear optimization problems* to capture the cost of each point in the space. The variables in our formulation of the optimization problem include both variables affecting the dataflow (modeled as tile sizes of multi-level tiled representation of a CNN loop nest) as well as architectural parameters such as the number of registers per PE, on-chip SRAM shared memory, and the number of PEs. The nonlinear objective function can model either energy or delay (or energy-delay product, although we do not). By solving these optimization problems using an existing nonlinear solver, we *systematically identify optimal designs in the entire design space*. This approach can also be applied to optimize the dataflow for a fixed architectural configuration, by simply replacing the corresponding architectural parameters with specific numerical constants instead of casting them as variables in the generated nonlinear optimization problem.

The key insight driving this work is that although the design space for $\{architecture\} \times \{dataflow\}$ is explosively large, a systematic exploration of the entire design space is feasible via automatic generation and solution of a collection of constrained nonlinear optimization problems. We develop a compile-time algorithm for generation of symbolic expressions that model the volume of data movement for each array at each level of the memory hierarchy. This is done via an inner-to-outer traversal of an arbitrary permutation of tiling loops in a multi-level tiled loop configuration to builds symbolic expressions for data volume. The modeling of array footprints as symbolic polynomial expressions of tile size parameters is achieved via an analytical model that exploits the specific form of array index expressions in the CNN computation. Using an off-the-shelf solver (CVXPY [2]) the *entire design space can be searched comprehensively* to find optimal design points, in contrast to existing heuristic-based design exploration approaches. Our results show that this new approach yields significant improvements to energy efficiency and performance.

Contributions: The paper makes the following contributions:

- It develops a novel solution to the dataflow optimization problem for CNN accelerators by formulating and solving a constrained nonlinear (DGP - Disciplined Geometric Programming) optimization problem that is solved by an

We thank the CGO reviewers for their valuable feedback, which helped to improve the paper substantially. This work was supported in part by the U.S. National Science Foundation through awards 1946752, 2018016, 2119677, and 2118737. The support and resources from the Center for High Performance Computing at the University of Utah are gratefully acknowledged.

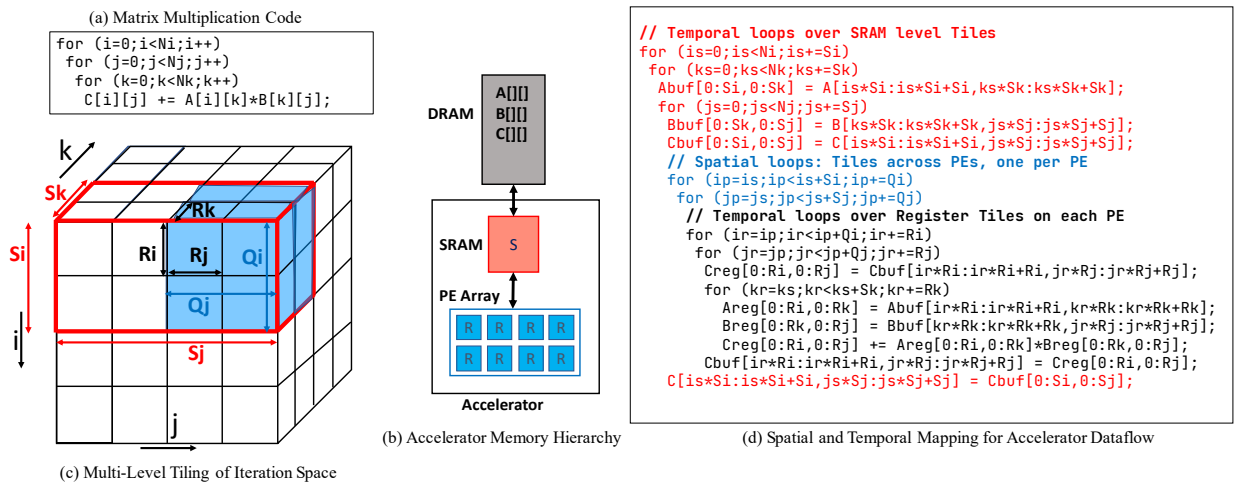

Fig. 1. Matrix multiplication: illustration of accelerator dataflow as multi-level tiling

off-the-shelf convex solver (CVXPY).

- It develops a novel solution to the architecture-dataflow co-design problem for CNN accelerators, optimizing across parameters for per-PE register capacity, SRAM capacity, and number of PEs, avoiding the need to solve the dataflow optimization problem for various combinations of choices for architectural parameters.
- It presents experimental results using a state-of-the-art accelerator modeling framework (Timeloop) showing that significant improvements to energy efficiency and performance over prior designs are achieved by use of the developed design-space exploration methodology.

II. Overview of Optimization Approach

In this section, we use the example of matrix-matrix multiplication to provide a high-level overview of the key ideas behind the model-driven approach to dataflow optimization and architecture-dataflow co-design for CNN accelerators. Technical details on the modeling are presented in Section III.

Illustrative example: matrix multiplication. We use the simpler matrix multiplication computation to explain the main ideas behind the new approach to comprehensive design space exploration and optimization. For the CNN computation, the analytical cost functions are more complex than for matrix multiplication, but have a similar structure.

Fig. 1(a) shows the loop code for standard matrix-matrix multiplication. To its left in Fig. 1(b) is shown an accelerator with a 3-level memory hierarchy: an array of processing elements (PEs) each with a set of R registers, a common on-chip shared memory with capacity of S words, and off-chip DRAM of sufficient capacity to hold all three operands.

The execution of the code in Fig. 1(a) on an accelerator will require slices of data of the three matrices to be moved between DRAM and the accelerator's SRAM and between SRAM to PE registers to perform the arithmetic MAC (Multiply ACcumulate) operations. The schedule of data movement between DRAM, buffers in SRAM, and registers is called the accelerator's *dataflow* and has a significant impact on the energy, delay and EDP (Energy-Delay Product). The enormous space of possible dataflows for a given nested loop computation on a given accelerator array can be systematically analyzed through the perspective of multi-level tiling. Fig. 1(c) depicts a tiled view of the 3D iteration space of the operations for the matrix-multiplication computation, corresponding to the tiled pseudocode shown in Fig. 1(d). The iteration space of N^3 points is partitioned at the outermost level into a temporal sequence of SRAM-level tiles. Each such tile is a contiguous $S_i \times S_j \times S_k$ 3D slice of the iteration space such that the required data slices for those iteration-space points can all fit within the capacity S of the SRAM.

Each SRAM-level tile is collectively executed by the set of P processing units, viewed as a logical 3D grid of size $P_i \times P_j \times P_k$. The tile is partitioned uniformly across the PEs and each PE covers a slice of the iteration space of size $Q_i \times Q_j \times Q_k$; here $Q_i = S_i/P_i$ and similarly for Q_j and Q_k. For simplicity, in the example we use $P_k = 1$ and thus $S_k = Q_k$. Each PE executes a sequence of register-level tiles of size $R_i \times R_j \times R_k$.

The pseudocode in Fig. 1(d) uses Python convention: indentation conveys the nesting structure of loops and a range $0{:}N$ denotes the set from 0 to N-1. At each level of the memory hierarchy, an explicit copy of the tile's data footprint must be made into an appropriately-sized buffer from a larger buffer at the next higher level in the hierarchy. The buffers in SRAM are named $Abuf$, $Bbuf$, and $Cbuf$ and the register-level buffers are $Areg$, $Breg$, and $Creg$.

Analytical modeling of data movement. Before execution of a tile, copy-in of the tile's data footprint for the input and in-out operands must be done from the next higher level of the memory hierarchy. The copy-in/copy-out operation can be hoisted out through loop iterators that are *absent* in an array's index expressions. For example, at the SRAM level tiles, the innermost tile-loop is along j, which is not used in the indexing expression for A. Hence, the copy into $Abuf$ from A can be moved above the js loop.

The total volume of data movement at each level in the memory hierarchy for each matrix can be computed by multiplying the volume of data associated with the copy-in/copy-out operations with the product of the number of surrounding loop iterations. For example, the volume of data movement from DRAM to SRAM for each copy operation (Abuf[0:Si,0:Sk]=A[is*Si:is*Si+Si,ks*Sk:ks*Sk+Sk]) is $S_i \times S_k$. The number of iterations of the surrounding ks and is loops are $\frac{N_k}{S_k}$ and $\frac{N_i}{S_i}$, giving a total data movement volume of $S_i \times S_k \times \frac{N_k}{S_k} \times \frac{N_i}{S_i} = N_i N_k$. The volume of data movement from DRAM to SRAM (denoted $D \rightarrow S$) and from SRAM to DRAM (denoted $S \rightarrow D$) for the three matrices is as follows:

$$\begin{aligned} DVol_A^{D\rightarrow S} &= S_i S_k \tfrac{N_k}{S_k} \tfrac{N_i}{S_i} = N_i N_k \\ DVol_B^{D\rightarrow S} &= S_k S_j \tfrac{N_j}{S_j} \tfrac{N_k}{S_k} \tfrac{N_i}{S_i} = \tfrac{N_i N_j N_k}{S_i} \\ DVol_C^{D\rightarrow S} &= DVol_C^{S\rightarrow D} = S_i S_j \tfrac{N_j}{S_j} \tfrac{N_k}{S_k} \tfrac{N_i}{S_i} = \tfrac{N_i N_j N_k}{S_k} \end{aligned} \tag{1}$$

The volume of data movement between SRAM and registers can be computed similarly. At this level, row-wise and column-wise multicasting of data is feasible, and the volume is counted as one word if the same word is loaded into registers by a set of processors. For example, the same set of $R_i R_k$ elements is loaded into registers by P_j processors and is counted as a multicast operation requiring a single read access of the on-chip SRAM. The data movement volumes from SRAM to registers ($S \rightarrow R$) and from registers to SRAM ($R \rightarrow S$) are:

$$\begin{aligned} DVol_A^{S\rightarrow R} &= R_i R_k \tfrac{S_k}{R_k} \tfrac{Q_j}{R_j} \tfrac{Q_i}{R_i} \tfrac{S_i}{Q_i} \tfrac{N_j}{S_j} \tfrac{N_k}{S_k} \tfrac{N_i}{S_i} = \tfrac{N_i N_j N_k}{R_j} \tfrac{Q_j}{S_j} \\ &= \tfrac{N_i N_j N_k}{R_j P_j} \\ DVol_B^{S\rightarrow R} &= R_k R_j \tfrac{S_k}{R_k} \tfrac{Q_j}{R_j} \tfrac{Q_i}{R_i} \tfrac{S_j}{Q_j} \tfrac{N_j}{S_j} \tfrac{N_k}{S_k} \tfrac{N_i}{S_i} = \tfrac{N_i N_j N_k}{R_i P_i} \\ DVol_C^{S\rightarrow R} &= DVol_C^{R\rightarrow S} = R_i R_j \tfrac{Q_j}{R_j} \tfrac{Q_i}{R_i} \tfrac{S_j}{Q_j} \tfrac{S_i}{Q_i} \tfrac{N_j}{S_j} \tfrac{N_k}{S_k} \tfrac{N_i}{S_i} \\ &= \tfrac{N_i N_j N_k}{S_k} \end{aligned} \tag{2}$$

Dataflow optimization. Given problem size parameters (N_i, N_j, N_k) and architectural parameters (number of processors P, SRAM size S, number of registers R per processor, energy per register access ϵ_R, SRAM access energy ϵ_S, DRAM access energy ϵ_D, energy per MAC operation ϵ_{op}), the dataflow optimization problem for energy is to find the tile-loop permutation and tile sizes (R_i, R_j, R_k, Q_i, Q_j, Q_k, S_i, S_j, S_k) that minimize the total energy for execution. The number of processors used is $P_i P_j P_k$, where $P_i = S_i/Q_i$, $P_j = S_j/Q_j, P_k = S_k/Q_k$.

Dataflow optimization can be formulated as follows:

$$\begin{aligned} &\text{minimize } \epsilon_{total} \text{ where} \\ &\epsilon_{total} = (4\epsilon_R + \epsilon_{op})N_{ops} + \\ &\epsilon_R(DVol^{S\rightarrow R} + DVol^{R\rightarrow S}) + \\ &\epsilon_S(DVol^{S\rightarrow R} + DVol^{R\rightarrow S} + DVol^{S\rightarrow D} + DVol^{D\rightarrow S}) + \\ &\epsilon_D(DVol^{S\rightarrow D} + DVol^{D\rightarrow S}) \\ &\text{subject to} \\ &R_i R_j + R_i R_k + R_j R_k \leq R \\ &S_i S_j + S_i S_k + S_j S_k \leq S \\ &S_i = P_i Q_i, S_j = P_j Q_j, S_k = P_k Q_k \\ &P_i P_j P_k \leq P \end{aligned} \tag{3}$$

For clarity, P_i, P_j, and P_k are included as unknown variables, although they can be eliminated via $P_i = S_i/Q_i$, etc. We use $DVol^{S\rightarrow R}$ to denote $DVol_A^{S\rightarrow R} + DVol_B^{S\rightarrow R} + DVol_C^{S\rightarrow R}$ and similarly for the remaining volume expressions.

The total energy is the sum of four components. The first component represents the energy corresponding to the execution of the MAC operations, and includes the energy for the MAC unit and the register energy for reading of the three inputs and writing back the result. The next three components account for the energy associated with all data movements from/to registers, from/to SRAM, and from/to DRAM, respectively: total number of accesses is multiplied by the corresponding energy per access. The minimization of total energy is subject to capacity constraints. The sum of sizes of the data buffers at the SRAM ($S_i S_k$ for A, $S_j S_k$ for B and $S_i S_j$ for C) must be less than the total SRAM capacity S, and a similar capacity constraint must be satisfied for registers. The total number of PEs used, $P_i P_j P_k$ cannot exceed the given number of PEs (P) on the accelerator. The product $P_i P_j P_k$ is not required to be equal to P, to allow for dataflows that do not necessarily use every PE. The total energy should include network energy for inter-PE data movement. In this work, we did not model network energy because our experimental evaluation as well as others [22] show this component to be non-dominant. However, the modeling of energy for inter-processor data movement could be included in a similar manner.

Thus a constrained nonlinear optimization problem is formulated in terms of unknown symbolic values for tile sizes and known constants for problem size parameters and architectural parameters. The above formulation corresponded to a specific permutation of tiling loops: ikj (outer-to-inner) for the SRAM-level tile-loops and ijk for the register-level tile-loops. A number of such constrained optimization problems can be similarly formulated for other permutations of the tile-loops (with significant pruning of the space of possible permutations, as explained in the next section). These constrained nonlinear optimization problem can be solved *directly* using off-the-shelf solvers; in our work, we use CVXPY [2]. *The formulation and direct solution of constrained nonlinear optimization problems for optimizing accelerator dataflow is a fundamental difference between the work presented here and all prior work on this problem that we are aware of.*

Architecture-dataflow co-design optimization. Next, consider architecture-dataflow co-design for the matrix-multiplication example. The three main architectural parameters are the number of processors (P), capacity of shared on-chip SRAM memory (S), and the number of registers per PE (R). The question of interest is: Given a specific set of problem size parameters (N_i, N_j, N_k), how should we choose architectural parameters and the corresponding dataflow to minimize energy? Such a question is of practical interest with CNNs because each CNN stage in a DNN pipeline (e.g., ResNet50) has specific fixed sizes for all tensors.

Several previous research efforts on accelerator co-design optimization [8], [12], [22] have been based on either heuristic

searches or bounded "grid" search, where specific combinations of architectural parameters are considered, and dataflow optimization is performed for each of the selected choices for architectural parameters. *In contrast, we demonstrate how architecture optimization for register/SRAM capacity and number of PEs, together with dataflow optimization, can be solved directly, without requiring an iterative search over a number of specific architectural configurations.*

The key to our *single-shot* approach to optimizing architectural parameters is that the per-access energy for registers and SRAM memory can be well approximated by analytical functions as follows:

$$\begin{aligned} \epsilon_R &= \sigma_R R \\ \epsilon_S &= \sigma_S \sqrt{S} \end{aligned} \tag{4}$$

The per-access-energy for register files (for a fixed word size) is linearly proportional to the number of registers, because the number of comparators grows linearly with the number of registers. The linear dependence is confirmed by the energy model used in Accelergy [21] and Alladin [20]. Similarly, the per-access energy for SRAM is quite well approximated by a square-root dependence on the capacity. This is because of the 2D structure of the storage bits in an SRAM, and the energy cost for the row/column decoders grows in proportion to the square root of the memory capacity. Assessment with Cacti [16] shows that this is a sufficiently accurate approximation for our approach to accelerator co-design.

Given the models in Eq. 4 for ϵ_R and ϵ_S, adding them to the system in Eq. 3 enables the direct solution to the architecture-dataflow co-design optimization problem of simultaneously optimizing for architectural parameters and tile parameters.

A fundamental constraint in choosing an accelerator's parameters is the total available chip area for the VLSI chip realizing the accelerator design. While the actual silicon area of a design cannot be determined until low-level VLSI design to layout the chip is done, we can perform high-level accelerator design-space exploration based on a simple linear model of area for each hardware component: the per-register area $Area_R$, the per-word SRAM area $Area_S$, and the area for a multiply-accumulate unit $Area_{MAC}$:

$$(Area_R \times R + Area_{MAC}) \times P + Area_S \times S \leq Area_{total} \tag{5}$$

We thus have the following nonlinear constrained optimization problem for finding the best combination of tile sizes and number of processors P for minimization of energy, for the specific tile loop permutation shown in Fig. 1:

$$\begin{aligned} &\text{minimize } \epsilon_{total} \text{ where} \\ &\epsilon_{total} = (4\sigma_R R + \epsilon_{op})N_{ops} + \\ &\sigma_R R(DVol^{S \to R} + DVol^{R \to S}) + \\ &\sigma_S \sqrt{S}(DVol^{S \leftrightarrow R} + DVol^{S \leftrightarrow D}) + \\ &\epsilon_D(DVol^{S \to D} + DVol^{D \to S}) \\ &\text{subject to} \\ &(Area_R \times R + Area_{MAC}) \times P + Area_S \times S \leq Area_{total} \\ &R_i R_j + R_i R_k + R_j R_k \leq R \\ &S_i S_j + S_i S_k + S_j S_k \leq S \\ &S_i = P_i Q_i, S_j = P_j Q_j, S_k = P_k Q_k \\ &P_i P_j P_k \leq P \end{aligned} \tag{5}$$

In this section, we used manual derivation of the analytical cost expressions for matrix multiplication to explain our new approach to dataflow optimization and architecture-dataflow co-design for accelerators. The expressions for CNN can be similarly derived but are more complex and the number of tile-loop permutations is much larger. We next present algorithmic details on how we automatically generate the multiple constrained nonlinear systems of equations for CNNs.

III. Analytical Modeling for CNN Optimization

In this section we elaborate on the details of the modeling for optimizing CNN accelerators. Consider the CNN code in Listing 1. Iterators n, k, c, r, s, h, w define a 7D iteration space. The code uses x and y to denote strides, which are small compile-time constants (convolution dilation can be handled similarly, but for simplicity is not discussed). The modeling of a multi-tiled version of this code, similar in structure to the example from Fig. 1, is described below. Such modeling is done at two levels. In the outer level of design exploration, different permutations of the tiling loops need to be considered, as each permutation can result in a different optimal dataflow and thus different energy and delay. In the inner level of exploration, given a particular permutation of tiling loops, a constrained optimization problem is generated and solved to obtain concrete values for tile sizes and architectural parameters. We first describe how to automatically generate such an optimization problem for a given permutation. Later in this section we describe how the large space of possible permutations is pruned.

Before presenting details on the algorithm for generating symbolic expressions for data volumes, we discuss a small change in the notation used for the symbolic expressions. Consider again the code in Fig. 1(c). Each level's tile loop steps through the tile origins by using a step size corresponding to that level's tile size (e.g., `is+=Si`). Alternatively, the code could be written with all tile-loops using unit step sizes (e.g., `is++`), with each tile-loop enumerating the tile number at that level. The loop bounds are modified accordingly (e.g., by using `Ni/Si` instead of `Ni`). With this form of tiled code, the tile size corresponding to any level of tiling is the product of the trip-counts (i.e., number of iterations) of all tile loops corresponding to that dimension nested within the current tile-loop. For example, tile-loop is in Fig. 1(c) would have loops ip and ir nested below it, and the tile size for is would be the product of their trip-counts. Note that the trip-count for the innermost loop is equal to its tile size. In the remainder of the paper, we use this alternative form, with the trip-count at a level being denoted using lower case letters and tile-sizes using upper case letters—for example, for the is loop in Fig. 1(c), the tile size would be S_i and the trip-count would be s_i. The

```
for(n = 0; n < N; n++)
 for(k = 0; k < K; k++)
  for(c = 0; c < C; c++)
   for(r = 0; r < R; r++)
    for(s = 0; s < S; s++)
     for(h = 0; h < H; h++)
      for(w = 0; w < W; w++)
       Out[n][k][h][w] +=
        In[n][c][x*h+r][y*w+s]*Ker[k][c][r][s]
```

Listing 1. CNN loops

constrained optimization problem will use the trip-counts as variables, as opposed to the tile sizes.

A. Generating a Constrained Optimization Problem

Suppose we are given a sequence TIP of tile-iterator permutations. Each element of TIP corresponds to a particular level of tiling. For example, for the code shown in Fig. 1, we have $TIP = \langle is, ks, js\rangle, \langle ip, jp, kp\rangle, \langle ir, jr, kr\rangle$. The first element of TIP represents the outer-to-inner order of iterators over SRAM tiles. The second element, which represents spatial loops over PEs, is included only for completeness, since the order of its iterators does not affect the modeled cost. The last element represents the order of register tiles on a PE. For the CNN computation, each element of TIP is a permutation of 7 iterators. Note that this representation, as well as the algorithm described below, allow an arbitrary number of tiling levels and arbitrary permutations at each level.

In order to auto-generate a constrained optimization problem, the approach incrementally builds two kinds of symbolic expressions. First, *data footprint* expressions, denoted by DF, are needed to generate the constraints of the optimization problem. For example, it is necessary to create a $\leq$ constraint that relates the seven SRAM-level tile sizes $S_{ns}, S_{ks}, S_{cs}, \ldots$ with the SRAM capacity S. To achieve this, the data footprints of SRAM data tiles for In, Ker, and Out need to be determined as symbolic expressions of the corresponding trip counts. These expressions also give the sizes of the corresponding buffers (similar to buffers $Abuf$, $Bbuf$, and $Cbuf$ in Figure 1).

In addition, in order to construct the objective function for the optimization problem, *data access volume* expressions are needed. These expressions, denoted by DV below, are the building blocks of the symbolic expression defining this objective function. This symbolic expression has a structure similar to the energy cost expression in Eq 3. For the rest of the discussion, we only consider the energy cost; cost expressions for delay can be constructed similarly.

Data footprint at the register level. The starting point of the approach is the computation of DF expressions for each array at the register level. For example, for array access $Out[n][k][h][w]$ and register loops with trip counts r_n, etc., the data footprint $DF^R_{Out} = r_n r_k r_h r_w$. Similarly, for array access $Ker[k][c][r][s]$ we have $DF^R_{Ker} = r_k r_c r_r r_s$.

Consider array access $In[n][c][h + r][2 * w + s]$. For concreteness, here we use stride-1 for the third data dimension and stride-2 for the last one. In the third dimension, the initial value of the index expression will be 0 and the final value will be $(r_h - 1) + (r_r - 1)$, leading to footprint extent of $r_h + r_r - 1$. For the last dimension, the accesses are from 0 to $2(r_w - 1) + (r_s - 1)$ and we have extent of $2r_w + r_s - 2$. The total footprint expression for this case is $DF^R_{In} = r_n r_c (r_h + r_r - 1)(2r_w + r_s - 2)$. The handling of the general case is done similarly.

Algorithm 1: Data volume/footprint symbolic expressions for one tensor and one tiling level

```
1  Function ConstructExpr(⟨it_1, ..., it_n⟩, DF^{l-1}):
2      DF^l ← DF^{l-1}
3      DV^l ← DF^{l-1}
4      CanHoist ← true
5      for k ← n to 1 do
6          c^l ← trip-count for it_k
7          c^{l-1} ← corresponding trip-count at lower level
8          if CanHoist then
9              if it_k is present in tensor reference then
                   // innermost present iterator
10                 CanHoist ← false
11                 DF^l ← replace(DF^l, c^{l-1}, c^l c^{l-1})
12                 DV^l ← replace(DV^l, c^{l-1}, c^l c^{l-1})
13             else
                   // before innermost present iterator
                   // no change to DF^l and DV^l
14             end
15         else
16             if it_k is present in tensor reference then
                   // non-innermost present iterator
17                 DF^l ← replace(DF^l, c^{l-1}, c^l c^{l-1})
18                 DV^l ← multiply(DV^l, c^l)
19             else
                   // no change to DF^l
20                 DV^l ← multiply(DV^l, c^l)
21             end
22         end
23     end
24     return DF_l, DV_l
```

Data footprint and data volume at higher tiling levels. Next, we outline the approach to construct symbolic expressions DF^l and DV^l at a tiling level l, given the footprint expression DF^{l-1} at next lowest tiling level. The register level footprint described earlier is DF^0. The computation is done separately for each tensor. Algorithm 1 defines the approach and the description below illustrates it with an example.

Data footprint. The algorithm considers the iteration permutation $\langle it_1, \ldots, it_n\rangle$ for tiling level l and models the effects of each loop, starting from the innermost one it_n and ending with the outermost one it_1. The construction of the data footprint symbolic expression DF^l determines the size of the buffer needed at this tiling level, which is necessary both for code generation and for constructing the capacity constraints in the optimization problem. This construction starts with the footprint DF^{l-1} at the lower level and then rewrites this expression as it traverses the loops. If a loop has an iterator that

TABLE I
EXAMPLE OF ACCESS VOLUME DV^1 COMPUTATION FOR $\langle w, n, k, h, c, s, r\rangle$ PERMUTATION USING ALGORITHM 1

Step	Iter	*In*	*Out*
DF^0		$r_n r_c(r_h + r_r - 1)(2r_w + r_s - 2)$	$r_n r_k r_h r_w$
1	r	$r_n r_c(r_h + q_r r_r - 1)(2r_w + r_s - 2)$	$2r_n r_k r_h r_w$
2	s	$q_s(r_n r_c(r_h + q_r r_r - 1)(2r_w + r_s - 2))$	$2r_n r_k r_h r_w$
3	c	$q_c q_s(r_n r_c(r_h + q_r r_r - 1)(2r_w + r_s - 2))$	$2r_n r_k r_h r_w$
4	h	$q_h q_c q_s(r_n r_c(r_h + q_r r_r - 1)(2r_w + r_s - 2))$	$2r_n r_k q_h r_h r_w$
5	k	$q_k q_h q_c q_s(r_n r_c(r_h + q_r r_r - 1)(2r_w + r_s - 2))$	$2q_k(r_n r_k q_h r_h r_w)$
6	n	$q_n q_k q_h q_c q_s(r_n r_c(r_h + q_r r_r - 1)(2r_w + r_s - 2))$	$2q_n q_k(r_n r_k q_h r_h r_w)$
7	w	$q_w q_n q_k q_h q_c q_s(r_n r_c(r_h + q_r r_r - 1)(2r_w + r_s - 2))$	$2q_w q_n q_k(r_n r_k q_h r_h r_w)$

is present in the tensor reference, the expression is rewritten to replace all occurrences of the previous level trip-count c^{l-1} with expression $c^l c^{l-1}$ where c^l is the corresponding trip-count at the current level (lines 11 and 17 in the algorithm).

For a simple example, consider DF^1 for tensor Ker. As discussed earlier, $DF^0 = DF^R_{Ker} = r_k r_c r_r r_s$. Suppose the loop order at level $l = 1$ is $\langle w, n, k, h, c, s, r\rangle$. The algorithm first considers r and since this iterator is present in reference $Ker[k][c][r][s]$, the occurrences of r_r are replaced with $q_r r_r$. Next, occurrences of r_s are replaced with $q_s r_s$, and so on. The final expression is $DF^1 = q_k r_k q_c r_c q_r r_r q_s r_s$.

As another example, consider DF^1 for tensor In, where the access expression is $In[n][c][h + r][2 * w + s]$. We have $DF^0 = DF^R_{In} = r_n r_c(r_h + r_r - 1)(2r_w + r_s - 2)$. When the tile loop for r is processed by the algorithm, the expression becomes $r_n r_c(r_h + q_r r_r - 1)(2r_w + r_s - 2)$. Next, the processing of s results in $r_n r_c(r_h + q_r r_r - 1)(2r_w + q_s r_s - 1)$. The final expression is $q_n r_n q_c r_c(q_h r_h + q_r r_r - 1)(2q_w r_w + q_s r_s - 1)$.

Data volume. To compute the data volume DV^l, we need to track the outermost loop level at which the data copy for a tensor can be hoisted. From inner to outer loop, this is the first loop for which the iterator is present in the tensor reference. In the algorithm this is done with flag $CanHoist$, which becomes $false$ when further hoisting is not possible. At this point, the occurrences of c^{l-1} are replaced with expression $c^l c^{l-1}$. From this point on, for all surrounding loops—regardless of whether the iterator is present or absent—the trip-counts need to be multiplied with the expression.

For example, for Ker and loop order $\langle w, n, k, h, c, s, r\rangle$, the flag becomes true as soon as we encounter the innermost loop r. The volume expression becomes $r_k r_c q_r r_r r_s$. After all surrounding loops are processed, the final expression is $DV^1 = q_w q_n q_k q_h q_c q_s(r_k r_c q_r r_r r_s)$. But for tensor reference $Out[n][k][h][w]$ the copy operation can be hoisted up to the h loop, since the innermost three loops do not contain iterators that appear in the reference. The resulting data volume expression is $DV^1 = 2q_w q_n q_k(r_n r_k q_h r_h r_w)$; factor 2 accounts for the fact that there are both read and write operations for this tensor. Finally, for tensor access expression $In[n][c][h+r][2*w+s]$, we have volume expression $r_n r_c(r_h + q_r r_r - 1)(2r_w + r_s - 2)$ when the innermost loop r is processed, and final expression $DV^1 = q_w q_n q_k q_h q_c q_s(r_n r_c(r_h + q_r r_r - 1)(2r_w + r_s - 2))$ at the end of the algorithm.

Table I provides a complete example of the steps (i.e., iterations of the loop at line 5) for computing DV^1 in the example discussed above, for tensors In and Out.

Pruning the design space. We use the following pruning techniques to significantly reduce the number of permutations to be considered in solving the tile-optimization problem. First, stencil-size parameters R and S take small odd numbers in practice and it is infeasible to divide them into a number of equal tiles. Hence tiling of these two loops is not considered. Next, as explained earlier, the construction of the symbolic expressions for data volume involves an inner to outer traversal of tiling loops, with a trip-count getting added to the expression after $CanHoist$ becomes $false$. Once we reach a tile-loop at a level such that $CanHoist$ is $false$ for all tensors, the relative order of outer surrounding tile loops for that memory level will not change the resulting expressions, i.e., we can prune away all but one of the possible permuted configurations of these loops. This allows a significant number of cases to be pruned out. Finally, we further reduce the number of tile-loop combinations by checking whether the cost functions for data volume are symmetric with respect to parameters H and W (i.e., they have the same strides) and pruning accordingly.

IV. SYSTEM DESCRIPTION

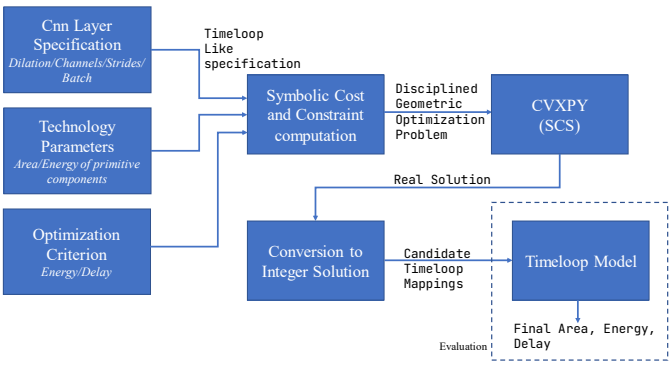

Fig. 2. Design space exploration with Thistle

Fig. 2 provides a high-level view of our optimizer, referred to as *Thistle*. The input includes: (1) CNN layer specification, (2) technology parameters, and (3) optimization criterion. The CNN layer specification defines problem parameters, e.g. batch size,

(a) Timeloop architecture specification

(b) Timeloop problem specification

(c) Equivalent problem in C notation

(d) Timeloop mapping

(e) Equivalent tiled and permuted version in C notation

Fig. 3. Timeloop input specification

output feature-map dimensions, number of input and output channels, strides, and dilation. The specification closely relates to the input YAML format used by Timeloop (discussed later in this section). The technology parameters include various area and energy parameters for components such as MAC units, registers, and SRAM/DRAM. In our evaluation we obtain technology parameters from Accelergy [21], Cacti [16], and Alladin [20]. The optimization criterion can be either energy or delay. Depending on the criterion used, Thistle formulates and solves appropriate constrained nonlinear optimization problems.

As described earlier, the symbolic cost and constraint computation takes in the aforementioned specifications and produces Disciplined Geometric Programs [1] that encode constraints and the objective function to be optimized. CVXPY [2] is used to solve the constrained optimization problems.

We convert the real solution provided by the solver to an integer solution which satisfies constraints of various memory capacities (e.g. sizes of SRAM usually being powers of two) and tile sizes as perfect divisors of problem sizes. For the memory capacity variables, we choose N closest powers of two near the real solution for those variables. For instance, if the real solution is 12 for register capacity and N is 2, we choose $8, 16$ as two candidates for register-capacity. We compute all the divisors of each problem extent. We start from the SRAM level tile-size variables. and select the n closest divisors to the corresponding real solutions for each of the variables. At this point, instead of considering all the divisors of problem extents, we only consider divisors of SRAM level candidates for a given variable. We repeat the process to subsequently choose n candidates for the processor level tile-sizes and the register level tile-sizes.

We use the cross product of these candidates and filter out any candidate solutions that: (1) violate divisibility constraints, (2)) violate area constraints, or (3) do not meet a minimum threshold on resource utilization. Typically we chose n to be 2 or 3 to avoid explosion of valid candidate solutions. The candidate solutions provided after conversion to integer solution are evaluated using Timeloop-model and the best candidate is chosen.

Timeloop [18] includes architectural estimation tools which accept a problem description, an architectural description and either a dataflow specification (mapping) or a dataflow constraints specification. The Timeloop mapper offers various search strategies to explore the search space of possible mappings. In our evaluation we use the default search strategy.

A Timeloop problem description includes: (1) a set of problem dimensions, i.e., a set of variables denoting a dense iteration space, (2) a set of data spaces, i.e., the arrays involved in the problem and the array access expressions, and (3) a problem instance with fixed problem parameter values.

Fig. 3(b) shows the Timeloop problem description for matrix multiplication $C[i][j] = A[i][j] \times B[k][j]$. The equivalent loop code is denoted in Fig. 3(c). The *dimensions* part denotes the 3D iteration space consisting of *I*, *J*, and *K* dimensions. The *data spaces* part describes the arrays involved in the computation and the iterators used to access them. The *read-write* attribute

specifies that buffers corresponding to the array C are both read and written in this process. The *instance* part specifies the maximum values for iterators *I*, *J*, and *K*.

Fig. 3(d) describes the tiling and loop permutation for the problem in Fig. 3(b), while the equivalent loop code is shown in Fig. 3(e). Here *target* specifies which memory level in the memory hierarchy will be accessed by a given set of loops. The register level loop permutation (target: RegisterFile) is *KJI*. Timeloop uses an innermost-to-outermost convention, so the iterator permutation would be $\langle i, j, k \rangle$. The *temporal* type signifies that these are sequential loops. The factors denote the trip counts of the loops for the given level. In case of RegisterFile, the trip counts for *I*, *J*, *K* are 4. The parallel loops, of type *spatial*, and the corresponding tile sizes are associated with higher memory levels by convention. The *spatial* block of mapping targeting SRAM specifies that the PE array is located below the SRAM (and above register file). The upper blocks similarly denote sequential loops executed by the PE array at the SRAM and DRAM level. The product of all factors for a given dimensions equals the total problem extent of the dimension, 64 in this example.

Fig. 3(a) provides a sample architecture specification. This architecture is similar to the three level memory hierarchy shown in Fig. 1(b). The nested tree denoted using YAML describes the memory hierarchy from the DRAM level down to the ALUs. Each memory level can be associated with read and write bandwidths supported by that level. Here *word-bits* denotes the number of bits in the primitive word; the example uses 16 bit words. The *depth* attribute denotes the depth of a memory level and corresponds to the size of that level. Each memory level has a *class* which is used by Accelergy to perform area and energy estimations. A name in array format such as *PE[0..15]* denotes duplicated instances of a sub-tree—in this instance, a total of 16 PEs, each with its own 16 bit integer MACC unit and register file.

Timeloop Mapper is a multi-threaded search space explorer. It spawns a given number of threads and each thread explores parts of search space until (1) it exceeded a specified maximum number of trials (timeout), or (2) it finds n consecutive non-optimal points compared to the current best solution (victory condition). In our evaluation, we provide much higher values than the default ones for both timeouts and victory conditions: 100000 for each, and termination after 3 hours.

V. Experimental Evaluation

In this section, we present an experimental evaluation of the effectiveness of the developed optimization methodology for dataflow-accelerator co-design optimization. We start with the architectural parameters of Eyeriss [6], a well known CNN accelerator that was designed and fabricated as a VLSI chip.

In our experimental evaluation, we ask the following question: *How much improvement in energy efficiency and performance is feasible over the Eyeriss design if we perform comprehensive* $\{architecture\} \times \{dataflow\}$ *design space exploration, while limiting the total area for the architectural components to that used by the original Eyeriss design?*

TABLE II
CONFIGURATIONS OF CONV2D OPERATORS, K: # OUTPUT CHANNELS; H, W: INPUT IMAGE HEIGHT AND WIDTH; C: #INPUT CHANNELS; R/S KERNEL SIZE; BATCH SIZE = 1; KERNEL STRIDE = 1/2 (2 IF MARKED WITH * AFTER RS, 1 OTHERWISE)

	Yolo				Resnet			
Layer	K	C	H W	R S	K	C	H W	R S
1	32	3	544	3	64	3	224	7*
2	64	32	272	3	64	64	56	3
3	128	64	136	3	64	64	56	1
4	64	128	136	1	128	64	56	3*
5	256	128	68	3	128	64	56	1*
6	128	256	68	1	128	128	28	3
7	512	256	34	3	256	128	28	3*
8	256	512	34	1	256	128	28	1
9	1024	512	17	3	256	256	14	3
10	512	1024	17	1	512	256	14	3*
11	28269	1024	17	1	512	256	14	1*
12					512	512	7	3

TABLE III
ARCHITECTURE PARAMETERS USED

Parameter	Value	Unit
Area per MAC	1239.5	μm^2
Area per register	19.874	μm^2
Area per SRAM word	6.806	μm^2
Energy per int16 MAC	2.2	pJ
Register energy-constant	9.06719×10^{-3}	$pJ/word$
SRAM energy-constant	17.88	
Energy per dram-access	128	pJ

We use all the convolutional layers of two DNN pipelines: Resnet-18 [10] and Yolo-9000 [19] for the evaluation. Table II shows the characteristics of the CNN stages in these DNN pipelines. We use both the DNN pipelines in evaluating energy efficiency and performance (delay minimization). In all the evaluations, the design parameters generated by Thistle are used to generate a Timeloop architecture specification (similar to that shown in Fig. 3(a)) and a Timeloop mapping (similar to that in Fig. 3(d)). Thus, while Thistle's architecture-dataflow co-design optimization is based on our generated models, the final reported energy/performance metrics are based on Timeloop's simulation of the architecture/dataflow produced by Thistle and not on Thistle's estimation of those metrics.

We use the architectural design parameters of the Eyeriss accelerator [6] as a baseline: 168 processors, 512 registers per processor, and 128 Kbytes of shared scratchpad SRAM memory. The original Eyeriss chip [6] was fabricated using 65nm VLSI technology. Since the closest VLSI technology for which architectural parameters were available in Accelergy [21] (used by the Timeloop accelerator modeling framework), we performed our evaluation using parameters for 45nm technology, as shown in Table III.

A. Energy Minimization

Energy optimization for Eyeriss architecture. Before assessing the impact of combined architecture-dataflow design space exploration on energy efficiency, we set up a baseline for the comparison: we performed dataflow optimization using the same architectural parameters as Eyeriss. We did this using Thistle as well as Timeloop's Mapper. Fig. 4 shows

the energy efficiency (picoJoules per multiply-accumulate) for all convolutional stages of the Resnet-18 and Yolo-9000 DNN pipelines. It may be seen that both Thistle and Timeloop's Mapper achieve similar energy efficiency, ranging between 20-30 pJ/MAC, with Thistle being slightly better.

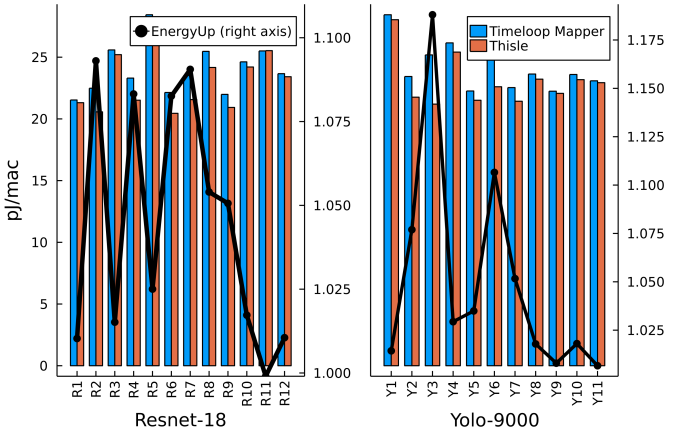

Fig. 4. Energy comparison between Timeloop Mapper and Thistle for Eyeriss architecture (lower is better). *EnergyUp* denotes $\frac{TimeloopEnergy}{ThistleEnergy}$.

Layer-wise architecture-dataflow co-design optimization. Next we explored architecture-dataflow co-design optimization using Thistle. For each CNN layer, we solved a collection of nonlinear optimization problems for different pruned combinations of tile-loop permutations, each involving the solution of an optimization problem similar to that illustrated in Eq. 5 for matrix multiplication. The best solution found by Thistle, of the form $\langle architecture\ parameters, tileloop\ permutations, tilesizes \rangle$, was used to generate Timeloop input and the Timeloop model was executed to generate the energy for execution of the CNN stage on the specified accelerator architecture.

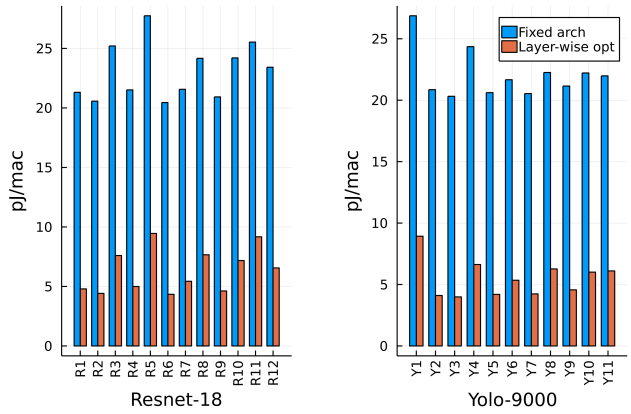

Fig. 5. Energy optimization: Eyeriss versus layer-wise optimized architecture with same area (lower is better)

Fig. 5 compares the achieved energy efficiency via architecture-dataflow co-design optimization with that of the best dataflow to minimize energy for the original Eyeriss architecture. It may be seen that a significant improvement is achieved for most of the stages of both the Resnet-18 and

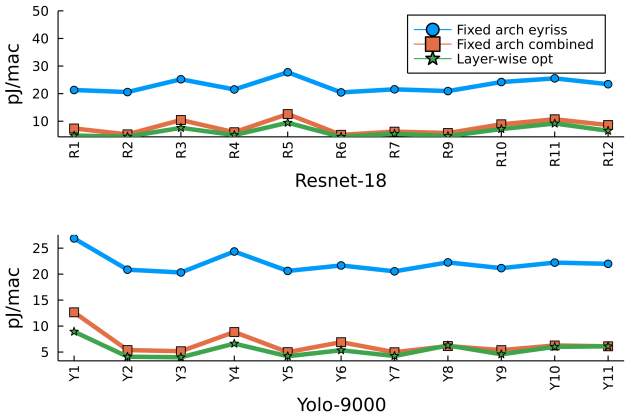

Fig. 6. Energy optimization: Eyeriss versus layer-wise optimal architecture versus fixed architecture optimized for energy-dominant layer (lower is better)

Yolo-9000 DNN pipelines. In contrast to the 20-30 pJ/MAC energy efficiency of the Eyeriss architecture, it is feasible to achieve a much better energy efficiency of around 5 pJ/MAC for most layers, and less than 10 pJ/MAC for all layers. For architecture-dataflow co-design optimization, we only show results based on Thistle in Fig. 5 because Timeloop Mapper can only perform dataflow optimization and cannot optimize across architecture parameters. The significant improvement in energy efficiency made possible by exploring the architecture design space underscores the importance of the new capability for rapid combined architecture-dataflow optimization that Thistle enables; we are unaware of any other accelerator optimization framework that can do so.

Energy optimization with single architecture for all layers. While the layer-wise optimization of the various stages of the Resnet-18 and Yolo-9000 yielded very significant improvements in energy efficiency over the original Eyeriss architecture, it is practically infeasible to customize an accelerator architecture for each layer of a CNN pipeline. We next assess the achievable energy efficiency for the stages of these two DNN pipelines under the constraint that a single common accelerator design must be created for execution of all the stages. We consider the stage consuming the most total energy among the individually layer-wise optimized accelerator designs (across both Resnet-18 and Yolo-9000) and use the architectural parameters corresponding to that stage. We then use Thistle to perform dataflow optimization for that fixed architecture. Fig. 6 shows the results. Each chart (one for Resnet-18 and one for Yolo-9000) shows three line graphs, plotting the energy efficiency across the various conv2D layers for (1) Eyeriss architectural parameters (blue line), (2) layer-wise optimized architecture (green line), and (3) fixed architecture corresponding to architecture optimization for energy-dominant layer (orange line). It may be seen that even with the requirement for a single accelerator design to execute all the stages, the energy improvements over the fixed Eyeriss architecture are very significant, with very few cases losing much in terms of energy efficiency when compared with the layer-specific optimized architecture.

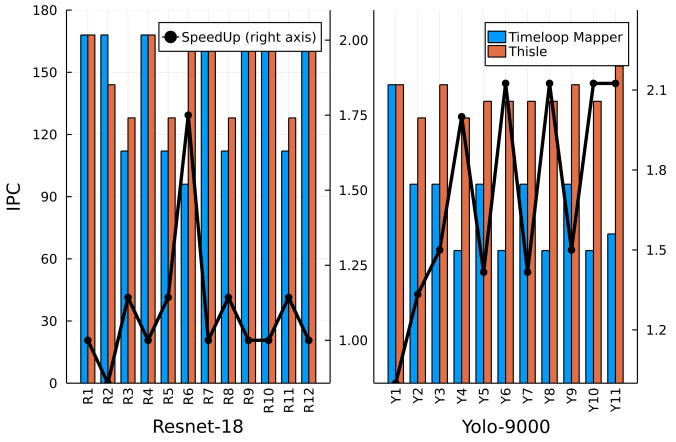

Fig. 7. Throughput comparison between Timeloop Mapper and Thistle for Eyeriss architecture (higher is better; theoretical maximum achievable IPC (MAC Instructions Per Cycle) value is the number of processors = 168)

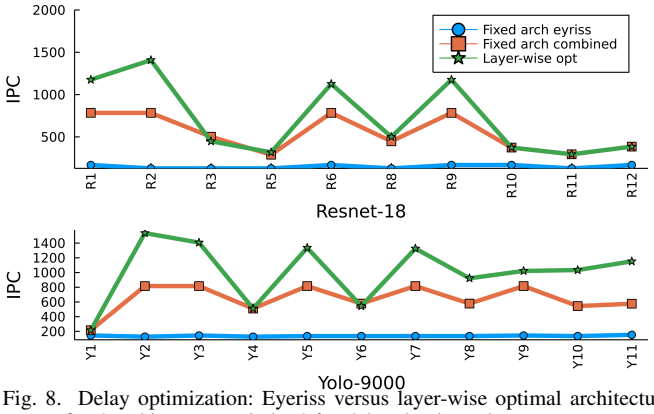

Fig. 8. Delay optimization: Eyeriss versus layer-wise optimal architecture versus fixed architecture optimized for delay-dominant layer

B. Delay Optimization

We can model delays associated with each component by multiplying the number of events by the throughput associated with the component. A difference from energy minimization is that instead of the sum of contributions from different components, the cost expression contains the maximum among the delays associated with each component.

Figure 7 denotes the MAC Instructions Per Clock (IPC) for the Eyeriss architecture, comparing optimized dataflows from Thistle and Timeloop Mapper. Deviating from the trends in the energy minimization results, the differences in achieved throughput from dataflow optimization by Thistle and Timeloop Mapper are more pronounced. (the line graph shows the improvement in speedup, with the scale shown on the right of the figure).

As for energy optimization, we performed layer-wise architecture-dataflow co-design for throughput optimization, identified the dominant-delay stage across all stages, and performed dataflow optimization for that fixed architecture. These results are shown in Fig. 8. The benefits of combined architecture-dataflow co-design for throughput are often orders of magnitude when compared to the fixed Eyeriss architecture. The drop from the throughput possible for a layer-optimized architecture to that achieved by a single architecture is also much higher than we observed for energy optimization. This is because energy optimization is not as sensitive to the total number of PEs in the accelerator as is the throughput.

VI. Related Work

Several prior efforts [3], [8], [12], [13], [14], [15], [18], [22] have addressed the problem of optimizing CNN on accelerators. dMazeRunner [8] provides an anlytical model to estimate energy and performance and conducts a search over a pruned search space. Interstellar [22] utilizes the Halide scheduling language to describe the design of the accelerator, and finds evergy-optimized designs by applying a pruned search guided by domain-specific knowledge. Interstellar only models energy optimization but does not model delay optimization.

Timeloop [18] is a comprehensive framework that models the energy/latency metrics of architecture mappings and has a Mapper that searches for optimized mapping over the space. We use TimeLoop as the accelerator modeling framework for evaluating our accelerator-dataflow optimization methodology.

Maestro [14] is an accelerator modeling framework that can evaluate the energy, latency, and throughput of a given mapping configuration on a given accelerator architecture. Several efforts have used the Maestro framework for dataflow and/or accelerator architecture optimization. Confuciux [12] uses reinforcement learning based search and genetic algorithms to determine optimized resource assignments for architecture parameters and dataflow. The GAMMA [13] system provides multiple optimization methods to drive the exploration of dataflow designs for a given accelerator architecture. Marvel decouples the dataflow design space into two parts—on-chip and off-chip—to accelerate design space exploration [4].

A key difference between the optimization approach we develop in this paper and all these prior efforts is that they all use a heuristic or iterative search through a bounded set of dataflow/accelerator configurations. In contrast, our presented approach formulates and solves a set of constrained non-linear optimization problems to simultaneously optimize architectural parameters for processor count, register count and SRAM capacity, along with the dataflow mapping.

VII. Conclusion

We present an efficient and effective approach to architecture-dataflow co-design for optimizing CNN on accelerator arrays. The key new idea is that the enormous design space can be comprehensively searched by automated synthesis and solution of a collection of constrained nonlinear optimization problems to find the combination of architectural parameters (number of registers per processor, capacity of shared on-chip memory, number of processing elements) and mapping choices (tile sizes at the register and shared-memory levels, parallelized dimensions, and tile loop permutations) that minimize energy (or delay). The experimental results demonstrate that significant improvement in energy efficiency and performance over prior designs can be achieved by use of the developed methodology.

REFERENCES

[1] Akshay Agrawal, Steven Diamond, and Stephen Boyd. Disciplined geometric programming. *Optimization Letters*, 13(5):961–976, 2019.

[2] Akshay Agrawal, Steven Diamond, and Stephen Boyd. Cvxpy: A rewriting system for convex optimization. 2021.

[3] Alessio Burrello, Angelo Garofalo, Nazareno Bruschi, Giuseppe Tagliavini, Davide Rossi, and Francesco Conti. Dory: Automatic end-to-end deployment of real-world dnns on low-cost iot mcus. *IEEE Transactions on Computers*, 2021.

[4] Prasanth Chatarasi, Hyoukjun Kwon, Natesh Raina, Saurabh Malik, Vaisakh Haridas, Angshuman Parashar, Michael Pellauer, Tushar Krishna, and Vivek Sarkar. Marvel: A data-centric compiler for dnn operators on spatial accelerators. *arXiv preprint arXiv:2002.07752*, 2020.

[5] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. Diannao: a small-footprint high-throughput accelerator for ubiquitous machine-learning. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*, pages 269–284, 2014.

[6] Yu-Hsin Chen, Tushar Krishna, Joel S Emer, and Vivienne Sze. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE journal of solid-state circuits*, 52(1):127–138, 2016.

[7] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, et al. Dadiannao: A machine-learning supercomputer. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 609–622. IEEE, 2014.

[8] Shail Dave, Youngbin Kim, Sasikanth Avancha, Kyoungwoo Lee, and Aviral Shrivastava. Dmazerunner: Executing perfectly nested loops on dataflow accelerators. *ACM Transactions on Embedded Computing Systems (TECS)*, 18(5s):1–27, 2019.

[9] Zidong Du, Robert Fasthuber, Tianshi Chen, Paolo Ienne, Ling Li, Tao Luo, Xiaobing Feng, Yunji Chen, and Olivier Temam. Shidiannao: Shifting vision processing closer to the sensor. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pages 92–104, 2015.

[10] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016.

[11] Qijing Huang, Aravind Kalaiah, Minwoo Kang, James Demmel, Grace Dinh, John Wawrzynek, Thomas Norell, and Yakun Sophia Shao. Cosa: Scheduling by constrained optimization for spatial accelerators. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 554–566. IEEE, 2021.

[12] Sheng-Chun Kao, Geonhwa Jeong, and Tushar Krishna. Confuciux: Autonomous hardware resource assignment for dnn accelerators using reinforcement learning. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 622–636. IEEE, 2020.

[13] Sheng-Chun Kao and Tushar Krishna. Gamma: automating the hw mapping of dnn models on accelerators via genetic algorithm. In *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pages 1–9. IEEE, 2020.

[14] Hyoukjun Kwon, Prasanth Chatarasi, Vivek Sarkar, Tushar Krishna, Michael Pellauer, and Angshuman Parashar. Maestro: A data-centric approach to understand reuse, performance, and hardware cost of dnn mappings. *IEEE micro*, 40(3):20–29, 2020.

[15] Linyan Mei, Pouya Houshmand, Vikram Jain, Sebastian Giraldo, and Marian Verhelst. Zigzag: A memory-centric rapid dnn accelerator design space exploration framework. *arXiv preprint arXiv:2007.11360*, 2020.

[16] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P Jouppi. Cacti 6.0: A tool to model large caches. *HP laboratories*, 1:1–24, 2009.

[17] NVIDIA Deep Learning Accelerator. http://nvdla.org/.

[18] Angshuman Parashar, Priyanka Raina, Yakun Sophia Shao, Yu-Hsin Chen, Victor A Ying, Anurag Mukkara, Rangharajan Venkatesan, Brucek Khailany, Stephen W Keckler, and Joel Emer. Timeloop: A systematic approach to dnn accelerator evaluation. In *2019 IEEE international symposium on performance analysis of systems and software (ISPASS)*, pages 304–315. IEEE, 2019.

[19] Joseph Redmon and Ali Farhadi. Yolo9000: better, faster, stronger. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 7263–7271, 2017.

[20] Yakun Sophia Shao, Brandon Reagen, Gu-Yeon Wei, and David Brooks. Aladdin: A pre-rtl, power-performance accelerator simulator enabling large design space exploration of customized architectures. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 97–108. IEEE, 2014.

[21] Yannan Nellie Wu, Joel S Emer, and Vivienne Sze. Accelergy: An architecture-level energy estimation methodology for accelerator designs. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8. IEEE, 2019.

[22] Xuan Yang, Mingyu Gao, Qiaoyi Liu, Jeff Setter, Jing Pu, Ankita Nayak, Steven Bell, Kaidi Cao, Heonjae Ha, Priyanka Raina, et al. Interstellar: Using halide's scheduling language to analyze dnn accelerators. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 369–383, 2020.