DeepOpt: Optimized Scheduling of CNN Workloads for ASIC-based Systolic Deep Learning Accelerators

Susmita Dey Manasi
University of Minnesota, Minneapolis, MN, USA

Sachin S. Sapatnekar
University of Minnesota, Minneapolis, MN, USA

ABSTRACT

Scheduling computations in each layer of a convolutional neural network on a deep learning (DL) accelerator involves a large number of choices, each of which involves a different set of memory reuse and memory access patterns. Since memory transactions are the primary bottleneck in DL acceleration, these choices can strongly impact the energy and throughput of the accelerator. This work proposes an optimization framework, DeepOpt, for general ASIC-based systolic hardware accelerators for layer-specific and hardware-specific scheduling strategy for each layer of a CNN to optimize energy and latency. Optimal hardware allocation significantly reduces execution cost as compared to generic static hardware resource allocation, e.g., improvements of up to 50× in the energy-delay product for VGG-16 and 41× for GoogleNet-v1.

CCS CONCEPTS

• **Hardware** → **Hardware accelerator**; **Operations scheduling**; • **Computing methodologies** → **Neural networks**.

KEYWORDS

CNN, scheduling, hardware accelerator

ACM Reference Format:
Susmita Dey Manasi and Sachin S. Sapatnekar. 2021. DeepOpt: Optimized Scheduling of CNN Workloads for ASIC-based Systolic Deep Learning Accelerators. In *26th Asia and South Pacific Design Automation Conference (ASPDAC '21), January 18–21, 2021, Tokyo, Japan.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3394885.3431539

1 INTRODUCTION

Inference tasks in deep convolutional neural networks (CNNs, or DNNs) involve massive amounts of data movement between the computation core and on-chip and off-chip memory. These transactions are a primary bottleneck in CNN performance and dominate the overall execution cost of a network [1, 2]. Standard CNN topologies consist of tens to hundreds of layers [3–6], where the sizes (i.e., data volumes) as well as configurations (i.e., data dimensions) of the layers vary widely even within the same network topology. The multidimensional structure of DNN layers offers multiple choices for mapping computations to hardware, which determine the level of data reuse, which in turn determine the memory access overheads. Prior works that map computations to systolic [7] or non-systolic accelerators [8–10] primarily opt for a fixed dataflow scheme for a network that is empirically tailored for the underlying hardware. Minerva [11] optimizes DNN hardware over a different optimization space than our work. The work in [12] does not systematically explore the scheduling space, but adopts different dataflows for coarse-grained layer classes. However, within each layer class, layer dimensions differ significantly and the optimal schedule may vary.

We introduce DeepOpt to methodically explore the search space for optimizing CNN computation by minimizing data access costs. DeepOpt specifically schedules operations in each layer in a given CNN, based on characteristics of the underlying hardware accelerator (e.g., on-chip SRAMs, number of parallel computational units). DeepOpt is open-sourced at github.com/manasiumn37/DeepOpt.

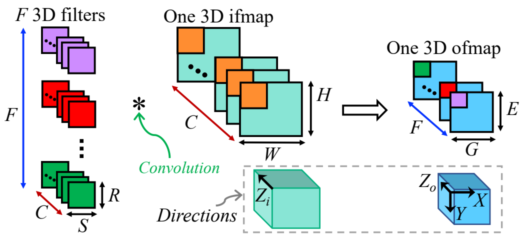

Figure 1: A *Conv* layer with ifmap, filter, and ofmap.

CNNs primarily consist of three types of layers: convolution (*Conv*), fully-connected (*FC*) and pooling (*Pool*), and computation is dominated by *Conv* layers. As illustrated in Fig. 1, each *Conv* layer convolves an input feature map (**ifmap**) with C channels of size $H \times W$ with a **filter** that has C channels of size $R \times S$, to produce an output feature map (**ofmap**) with F planes, each of size $E \times G$.

The convolution of the ifmap with a 3D filter involves element-wise multiplication between each channel of the 3D filter and a same-sized sub-region of the corresponding ifmap channel to produce intermediate partial sums (**psums**). For example, each channel of the first (purple) filter is multiplied by the shaded regions in each ifmap channel to produce the purple psum shown in the ofmap. We refer to this as a *unit operation*. Similar unit operations are performed on each of the F filters to generate psums for the F channels of the ofmap, as shown by the color-coded correspondence between filters and ofmap channels. For each filter, the weights in a channel slide through its corresponding channel of the ifmap with a convolution stride of U, and a multiply-accumulate (MAC) operation is used to add the generated psums. *Conv* layers may be followed by *Pool* layers that reduce the dimension of ofmap data.

Design Space for Scheduling Conv Layers For a given hardware configuration, unit ofmap operations can be scheduled as follows:

- X: taking strides of U, proceeding along the width of ofmap
- Y: taking strides of U, proceeding along the height of ofmap

ASPDAC '21, January 18–21, 2021, Tokyo, Japan
© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-7999-1/21/01…$15.00
https://doi.org/10.1145/3394885.3431539

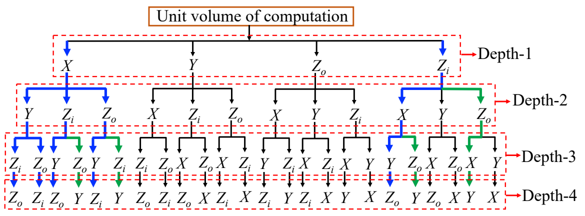

Figure 2: Computational search tree for convolution layers.

- Z_o: processing 3D filters sequentially along ofmap channels
- Z_i: processing each filter channel and the corresponding ifmap plane, proceeding along the channels of ifmap

These operations can be scheduled in many ways, as illustrated in Fig. 2 (blue and green paths are explained in Section 2.4). Each level of the tree (Depth-1 – Depth-4) indicates a direction for processing data and corresponds to a nesting of the computational loop: Depth-1 is the innermost loop and Depth-4 is the outermost. We will show how this tree can be pruned so that only the blue paths remain.

Table 1: Comparison of schedules XYZ_iZ_o vs. XZ_iYZ_o for *Conv2* and *Conv3* of AlexNet, under two hardware configurations. Cost function = Normalized memory access energy.

Hardware1, *Conv3*		Hardware1, *Conv2*		Hardware2, *Conv2*	
Winner	Penalty for loser	Winner	Penalty for loser	Winner	Penalty for loser
XYZ_iZ_o	39.2%	XZ_iYZ_o	50.3%	XYZ_iZ_o	60.0%

With limited on-chip storage, each scheduling scheme corresponds to a different set of memory access patterns for the ifmap, filter, and psum data. Since data access can be a primary determinant of CNN performance, the execution time and energy for CNN strongly depend on operation scheduling. Table 1 considers two candidate paths in the tree for computing *Conv2* and *Conv3* in AlexNet and uses our memory access models from Sections 2 and 3. For hardware configuration Hardware1, XYZ_iZ_o wins over XZ_iYZ_o for *Conv3*, but loses for *Conv2*: the energy penalty for using the other (losing) path is significant. If configuration Hardware2 is used, XYZ_iZ_o wins even for *Conv2*. Thus, the optimal branch in Fig. 2 is specific to the *Conv* layer and the hardware configuration.

In the innermost loop of the tree (Depth-1), multiple scheduling choices are possible, and data movement overheads are minimized when either the ifmap, ofmap, or filter data are kept stationary. By definition, one of these three is not possible for a given direction – e.g., computation along X proceeds along ifmap, and therefore ifmap cannot be stationary, leaving two choices for each direction in the innermost loop. DeepOpt analytically analyzes these choices.

The sequence of direction variables in the nested loops at Depth-2–Depth-4 creates additional scheduling choices. We develop methods to prune suboptimal choices, thus allowing a limited number of choices that can be exercised *at run-time*, and *individualized to specific layers*. Pruning the number of available choices is vital for reducing the hardware overhead of run-time configurability.

Target ASIC Hardware Platform We evaluate our scheduling framework on a general ASIC-based systolic hardware accelerator (Fig. 3) for DNN inference, similar to [13, 14]. The core of the accelerator is a two-dimensional $J{\times}K$ systolic array of processing elements (PE). Each PE contains a MAC unit and registers to hold one ifmap and one filter element locally, as well as a pipeline register to forward the computed psum data to the PE below. The architecture

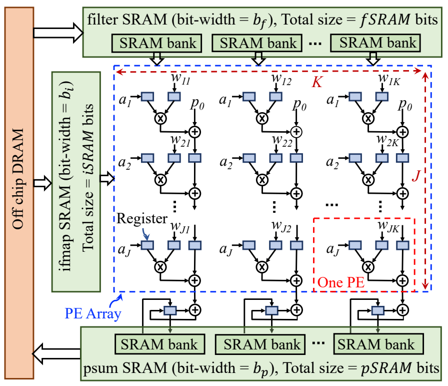
Off chip DRAM
filter SRAM (bit-width = b_f), Total size = $fSRAM$ bits
SRAM bank
SRAM bank
SRAM bank
ifmap SRAM (bit-width = b_i) Total size = $iSRAM$ bits
K
J
Register
One PE
PE Array
psum SRAM (bit-width = b_p), Total size = $pSRAM$ bits

Figure 3: Block diagram of the hardware architecture.

also consists of three separate on-chip SRAMs for filter, ifmap, and psum data, which communicate data with off-chip DRAM.

In each clock cycle, the PE array performs multiply-accumulate (MAC) operations between a $1{\times}J$ ifmap vector and a $J{\times}K$ weight matrix. In the PE array, a_i and w_{ij} denote individual elements of the ifmap vector and weight matrix, respectively. In the systolic flow, data in the ifmap vector are reused horizontally in each PE row and psums are accumulated vertically from top to bottom. Once the pipeline is full, the array produces a $1{\times}K$ vector of psums (one psum/column) per cycle. *All notation is visually illustrated in Fig. 3.*

2 THE OPTIMIZATION FRAMEWORK

The execution cost of a CNN consists of two primary components: (i) MAC computation in the PE array, and (ii) SRAM/DRAM data access. For a given PE array size, the MAC computation cost of a layer *is independent* of the choice of scheduling: scheduling merely changes the order of MAC operations, but not their total number. However, the data access cost from on-chip and off-chip memory can vary significantly based on the scheduling choice. For DNNs, the cost of arithmetic computation constitutes *less than 10%* of the total cost [2] and data access cost is the dominant component. DeepOpt finds an optimized schedule cost for each layer among the choices in the search tree (Fig. 2) while minimizing the number of accesses from the three on-chip SRAMs (for ifmap, filter, psum), and the off-chip DRAM. DeepOpt analytically models memory access patterns and prunes out branches from the search tree.

Offline, DeepOpt determines the schedule for each layer, for a given hardware configuration and target CNN topology. The layer-specific individualized schedule is exercised at runtime. DeepOpt applies this schedule on the computationally heavy *Conv* and *FC* layers while light layers (i.e., *ReLU*, *Pool*, etc.) are not optimized. The DeepOpt framework is also applicable for cross-layer fusion where *Conv*/*FC* computation is fused with an adjacent light layer.

2.1 Computation in the Search Tree

Unit volume of computation: The unit volume of computation in the PE array of Fig. 3 corresponds to the multiplication between the filter matrix and ifmap vector. Fig. 4 shows how computation in a *Conv* layer is mapped into a matrix-vector multiplication. The light blue boxes correspond to unprocessed data, and darker colored boxes are processed in the multiplication. The ifmap vector is formed using S ifmap elements from each $\lfloor J/S \rfloor$ ifmap channels (dark-colored region in the ifmap); S is small in a CNN so that $J > S$.

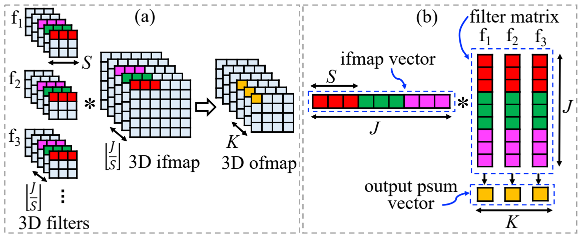

Figure 4: Unit volume of computation.

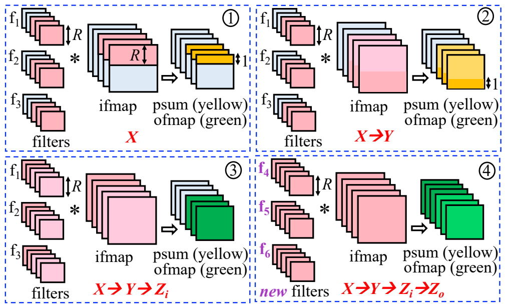

Figure 5: Example of computation using the XYZ_iZ_o branch.

The filter matrix is formed using data from K 3D filters, where each column of the matrix comes from $\lfloor J/S \rfloor$ channels of a single 3D filter (dark-colored region in each f_i filter). After the pipeline is filled (in typical networks, the pipeline setup overhead is small and can be neglected in practice), in each cycle, the systolic PE array performs the matrix-vector multiplication and produces K psums in K channels of the ofmap (yellow ofmap boxes).

The number of ifmap/filter channels, C_{uv}, and the number of 3D filters, F_{uv}, that are processed in parallel in the unit volume are

$$C_{uv} = \min\left(\lfloor J/S \rfloor, C\right) \quad ; \quad F_{uv} = \min(K, F) \tag{1}$$

The second terms cover the corner case where C and F are small.

Conceptual illustration: In boxes ① – ④ in Fig. 5, we show the computation for the XYZ_iZ_o branch. As before, unprocessed data are shown in light blue while darker colors refer to processed data.

Depth-1 (X direction, box ①): The innermost loop simultaneously performs matrix-vector multiplications between three filters (i.e., f_1, f_2, and f_3; $F_{uv} = 3$) and three pink ifmap/filter channels ($C_{uv} = 3$). Depending on the scheduling choice, the filter or the psum data can remain stationary. The process continues for multiple cycles until the pink ifmap and filter regions produce the yellow psum elements in the X direction, in the first row of ofmap.

Depth-2 (XY, box ②): In the next nested loop, the computation in box ① is repeated to cover all ofmap rows (the entire Y direction of ofmap), generating the yellow psums of the first F_{uv} ofmap planes.

Depth-3 (XYZ_i, box ③): At the next level of nesting, the computation in box ② is repeated to cover all ifmap/filter channels, processing the entire Z_i direction of the ifmap to produce the final (green) F_{uv} ofmap planes; these ofmaps are then moved to DRAM.

Depth-4 (XYZ_iZ_o, box ④): In the outermost loop, the computation in box ③ is repeated using the next F_{uv} sets of 3D filters until the full Z_o ofmap direction is covered, yielding the full ofmap volume.

2.2 Search Tree: Depth-1

We now begin to explore the search tree in Fig. 2, starting from Depth-1. As illustrated in Fig. 6, there are four candidate paths at Depth-1 of the tree, corresponding to directions X, Y, Z_o, and Z_i. During computation along these paths, one type of data (filter/ifmap/psum) can remain stationary in the PE array while the other two types change every cycle. Depending on which data is stationary, SRAM accesses vary, resulting in different path costs. At other levels of the tree, DRAM access patterns may vary across paths.

For each candidate (sub)path O in the search tree, we analyze the number of accesses per cycle to the filter SRAM, ifmap SRAM, and psum SRAM, denoted as $A^S_{f,O}$, $A^S_{i,O}$, and $A^S_{p,O}$, respectively.

Candidate subpaths for X: Along X, there are two ways to perform the computation: the unit volume of computation repeats in every cycle to convolve the pink ifmap and filter region of Fig. 6(a) to produce the yellow psum rows in the ofmap, either according to Fig. 6(e) or Fig. 6(f). We refer to these as X_a and X_b, respectively.

Cycle count: For X_a and X_b, in each cycle, every PE column outputs a psum element. This requires G cycles to generate the psums for an ofmap row, repeated for R filter/ifmap rows. Ignoring the small pipeline setup overhead, this adds up to $(G \times R)$ cycles.

filter SRAM accesses: For X_a in Fig. 6(e), each filter is loaded only once from the SRAM to PE array: it remains stationary while the ifmap row slides through. Hence, the number of filter SRAM accesses is the total number of filter elements in the computation, $(R \times S \times C_{uv} \times F_{uv} \times b_f)$. Thus, the number of accesses per cycle is

$$A^S_{f,X_a} \quad = \quad (S \times C_{uv} \times F_{uv} \times b_f)/G \tag{2}$$

For X_b in Fig. 6(f), the unit volume of computation changes the filter matrix in each cycle, requiring a total of $(S \times C_{uv} \times F_{uv} \times b_f)$ loads to the filter SRAM per cycle, i.e., $A^S_{f,X_b} = G \times A^S_{f,X_a}$.

ifmap SRAM accesses: For X_a, while the filter matrix remains stationary, the ifmap vector changes every cycle. In each cycle, $(S \times C_{uv})$ ifmap elements are loaded from the ifmap SRAM to the PE array, so that the number of ifmap SRAM accesses per cycle is

$$A^S_{i,X_a} \quad = \quad S \times C_{uv} \times b_i \tag{3}$$

For X_b, the ifmap also changes every cycle, and $A^S_{i,X_b} = A^S_{i,X_a}$.

psum SRAM accesses: For X_a, a psum operation involves G psum data for F_{uv} PE columns. The number of accesses/operation is

$$a^S_{p,X_a} = G \times F_{uv} \times b_p \tag{4}$$

For the first filter/ifmap row, only a write access is required, but for the remaining $(R-1)$ filter/ifmap rows, the psum must be both read and written. Therefore, a total $(1 + 2(R-1)) = 2R - 1$ sets of accesses are required, so that the number of accesses per cycle is

$$A^S_{p,X_a} = (2R-1) \times a^S_{p,X_a}/(G \times R) = (2R-1)F_{uv} \times b_p/R \tag{5}$$

For X_b, since the unit volume of computation moves to the subsequent filter/ifmap row in every cycle, psum data are written once to the SRAM after accumulating all the psums over the $(R \times S)$ window. Thus, after every R cycles one psum write to SRAM occurs from each of the F_{uv} PE columns. Over $(G \times R)$ cycles, the number of psum SRAM accesses for X_b is same as (4). Thus,

$$A^S_{p,X_b} = F_{uv} \times b_p/R \tag{6}$$

The number of accesses for subpaths Y, Z_o, Z_i can be similarly derived, as outlined below and summarized in Table 2.

Candidate subpaths for Y: The possibilities for scheduling computations are equivalent to those for subpath X, except that all

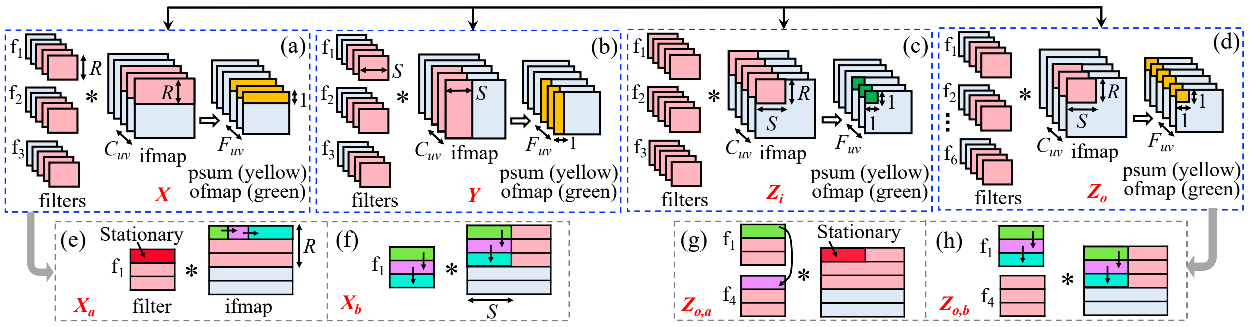

Figure 6: Illustration of computation using the four candidate paths at Depth-1.

operations are transposed with respect to the ifmap. Therefore, the two possibilities, Y_a and Y_b, parallel those for the previous case, and the number of accesses is identical to X_a and X_b, respectively.

Candidate subpath Z_i: The choice of moving along Z_i requires the psum to be stationary, so that only an output-stationary scenario exists. Fig. 6(c) shows option Z_i, where the data processed by this path is shown by the pink ifmap and filter region, which produces the green ofmap elements. Since each cycle processes a new ifmap/filter row that must be loaded, the number of filter and ifmap SRAM accesses for this path are identical to X_b. Since the ofmap is fully constructed, one psum write operation is required per PE column, i.e., $F_{uv} \times b_p$ accesses to the psum SRAM (we omit a second mapping with the same filter, ifmap, and psum access costs).

Each cycle processes one filter/ifmap row from each of the C_{uv} channels, and this is repeated for R rows. All C channels are covered in $R \times (C/C_{uv})$ cycles. Therefore, the number of psum SRAM accesses per cycle is $(F_{uv} \times b_p)/[R \times (C/C_{uv})]$, i.e., $A^S_{p,Z_i} = A^S_{p,X_b}/(C/C_{uv})$. Note that from Equation (1), $C/C_{uv} \geq 1$.

Candidate subpaths for Z_o: The pink ifmap and filter regions in Fig. 6(d) are processed to produce the yellow psum element in each channel of the ofmap. Under one possible schedule, shown in Fig. 6(g), after every cycle the unit volume of computation processes rows from a new set of F_{uv} 3D filters while the ifmap data remains stationary in the PE array and the filter matrix changes every cycle.

The ifmap access count is $(R \times S \times C_{uv} \times b_i)$, and the ifmap data are used for $(F/F_{uv} \times R)$ cycles. Therefore, the average number of accesses per cycle, $A^S_{i,Z_{o,a}} = (S \times C_{uv} \times b_i)/(F/F_{uv})$, i.e.,

$$A^S_{i,Z_{o,a}} = A^S_{i,X_a}/[F/F_{uv}] \tag{7}$$

using (3). From (1), $F/F_{uv} \geq 1$. The filter data is loaded in each cycle, as in the case of X_b, and therefore, $A^S_{f,Z_{o,a}} = A^S_{f,X_b} = G \times A^S_{f,X_a}$. The psum accesses are shown in Table 2 (details omitted).

Alternatively, $Z_{o,b}$ in Fig. 6(h), produces the yellow ofmap psum elements with row-wise processing of $(R \times S)$ windows of the ifmap and F_{uv} filters. The process is repeated to cover all filters. The SRAM access patterns are identical to those for X_b: both the filter matrix and ifmap vector change every cycle; one psum element from each of the F_{uv} PE columns is written to the SRAM after every R cycles, after accumulating psums over the $(R \times S)$ window.

2.3 Pruning Paths at Depth-1

All four paths at Depth-1 cover small data volumes where the computations require only a single fetch per data from the DRAM to SRAM. Hence, the four candidate paths at Depth-1 are equivalent in terms of DRAM accesses, and it is the SRAM access that differentiates one path from another at this depth of the tree.

Table 2: Filter, ifmap, psum SRAM access count (Depth-1).

Direction	X_a, Y_a	$X_b, Y_b, Z_{o,b}$	Z_i	$Z_{o,a}$
filter	Eq. (2)	Eq. (2) $\times G$	Eq. (2) $\times G$	Eq. (2) $\times G$
ifmap	Eq. (3)	Eq. (3)	Eq. (3)	$\frac{\text{Eq. (3)}}{(F/F_{uv})}$
psum	Eq. (5)	$\frac{\text{Eq. (5)}}{(2R-1)}$	$\frac{\text{Eq. (6)}}{(C/C_{uv})}$	Eq. (5)

Using Table 2, we analytically compare the four paths. Depth-1 is in the innermost loop of the computation. Regardless of the mapping scheme, the number of MAC operations across the four iterative loops is identical and equals the product of the number of iterations at each level. The total ifmap/filter/psum SRAM access cost is the product of this constant and the number of operations.

Hence, given two mapping schemes at Depth-1, if one scheme has higher memory access cost *per operation* over the ifmap SRAM, filter SRAM, and psum SRAM, it can be said to be provably suboptimal and can be pruned from the search. We make a mild approximation of counting all SRAM accesses equally, since all SRAM sizes are similar. For pruning, this approximation is reasonable in practice.

Candidate subpaths $(X_b, Y_b, Z_{o,b})$ vs. Z_i: From Table 2, both paths have same SRAM access pattern for the filter and ifmap data. However, subpaths $X_b, Y_b, Z_{o,b}$ incur C/C_{uv} times higher cost for the psum data than Z_i, implying that subpath Z_i is always better. *Therefore, we prune out choices $(X_b, Y_b, Z_{o,b})$ at Depth-1.*

Candidate paths (X_a, Y_a) vs. $Z_{o,a}$: Table 2 shows that these subpaths have the same SRAM accesses for psum; $Z_{o,a}$ incurs G times higher filter access; X_a, Y_a incurs F/F_{uv} times higher ifmap access. Thus, (X_a, Y_a) is a better choice than $Z_{o,a}$ when

$$(A^S_{f,X_a} + A^S_{i,X_a}) < (A^S_{f,X_a} \times G) + (A^S_{i,X_a}/[F/F_{uv}]) \tag{8}$$

Using analytical methods using Equations (2) and (3), we can show that this translates to $F_{uv} \geq b_i/b_f$. Typically, $F \gg K$, and $F_{uv} = K$, and the above condition translates to $b_i \leq K \times b_f$. Typically, the bit-widths are similar, but a typical value of K is 8 or larger, so that the condition is always satisfied. *Hence, direction Z_o is pruned out.*

Candidate paths (X_a, Y_a) vs. Z_i: These two paths have same ifmap data access pattern. However, path (X_a, Y_a) incurs $(2R - 1) \times C/C_{uv}$ times higher access for psum data while path Z_i incurs G times higher access for the filter data. Therefore, none of these paths are pruned out and we explore them at further tree levels.

The result of pruning is that only X_a (henceforth referred to as X) and Z_i remain unpruned at Depth-1 of the tree in Fig. 2.

2.4 Search Tree: Depth-2 through Depth-4

As discussed in Section 2.2, the SRAM access patterns are determined at Depth-1. Choices at Depth-2 – Depth-4 do not change SRAM patterns, but may have different DRAM access counts.

At Depth-2, the amount of data processed by typical networks is small enough that it does not have to be fetched more than once from the DRAM to the SRAM. Therefore, since the DRAM access overheads are similar, none of the paths can be pruned out at Depth-2. For a 16×16 8-bit array, for AlexNet, VGG16, GoogleNet-v1, and SqueezeNet-v1.1, SRAM sizes of $1 - 4$kB each are sufficient.

Our analysis yields **nine surviving options** at Depth-4 (highlighted paths with blue and green in Fig. 2):
With X at Depth-1, the three choices at Depth-2 are XY, XZ_i and XZ_o. Each has two choices at the two remaining depths (e.g., XY can choose $\{Z_iZ_o, Z_oZ_i\}$ at the next two levels; XZ_i can expand to $\{YZ_o, Z_oY\}$, and XZ_o to $\{YZ_i, Z_iY\}$), for a total of **six options**.
With Z_i at Depth-1, it can similarly be shown that there are six choices, of which three have identical cost because of the X, Y symmetry, leaving **three options**, Z_iXYZ_o, Z_iXZ_oY, and Z_iZ_oXY.

The total number of SRAM accesses, for each type of access, corresponds to the number of SRAM accesses per cycle for X or Z_i at Depth-1 (listed in Table 2), multiplied by the number of iterations in the four nested loops; this multiplier is given by:

$$(R \times G) \times (E) \times (C/C_{uv}) \times (F/F_{uv}) \tag{9}$$

i.e., the product of the iteration counts of the X, Y, Z_i, and Z_o loops.

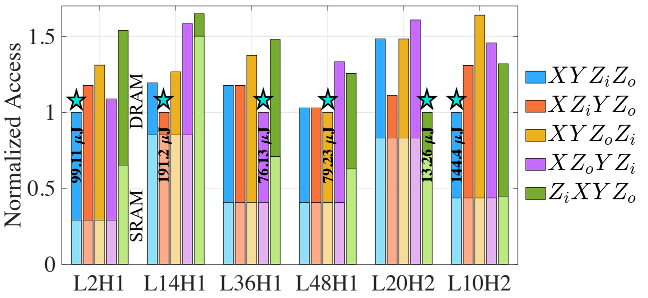

Figure 7: Normalized memory (SRAM + DRAM) energy for six different layers of GoogleNet-v1 with two hardware configurations, for five branches in the search tree. (Ln = Layer number. **H1**: b_f=8, $pSRAM$=32kB. **H2**: b_f=4, $pSRAM$=8kB. **H1 and H2**: b_i=8, b_p=32, J=16, K=32, $fSRAM$=64kB, $iSRAM$=32kB.)

2.5 Off-chip DRAM Access

We motivate the impact of DRAM access patterns by showing the effect of choosing different paths in the tree. These results are shown for two hardware configurations. For layers (L2, L14, L36, L48, L20, and L10) of GoogleNet-v1, Fig. 7 shows the combined energy cost for SRAM (lighter shade) and DRAM (darker shade) accesses for psum (SRAM only), ifmap, and filter data. Each set of bars is normalized to the smallest in the class where the absolute energy value associated with the smallest (i.e., the value of 1) is annotated in the figure. Following [15], one DRAM access is taken to have 33.3× the energy of one SRAM access. The results are executed on hardware H1 and H2, using various candidate branches. The optimal choice in each case is marked with a star. The most notable aspect is to see that *the optimal path is different for each case.*

The DRAM access pattern of a branch depends on various conditionals which primarily represent how the hardware configuration (i.e, sizes of on-chip SRAM, PE array size) and the shape of a layer can play together to alter the number of off-chip accesses. The DRAM is used to load ifmap and filter data (psum data stays in the SRAM). Due to space limits, we illustrate the number of DRAM accesses for one example branch, XYZ_iZ_o (Fig. 5). Similar results are derived for other branches; the precise derivations for some branches (e.g., XZ_iYZ_o) involve the enumeration of more cases.

To reduce costly off-chip accesses, DNN accelerators [2, 13, 16] typically move psums between the on-chip SRAM and the PE array without going to the off-chip DRAM. After psums are accumulated along the full Z_i direction, the ofmap is written to DRAM. Due to the limited size, *pSRAM*, of on-chip psum SRAM, in some cases the entire dimension $\phi \in \{H, W, E, G, F\}$ cannot be processed at a time: we then replace ϕ by a value ϕ_{eff} that reflects the actual processed dimension. For example, for branch XYZ_iZ_o, if the psum SRAM is not large enough for the entire row, G is replaced by

$$G_{\mathit{eff}} = \min\left(G, \lfloor pSRAM/(F_{uv} \times b_p) \rfloor\right) \tag{10}$$

and the computation is repeated over $\lceil G/G_{\mathit{eff}} \rceil$ iterations. Similarly, the entire Y dimension of E rows may have to be replaced by an iteration over $E_{\mathit{eff}} = \min(E, \lfloor pSRAM/(G_{\mathit{eff}} \times F_{uv} \times b_p) \rfloor)$. Thus, the DRAM loads the SRAM $n_i = \lceil G/G_{\mathit{eff}} \rceil \times \lceil E/E_{\mathit{eff}} \rceil$ times.
DRAM access for filter data: Through the sliding window operation, filter data are reused across the width and height of an ifmap. For branch XYZ_iZ_o, the inner loops are over X and Y, which allows maximal filter reuse by loading all filter data only once from the DRAM. Thus, the filter requires $V_{4Df} \times n_i$ DRAM accesses, where $V_{4Df} = (R \times S \times C \times F \times b_f)$ is the full 4D filter data volume.
DRAM access for ifmap data: If the ifmap SRAM is large enough to fit the full 3D volume of an ifmap during an iteration, then each ifmap element is loaded just once from the DRAM, and thus, the ifmap requires $V_{ifmap} \times n_i$ DRAM accesses. However, if the SRAM is not large enough for the full ifmap, ifmap data is reloaded $\lceil F/F_{uv} \rceil$ times for the computation using each F_{uv} sets of 3D filter and, the number of DRAM accesses for XYZ_iZ_o is

$$A_{D_i - XYZ_iZ_o} = V_{ifmap} \times \lceil F/F_{uv} \rceil \times n_i \tag{11}$$

2.6 Final Set of Candidates After Pruning

In Section 2.4, the 24 options in the search tree of Fig. 2 were reduced to nine. For four of these branches (XZ_iZ_oY, XZ_oZ_iY, Z_iXZ_oY, Z_iZ_oXY), where both Z_i and Z_o appear at Depth-3 or earlier, in order to process the first ofmap row (i.e., data up to Depth-3), the filter SRAM must work with all the channels of all 3D filters, i.e., the entire 4D filter volume. This is typically too large for an on-chip filter SRAM and requires repeated replacement from the DRAM, incurring high cost, significantly more than any other branch. After, we prune out these four branches, **five surviving options** remain: XYZ_iZ_o, XZ_iYZ_o, XYZ_oZ_i, XZ_oYZ_i, and Z_iXYZ_o.

3 ANALYZING ENERGY AND PERFORMANCE

3.1 Energy Computation

The total energy of a layer, E_L, consists of four components:

$$E_L = E_{MAC} + E_R + E_S + E_D \tag{12}$$

where E_{MAC} indicates the computation energy of a layer to perform the MAC operations in the PE array, E_R is the energy to access data from the registers in the PEs, while E_S [E_D] denotes the data access energy from the SRAMs [DRAM]. E_{MAC} is computed by multiplying the total number of MAC operations in a layer, N_{MAC}, by the energy per MAC operation, e_{MAC}, i.e.,

$$E_{MAC} = N_{MAC} \times e_{MAC} \tag{13}$$

Since each ofmap entry requires $R \times S \times C$ operations, $N_{MAC} = (R \times S \times C) \times (E \times G \times F)$. A MAC operation is accompanied by four register accesses in a PE: three filter, ifmap, and psum reads and one psum write. Therefore, the register access energy is

$$E_R \quad = \quad N_{MAC} \times (b_f + b_i + 2b_p) \times e_R \tag{14}$$

where e_R is the energy for per-bit register access. Denoting e_S and e_D as the energy for per bit SRAM and DRAM access, respectively, the SRAM and DRAM data access energy for each branch is

$$E_S = e_S \times \textstyle\sum_{\alpha=f,i,p} A_{S_\alpha - O} \tag{15}$$

$$E_D = e_D \times \left(\textstyle\sum_{\alpha=f,i} A_{D_\alpha - O} + (E \times G \times F \times b_i)\right) \tag{16}$$

where $A_{S_\alpha - O}$ ($A_{D_\alpha - O}$) denotes the total number of SRAM (DRAM) accesses for branch O for psum (SRAM only), filter, and ifmap data. For the SRAMs, this can be obtained as the product of the access per cycle in Section 2.2 and the multiplication factor over all four nested loops (given by (9)), and for the DRAM, using the procedure in Section 2.5. The last term in (16) represents the energy to write back the final ofmap data to the DRAM.

3.2 Inference Delay/Performance Computation

The total inference delay for a layer is the sum of two components: (i) **Computation cycles**, i.e., the number of cycles required to perform the MAC operations in the PE array, given by:

$$D_{Comp} = N_{MAC}/((C_{uv} \times S) \times F_{uv}) \tag{17}$$

where the denominator determines the #of PEs working in parallel, with parameters C_{uv} and F_{uv} are computed using (1). The number of computation cycles largely depends on the size of the PE array and does not change based on the choice of scheduling.

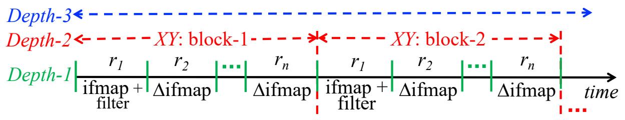

Figure 8: Sequence of computation for branch XYZ_iZ_o.

(ii) **DRAM stall cycles** during which the systolic PE array remains idle, waiting for data to be fetched from the off-chip DRAM. Typically, due to the limited off-chip bandwidth and the high volume of DRAM transfer required to process a layer, there can be significant memory transfer delays that affect the overall throughput of computation. For a given off-chip bandwidth, the scheduling choice determines the number of DRAM accesses to compute a layer, and hence, the number of DRAM stall cycles.

DeepOpt overlaps DRAM access cycles with computation cycles to reduce the number of DRAM stall cycles, as shown in Fig. 8 for branch XYZ_iZ_o (other branches are handled similarly). The figure shows the granularity of computations for Depth-1 – Depth-3, with ofmap rows r_i processed at Depth-1, multiple rows in C_{uv} channels at Depth-2 (reusing filter data), and new filter channels at Depth-3.

Filter data is fetched once every n rows. For the ifmap, new data is always fetched for r_1 but with each stride, older data that overlaps with previous computations is reused and only some data, Δifmap, is fetched. Thus, the memory overhead for r_1 always exceeds that for any other r_i. Data fetch for each cycle overlaps with computation from the previous cycle, and if data access time exceeds computation time, a stall is required. The process is repeated to cover all the 3D filters at Depth-4 to determine the total DRAM stall cycles.

4 RESULTS

4.1 Layer-Specific Optimal Scheduling (LOS)

We use DeepOpt to evaluate the layer-specific optimal scheduling (LOS) using five CNN topologies: AlexNet, VGG16, GoogleNet-v1, SqueezeNet-v1.1, and ResNet-50. We perform our evaluation under 65nm technology parameters. For SRAMs, the access energy vs. size trend is obtained from CACTI [17]. The access energy/bit for DRAM, SRAM, and register, and per unit energy for addition/multiplication are obtained from [10, 15, 18]. The bandwidth of off-chip DRAM is 128 bits/cycle. The five non-pruned branches (XYZ_iZ_o, XZ_iYZ_o, XYZ_oZ_i, XZ_oYZ_i, and Z_iXYZ_o[1]) are considered for scheduling. The hardware is parameterized by PE array ($J \times K$), and SRAM sizes ($fSRAM$, $iSRAM$, $pSRAM$) and bit-widths (b_i, b_f, b_p).

DeepOpt uses a user-specified performance metric (PM) as an optimization objective. This could be the Energy, Delay, Energy×Delay, Energy^2×Delay, or Energy×Delay^2. *Here, the Delay represents the per image inference time of a network.* Similar performance metric is used for the evaluation of DNN in [15]. For a given network topology and an underlying hardware accelerator configuration, DeepOpt uses layer-specific scheduling to determine the branch for each layer within the network that optimizes the specified PM.

Table 3: Penalty of FS as compared to LOS (GoogleNet-v1).

Fixed scheduling (FS) scheme	Penalty relative to layer-specific optimal scheduling (LOS)				
	Energy	Delay	Energy× Delay	Energy^2× Delay	Energy× Delay^2
XYZ_iZ_o	11.7%	19.4%	44.7%	77.4%	89.1%
XZ_iYZ_o	12.5%	33.8%	22.4%	9.5%	16.4%
XYZ_oZ_i	15.4%	27.2%	61.3%	107.5%	132.0%
XZ_oYZ_i	32.2%	66.6%	89.5%	120.0%	152.3%
Z_iXYZ_o	613.7%[1]	(large)[1]	(large)[1]	(large)[1]	(large)[1]

Table 3 shows the percent penalty on the total execution cost of GoogleNet-v1 when all layers of the network are computed using a fixed scheduling (FS) scheme, chosen from the five good, non-pruned schemes of Section 2.6, as compared to the optimal LOS from DeepOpt. The evaluations are shown across all five PMs for an accelerator specification of: $b_i = b_f = 8$, $b_p = 32$, $J \times K = 32 \times 64$, $fSRAM$ = 16kB, $iSRAM$ = 32kB, $pSRAM$ = 8kB. Note that the per-layer optimal configuration in each column is different, and therefore, for example, the optimal (Energy×Delay) is not the same as the optimal(Energy) × the optimal(Delay): the latter is a lower bound that is achievable only when the optimal Energy and optimal Delay configurations are identical (this is typically not the case). As seen from the table, across all five PMs, the penalty for each of

[1]In practice, off-chip access of Z_iXYZ_o greatly exceeds other four branches unless the filter SRAM can fit ($R \times S \times C \times F_{uv} \times b_i$) volume of data. When this large storage constraint is not met, we evaluate this branch for Energy and not the other PMs.

the FS schemes with respect to LOS is significant, e.g., up to 90% for Energy×Delay if FS scheduling choices are not explored.

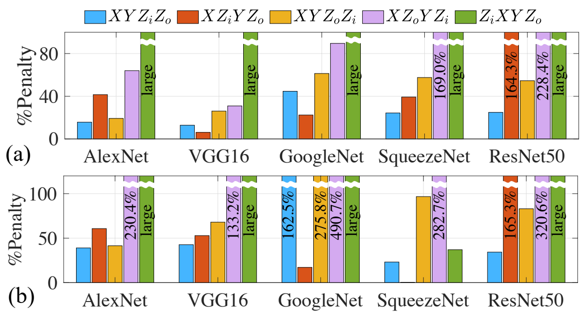

Figure 9: Energy×Delay penalty of FS schemes vs. LOS for two hardware configurations (for Z_iXYZ_o, see Footnote 1).

Similar trends are seen for AlexNet, VGG16, SqueezeNet-v1.1, and ResNet-50 as summarized in Fig. 9(a) for the same hardware configuration: due to space constraints, only the results for Energy×Delay are shown. For a second hardware configuration ($b_i = b_f = 16$, $b_p = 32$, $J \times K = 64 \times 32$, $fSRAM$ = 64kB, $iSRAM$ = 32kB, $pSRAM$ = 8kB), Fig. 9(b) shows the Energy×Delay penalties for all the FS schemes over LOS for these networks. For this hardware, most layers in SqueezeNet-v1.1 opt for the same configuration (XZ_iYZ_o) under LOS, due to which the penalty of FS is small for this choice.

4.2 Optimal Allocation of Hardware Resources

We use DeepOpt's layer-specific optimal scheduling (LOS) to perform design space exploration, optimally selecting hardware resources under an area budget of 16mm^2, permitting a deviation of 10% to allow greater flexibility in choosing the optimum. We set $b_f = b_i = 8$, and $b_p = 32$ [13]. The area per PE is obtained from a post place-and-route implementation of the systolic array, and SRAM area is estimated using a commercial 65nm memory compiler.

Table 4 shows the optimal allocation of the accelerator parameters for GoogleNet-v1 across all five PMs, and shows the impact of optimal allocation over an arbitrary approach. The rightmost column shows the reduction in the PM at the optimal hardware point, as compared to the worst-case allocation, and thus represents the performance range from best-case to worst-case allocation: e.g., if the allocation is chosen without careful analytical modeling, as in this work, the Energy×Delay product could be up to 40.71× higher. Similar degradations are seen for other PMs. For AlexNet, VGG16, SqueezeNet-v1.1, and ResNet-50, Energy×Delay deteriorates from best to worst case by 12.0×, 50.1×, 21.6×, and 20.2×, respectively.

Table 4: Optimal vs. worst-case hardware allocation for GoogleNet-v1 under a 16mm^2 area budget.

Performance metric (PM)	Optimal hardware allocation					Improvement over worst-case allocation
	J	K	SRAM size (kB)			
			filter	imap	psum	
Energy×Delay	48	32	32	64	32	40.7×
Energy^2×Delay	48	32	64	64	4	87.7×
Energy×Delay^2	48	32	64	64	8	1134.5×
Energy	38	32	64	128	4	2.0×
Delay	48	36	16	16	64	17.1×

Finally, Table 5 quantifies the suboptimality in running a different network on hardware tuned to optimize another network, e.g., running AlexNet on hardware tuned for the other networks. It can be seen that AlexNet, VGG16, and GoogleNet-v1 exhibit a small penalty on hardware tuned for any of the other two, but the penalty is much higher when SqueezeNet-v1.1 or ResNet-50 is run on hardware tuned on any of these three, or vice versa. This is not surprising: SqueezeNet-v1.1 has fewer parameters and a smaller model size, while ResNet-50 is a deeper model with residual building blocks. DeepOpt successfully finds these separate optima.

Table 5: Percent penalty on Energy×Delay when a network is executed on the optimal hardware for another network.

Optimal hardware $J/K/fSRAM/$ $iSRAM/pSRAM$*	Percent penalty				
	Alex-Net	VGG16	Google-Net-v1	Squeeze-Net-v1.1	ResNet-50
AlexNet: 50/32/16/32/64	0%	3.18%	3.92%	34.20%	7.04%
VGG16: 54/32/16/16/64	1.01%	0%	5.72%	32.86%	11.18%
GoogleNet-v1: 48/32/32/64/32	3.69%	2.64%	0%	22.79%	9.83%
SqueezeNet-v1.1: 40/34/16/128/32	8.77%	22.92%	20.87%	0%	15.33%
ResNet-50: 40/32/32/32/128	8.58%	28.33%	22.79%	24.63%	0%

Unit of $fSRAM$, $iSRAM$, and $pSRAM$ is in kB.

5 CONCLUSION

A systematic approach for scheduling computations on a DNN, minimizing the overhead of costly memory accesses, has been proposed. It is shown that substantial savings are possible by using layer-specific optimized scheduling as compared to fixed scheduling schemes which use popular weight/output stationary methods in the innermost loop. Design space exploration is performed to optimally tune hardware resources for running a specific DNN.

ACKNOWLEDGMENTS

We thank Z. Wang and A. B. Kahng (UCSD) for helping in modeling SRAM area. This work is supported in part by NSF (CCF-1763761).

REFERENCES

[1] V. Sze, *et al.*, "Efficient Processing of Deep Neural Networks: A Tutorial and Survey," *Proc. of the IEEE*, vol. 105, pp. 2295–2329, Dec. 2017.
[2] Y. H. Chen, *et al.*, "Eyeriss: An Energy-efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks," *IEEE Journal of Solid-State Circuits*, vol. 52, pp. 127–138, Jan. 2017.
[3] A. Krizhevsky, *et al.*, "ImageNet Classification with Deep Convolutional Neural Networks," in *Proc. Adv. NIPS*, pp. 1097–1105, 2012.
[4] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *Proc. ICLR*, 2015.
[5] C. Szegedy, *et al.*, "Going Deeper with Convolutions," in *Proc. CVPR*, Jun. 2015.
[6] K. He, *et al.*, "Deep Residual Learning for Image Recognition," in *Proc. CVPR*, 2016.
[7] A. Samajdar, *et al.*, "SCALE-Sim: Systolic CNN Accelerator Simulator," *arXiv:1811.02883*, 2018.
[8] Y. Ma, *et al.*, "Performance Modeling for CNN Inference Accelerators on FPGA," *IEEE T. Comput. Aid. D.*, vol. 39, pp. 843–856, April 2020.
[9] X. Yang, *et al.*, "Interstellar: Using Halide's Scheduling Language to Analyze DNN Accelerators," in *Proc. ASPLOS*, pp. 369–383, 2020.
[10] S. D. Manasi, *et al.*, "NeuPart: Using Analytical Models to Drive Energy-Efficient Partitioning of CNN Computations on Cloud-Connected Mobile Clients," *IEEE T. VLSI Syst*, vol. 28, pp. 1844–1857, Aug. 2020.
[11] B. Reagen, *et al.*, "Minerva: Enabling Low-Power, Highly-Accurate Deep Neural Network Accelerators," in *Proc. ISCA*, pp. 267–278, 2016.
[12] H. Kwon, *et al.*, "Understanding Reuse, Performance, and Hardware Cost of DNN Dataflow: A Data-Centric Approach," in *Proc. MICRO*, pp. 754–768, 2019.
[13] N. P. Jouppi *et al.*, "In-datacenter Performance Analysis of a Tensor Processing Unit," in *Proc. ISCA*, pp. 1–12, 2017.
[14] H. Sharma, *et al.*, "Bit Fusion: Bit-Level Dynamically Composable Architecture for Accelerating Deep Neural Network," in *Proc. ISCA*, pp. 764–775, 2018.
[15] Y. Chen, *et al.*, "Eyeriss: A Spatial Architecture for Energy-efficient Dataflow for Convolutional Neural Networks," in *Proc. ISCA*, pp. 367–379, 2016.
[16] Y. Ma, *et al.*, "Optimizing the Convolution Operation to Accelerate Deep Neural Networks on FPGA," *IEEE T. VLSI Syst*, vol. 26, pp. 1354–1367, Jul. 2018.
[17] "CACTI." available at http://www.hpl.hp.com/research/cacti/.
[18] M. Horowitz, "Computing's Energy Problem (and What We Can Do About It)," in *Proc. ISSCC*, pp. 10–14, 2014.