CNNFlow: Memory-driven Data Flow Optimization for Convolutional Neural Networks

QI NIE and SHARAD MALIK, Princeton University

Convolution Neural Networks (CNNs) are widely deployed in computer vision applications. The datasets are large, and the data reuse across different parts is heavily interleaved. Given that memory access (SRAM and especially DRAM) is more expensive in both performance and energy than computation, maximizing data reuse to reduce data movement across the memory hierarchy is critical to improving execution efficiency. This is even more important for the common use case of CNNs on mobile devices where computing/memory resources are limited. We propose CNNFlow, a memory-driven dataflow optimization framework to automatically schedule CNN computation on a given CNN architecture to maximize data reuse at each level of the memory hierarchy. We provide a mathematical calculation for data reuses in terms of parameters including loop ordering, blocking, and memory-bank allocation for tensors in CNN. We then present a series of techniques that help prune the large search space and reduce the cost of the exploration. This provides, for the first time, an *exact* and *practical* search algorithm for optimal solutions to minimize memory access cost for CNN. The efficacy is demonstrated for two widely used CNN algorithms: AlexNet and VGG16 with 5 and 13 convolution layers, respectively. CNNFlow finds the optimal solution for each layer within tens of minutes of compute time. Its solution requires about 20% fewer DRAM accesses and 40%–80% fewer SRAM accesses compared to state-of-the-art algorithms in the literature.

CCS Concepts: • **Hardware → Hardware-software codesign;**

Additional Key Words and Phrases: Accelerator, Convolutional Neural Network, software-hardware co-design, memory utilization, data scheduling

ACM Reference format:
Qi Nie and Sharad Malik. 2023. CNNFlow: Memory-driven Data Flow Optimization for Convolutional Neural Networks. *ACM Trans. Des. Autom. Electron. Syst.* 28, 3, Article 40 (March 2023), 36 pages.
https://doi.org/10.1145/3577017

1 INTRODUCTION

Convolutional neural networks (CNN) are the main class of algorithms used for vision tasks such as image classification, object detection, and localization. They are increasingly used in mobile devices where the computing/memory resources are limited. Further, mobile devices require high performance but are power-constrained, which drives the need to improve the energy efficiency of executing CNNs. The datasets of CNN are large, and their memory access time (SRAM and especially DRAM) dominate performance and energy efficiency in comparison with computation

Authors' addresses: Q. Nie, Department of Electrical and Computer Engineering, Princeton University, Princeton, NJ 08544; email: qnie@princeton.edu; S. Malik, Department of Electrical and Computer Engineering, Princeton University, Princeton, NJ 08544; email: sharad@princeton.edu.

1084-4309/2023/03-ART40 $15.00
https://doi.org/10.1145/3577017

[1]. Therefore, to improve the energy efficiency, we need to reduce the number of data movements across the memory hierarchy during the CNN computation.

Many novel processors/accelerators have been proposed to execute CNN with reduced data movement. Some consider increasing the local memory size [2]. Others utilize more efficient and lighter-weight machine learning algorithms [3, 4]. In-memory computing also helps with eliminating the data movement between the storage and compute units [5]. However, for a given algorithm and hardware resources, the number of data movements is controlled by the degree of data reuse in each level of the memory hierarchy, including datapath registers in the processors/accelerators, program-managed on-chip SRAM, and DRAM. This is further controlled by how the computation is scheduled. The schedule determines when and where each operation happens in the architecture. We need to orchestrate a dataflow to maximize the data reuse in the lowest levels of the memory and thus better utilize the local memory for storing the intermediate results of the computation.

The programmer/compiler is responsible for managing the dataflow across the memory hierarchy [6–9]. This is currently done using heuristic algorithms. These algorithms exploit several degrees of freedom to maximize data reuse—loop ordering in the computation, blocking data, and allocation of memory banks to specific tensors. However, it is done heuristically without any guarantee of optimality as the search space, for these degrees of freedom is large. In this work, we improve on this by providing an exact solution to a precise problem formulation that minimizes memory access cost. Specifically, this work presents CNNFlow, a dataflow optimization framework for optimally mapping CNN to accelerators that have a spatial array of processing elements and a global on-chip buffer (SRAM). In doing so, this work makes the following contributions:

- It defines an optimization problem for minimizing memory access cost for a given CNN kernel and a target architecture.
- It provides mathematical models to efficiently evaluate the memory access cost for each point in the search space.
- It proposes a set of techniques to prune the search space and find the optimal solution efficiently.
- It demonstrates its efficacy for two important CNN algorithms, AlexNet and VGG16, each with multiple kernels. CNNFlow finds the optimal solution within tens of minutes of compute time. This solution requires about 20% fewer DRAM accesses and 40%–80% fewer SRAM accesses compared to state-of-the-art algorithms in the literature.

This article is organized as follows: We start by reviewing related work in Section 2. This is followed by the overview of the CNNFlow Framework in Section 3. In this section, we cover the architecture and application templates used in our problem formulation, a discussion of the complete parameter space, and the analytical formulation of the optimization problem for optimizing accesses across the memory hierarchy. Following this, Section 4 provides an analytical calculation of the number of DRAM accesses for a given point in the parameter space. This replaces the need to perform a simulation of the application on the architecture for each point in the parameter space. Next, Section 5 discusses how the large parameter space can be pruned using a set of algorithms for the DRAM accesses. This is extended to include SRAM accesses in Section 6. Section 7 provides the results of the experimental evaluation of these algorithms. Finally, Section 8 provides some concluding remarks.

2 RELATED WORK

Accelerator Design: There are many accelerator designs targeting CNNs. They customize function units, datapath interconnect, memory organization, and controllers for different CNN kernels. Work in References [4, 10–15] adopt a parallel array of **processing elements (PEs)**. The

specialization of PE-based architecture is largely reflected in the interconnect of PEs. Besides the commonly used systolic array where PEs are connected by a 2D mesh [15], some designs only use 1D interconnect [11, 13] but others may allow PEs to communicate with their diagonal neighbors [16]. Designs generally have software-controlled SRAM as on-chip local memory but the SRAM partitioning is fixed for all computation primitives being executed [4, 12, 15, 17, 18]. However, as Chen and Anderson [19] point out, customizing the local memory partitioning based on the algorithm could improve the utilization of memory bandwidth.

Besides the hardware, accelerators also make decisions about the computation scheduling. In terms of partitioning the CNN computation to reuse the accelerator, most designs adopt the layer-by-layer computation in that the execution of one layer will not start until the completion of previous layer [14, 17], while some designs allow multiple layers being simultaneously processed on the chip [20–22]. Within each layer, scheduling needs to allocate the parallelism across different data arrays. Some designs only consider parallelism within convolution kernels [23, 24], while other designs also consider parallelism in computing multiple output images [25]. The design in Reference [26] considers both types of parallelism. With the 2D PE array, data from two tensors move horizontally or vertically across PEs, and those from the other tensor will stay within each PE. Some designs keep the weights data stationary in PEs [18, 23, 25, 27], while other designs make the output data stationary [12, 15, 28, 29]. The FlexFlow approach [30] presents both an architecture as well are corresponding dataflow that provides additional flexibility based on certain compute patterns. The compute patterns consider different forms of parallelism, and the dataflow optimizes transfers from the on-chip buffers to the PEs. In contrast, CNNFlow provides an analytical formulation and mathematical optimization for maximizing reuse that is not based on a specific set of compute patterns and considers multiple levels of memory hierarchy across DRAM, SRAM, and PE Register Files.

Automatic Scheduling of Computation: The optimum choice of computation scheduling depends on the algorithm and available hardware resources, as pointed out in MemFlow [31, 32]. Handcrafting the design for each case is very time-consuming. Thus, there are many previous works investigating automation and optimization for hardware generation and computation scheduling [11, 14, 28, 33–36]. **High-level Synthesis (HLS)** automates the hardware generation through providing a high-level behavioral input for hardware designers. However, HLS focuses on designing the datapath for local computation where data are already in **Block RAM (BRAM)**. Advanced HLS tools such as SDAccel [37] provide a framework to optimize data movement across the memory hierarchy. However, it only offers support for emulation and profiling, but requires manual effort to select the design parameters impacting dataflow across the memory hierarchy. Zhang et al. [38, 39] and Motamedi et al. [26] utilize a roofline model to identify the optimal design. DeepBurning [40] takes neural network descriptions and generates the hardware and the control flow to meet the given area/power constraints. DNNBuilder [20] maps neural networks to FPGAs in a layered-pipeline implementation and balances the allocation of compute and memory resources across layers. These designs do not explicitly optimize the mapping to improve the local memory utilization. Work in Reference [15] sets loops being mapped to the PE array, the PE array dimension, and blocking size as configurable parameters. AngelEye [41] optimizes block partitioning and memory mapping in its compiler. Work in Reference [38] and Caffine [39] has one-level blocking, and it selects tile size based on computation-to-communication ratio. Eyeriss [1] configures a set of mapping parameters that are essentially blocking size. These works only explore a limited design space and miss other parameters that could affect the dataflow across the memory hierarchy. Work in Reference [29] does consider the larger design space with loop unrolling, loop blocking, and loop interchange. But it assumes that only one data block from each tensor stays in on-chip SRAM. Further, it randomly samples the design space to select the best combination

of parameters. Timeloop [42] tunes block dimensions and loop permutation, but its optimization is also based on random sampling. TENET [43] is a modeling framework using a novel relation-centric approach that provides for analytical computation for various "volume" metrics relating to accesses of the tensor across the **processing elements (PEs)** in a spatial architecture including their reuse across the PEs and the minimum data size that needs to be transferred between PEs and scratchpad (SRAM). As an application of this framework, the article presents a simple pruning strategy for considering the dataflow design space exploration and "leave a more efficient design space exploration as future work." The article does not consider DRAM accesses or a complete optimization of accesses across the memory hierarchy. The SmartShuttle approach [44] does provide an analysis based algorithm that selects the tile size and one of three scheduling schemes for each layer in the CNN. However, this is done using empirical rules rather than mathematical algorithms to minimize DRAM accesses. Further, it does not consider minimizing SRAM accesses and does not have a comprehensive treatment of all the parameters considered in this article. The CoSA framework [45] is similar to CNNFLow in that it uses mathematical optimization (MILP) for the scheduling. However, it assumes each tensor is at a specific level in the memory and does not consider architectures that move data across the memory hierarchy as with CNNFlow. The article provides as example of this: "dedicated input and weight buffers for input activation and weight, respectively, while providing a shared global buffer to store input and output activations." The objective functions considered in CoSA are "maximizing buffer utilization or achieving better parallelism." This is in constrast to CNNFlow's optimizing accesses across the memory hierarchy. GAMMA [46] provides a genetic algorithm for mapping layers onto an accelerator such that the mapping fits in the available resources, and with a focus similar to CoSA on maximizing parallelism. It considers on-chip and PE buffers, but not DRAM. Again, it is unclear how this can be extended to a precise formulation of optimizing accesses across the memory hierarchy. In contrast, CNN-Flow considers the complete optimization space, precisely formulates an optimization problem for minimizing memory access cost for this space, and provides an exact solution for this problem.

3 CNNFLOW FRAMEWORK

Layer-by-layer computation. In CNNFlow, the neural network is executed layer-by-layer. The computation of one layer generally does not start until the completion of its preceding layer. While it is possible to start the next layer before the preceeding layer is complete, there are good reasons for this layer-by-layer ordering. After computing a block of results in layer 1, starting the execution of layer 2 could reuse intermediate results just generated from layer 1. However, inputs to layer 1 are still alive and could be further reused to generate other results of layer 1. Proceeding with layer 2 needs to bring other new input data to the local memory, which may potentially evict live data from layer 1. Given that local memory is limited and unable to accommodate the working-set data of a single layer, working with more than one layer at a time does not have an obvious advantage in maximizing data reuse for all tensors. Thus, we assume that one layer finishes before the next starts. This is consistent with almost all implementations of CNNs.

3.1 Application Template

Each CNN layer implements operations on tensors (multidimensional arrays), drawn from a limited set of operations such as convolution, matrix multiplication, pooling, and local response normalization, as shown in Figure 1. Each layer can be captured by an application template that includes a nested loop, termed as **algorithm loops**, and a macro operation (**MacroOp**) in each iteration of the nested loop. These layers share two common characteristics: (1) algorithm loops are deep; (2) the MacroOp is simple. The algorithm loops are independent of each other and are interchangeable. This provides a high degree of parallelism and massive data reuse in MacroOp

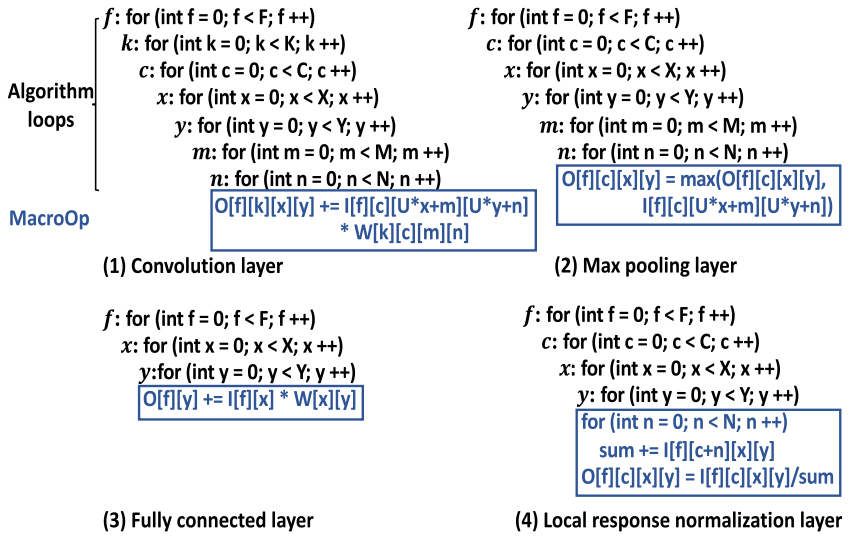


Fig. 1. CNN layers displaying the application template.

execution. Each MacroOp computes one scalar result, and it may or may not have loops in its definition (in Figure 1, (1)–(3) do not, (4) does). Loops within a MacroOp are not interchangeable with the algorithm loops. The input parameters to CNNFlow from the application level are: (1) the type of MacroOp, (2) the algorithm loops represented by the set of loop indicators $\mathcal{I}_{in} = i_1, \ldots, i_m$, and (3) their input dimensions $\mathcal{B}^d = B^d_{l_1}, \ldots, B^d_{l_m}$.

3.2 Architecture Template

The accelerator architecture we target consists of a uniform software-controlled SRAM scratchpad memory and a spatial array of **processing elements (PE)**, as shown in Figure 2. Each PE has **function units (FU)** for computation and a small software-controlled register file scratchpad. The accelerator communicates with DRAM through DMA, and data can move between DRAM and SRAM. SRAM can talk to each PE through the row and column buses in the PE array. Each PE can talk to its neighbors in all four directions. SRAM has a very limited number of ports in each bank. To avoid port conflict among tensors, one SRAM bank is only assigned to store one tensor. Further, to simplify memory control, SRAM and the register file in the PE are partitioned such that each tensor has a fixed region to hold its own data in one kernel execution. This memory partitioning is determined at the program level.

This architecture can help with the high degree of parallelism and data reuse in our targeted applications and is commonly used by other CNN accelerators [1, 47, 48]. The accelerator template in CNNFlow is configured by the following input parameters: (1) the number of SRAM banks N^{sb}_{in}, (2) the bank size M^{sb}, (3) the register file size in each PE M^r_{in}, and (4) the PE array dimension $D^{pe}_1 \times D^{pe}_2$. Note that the impact of memory organization and access times is captured through (i) the SRAM bank parameters being an input to the overall optimization and (ii) the access time ratio between DRAM and SRAM indicated by the input parameter w, as shown in Equation (5) below.

3.3 Computation Mapping

The input data of the application layer are initially stored in DRAM and need to be moved to the FUs in the PEs for computation. Executing the applications in Figure 1 on the architecture

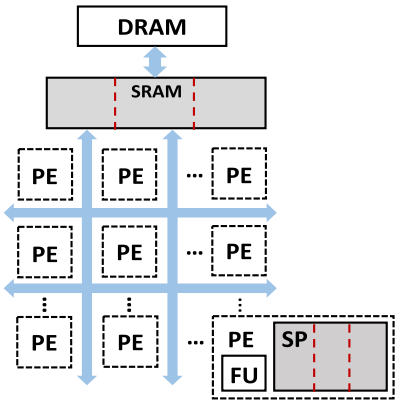


Fig. 2. Architecture template.

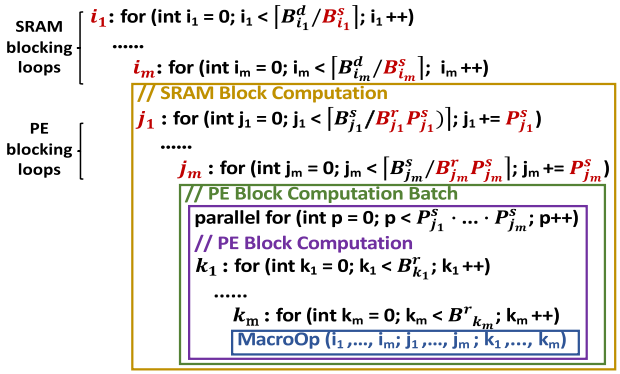


Fig. 3. Computation mapping.

in Figure 2 is about scheduling the data movement across the memory hierarchy. The execution essentially specifies the data and the path of data moving through the memory and PEs. The goal of computation mapping is to reduce the overall cost of data movement needed for computation. We consider scheduling data movements between DRAM and SRAM, between SRAM and register files, and between register files in the PEs. Figure 3 shows the hierarchical computation mapping, where the design parameters we optimize are marked in red.

The input data are first partitioned into SRAM-level blocks such that each block can fit in SRAM. The block dimension variables are given as $\mathcal{B}^s = B^s_{i_1}, \ldots, B^s_{i_m}$, where B^s_i is the block dimension for loop $i \in \mathcal{I}_{in}$. The data blocking increases local data reuse in SRAM. Each block dimension controls the maximum degree of data reuse one tensor could have. Through adjusting block dimensions, we can balance the data reuse between tensors. To avoid SRAM bank conflict, we allocate one complete SRAM bank to one tensor. We can control the utilization of SRAM size and bandwidth through adjusting how SRAM banks as well as its ports are partitioned among tensors. The design variables $\mathcal{N}^{sb} = \{N^{sb}_t\}_{t \in T}$ are used to represent the SRAM partitioning, where tensor t from the operand set T has N^{sb}_t SRAM banks. Given N^{sb}_{in} as the total number of SRAM banks, the sum of N^{sb}_t equals to N^{sb}_{in}, $\sum_{t \in T} N^{sb}_t = N^{sb}_{in}$. We define the SRAM Block Computation as the computation that takes one SRAM-level data block from each operand tensor and produces and stores the resulting data block in SRAM. The execution of the SRAM Block Computation will use up all the SRAM ports and PE resources; thus, they are scheduled one after the other in sequence. Their execution ordering is determined by the ordering of SRAM blocking loops, as shown in Figure 3. We use ordered list $\boldsymbol{I}^d = [i_{o1}, \ldots, i_{om}]$ to indicate the loop ordering, where $i_{ol} \in \mathcal{I}_{in}$ is the loop indicator for the lth SRAM blocking loop.

The second level of blocking is to further partition the SRAM-level data block into PE-level data blocks with block dimensions given by $\mathcal{B}^r = B^r_{j_1}, \ldots, B^r_{j_m}$. The PE Block Computation operates on PE-level data blocks and is executed inside one PE. Due to the limited number of PEs in the datapath, the PE Block Computations of one SRAM Block Computation are mapped to the PE array in batches. Each batch executes $D^{pe}_1 \times D^{pe}_2$ PE Block Computations in parallel, and the ordering of batch execution is determined by the ordering of PE blocking loops as marked in Figure 3. The ordered list $\boldsymbol{I}^s = [j_{o1}, \ldots, j_{om}]$, $j_{ol} \in \mathcal{I}_{in}$, sets loop j_l as the lth PE blocking loop.

We use variables $\mathcal{P}^s = P^s_{j_1}, \ldots, P^s_{j_m}$ to represent the PE batch dimension. They are given as the step size of PE blocking loops in Figure 3. The product of P^s_j for all loops should equal to the number of physical PEs available, $\prod_{j \in \mathcal{I}_{in}} P^s_j = D^{pe}_1 \times D^{pe}_2$. To simplify the mapping, we let each

batch dimension be completely mapped to either D_1^{pe} or D_2^{pe}. Thus, the product of P_j^s for loops in a subset $\mathcal{I}_1 \subset \mathcal{I}_{in}$ is equal to D_1^{pe}, $\prod_{j\in\mathcal{I}_1} P_j^s = D_1^{pe}$, and the product of others is equal to D_2^{pe}, $\prod_{j\in\mathcal{I}_{in}-\mathcal{I}_1} P_j^s = D_2^{pe}$. This mapping rule captures the common inter-PE dataflow strategies used by other designs. In the systolic-array-based designs, data of one tensor are moved horizontally and reused across PEs in the same row while data in another tensor are moved vertically and reused across same-column PEs. For a fully connected layer, data in tensors I, W, and O are reused with the batch dimension of P_y^s, P_x^s, and P_f^s, respectively. We could realize the systolic array by setting one batch dimension from P_y^s, P_x^s, and P_f^s equal to D_1^{pe}, one equal to D_2^{pe}, and the other equal to 1. For example, $P_f^s = 1$, $P_x^s = D_1^{pe}$, $P_y^s = D_2^{pe}$ corresponds to the systolic-array dataflow where partial sums are reused locally inside the PEs and two operand tensors are reused across PEs. For convolution layer, the Weight Stationary [10] scheduling keeps filter weights to stay stationary inside the PEs and input and output tensors move across PEs. This is represented as $P_m^s = D_1^{pe}$, $P_n^s = D_2^{pe}$, $P_k^s = P_f^s = P_x^s = P_y^s = P_c^s = 1$ in our mapping framework. Similarly, Output Stationary [10] makes output elements stay in PEs while moving input tensor and filter weights across PEs, which corresponds to $P_f^s = D_1^{pe}$, $P_k^s = D_2^{pe}$, $P_m^s = P_n^s = P_x^s = P_y^s = P_c^s = 1$ in our framework.

3.4 Problem Statement

DRAM accesses have dominant cost in terms of time and energy consumption over SRAM accesses and computation, and thus, the main optimization goal is to reduce the number of DRAM accesses. DRAM accesses include compulsory ones and the additional ones caused by data eviction from SRAM. Compulsory DRAM accesses consist of the initial read for input tensors and write-back of the output tensor. Its total number is equal to the sum of tensor sizes, which is independent of dataflow. However, each tensor requires additional DRAM accesses when subsequent data evict their previous copies in local SRAM memory. It is these additional DRAM accesses that we optimize through the dataflow. The additional DRAM accesses of input tensors equals to the number of data spilled during computation. However, for data in output tensors, the eviction requires first writing back the current value and then re-fetching it from DRAM at its next use. Each data spill for an output tensor costs one DRAM write and one DRAM read. Equation (1) summarizes the components of the additional DRAM accesses. T_{in} and T_{out} are the sets of input and output tensors, respectively. A_t^d and E_t^d are the number of additional DRAM accesses and the number of data spilled to DRAM from tensor t. The set of parameters affecting A^d is $\mathcal{X}^s$, which includes: (1) SRAM-level block dimensions $\mathcal{B}^s$; (2) the ordering of loops $\boldsymbol{I}^d$; (3) $\mathcal{N}^{sb} = \{N_t^{sb}\}_{t\in T_{in}\cup T_{out}}$ describing how SRAM is partitioned so tensor t has N_t^{sb} SRAM banks.

$$A^d(\mathcal{X}^s) = \sum_{t\in T_{in}\cup T_{out}} A_t^d = \sum_{t\in T_{in}} E_t^d + \sum_{t\in T_{out}} 2\cdot E_t^d \tag{1}$$

The secondary target for optimization is the number of SRAM accesses A^s. SRAM accesses include compulsory accesses A^{sc} and additional accesses due to data spilling from register files A^{sa}. The compulsory accesses in each SRAM Block Computation are composed of SRAM reads for initially moving input data from SRAM to the datapath registers and SRAM writes for moving back results. The number of compulsory accesses of each SRAM Block Computation is equal to the sum of the operand data block size and the result data block size. Equation (2) shows the calculation of A^{sc}, where $N^{s,db}$ represents the number of SRAM Block Computations and C_t^s gives SRAM data block of tensor t. Therefore, C_t^s, and thus A^{sc}, is determined by SRAM-level blocking dimensions $\mathcal{B}^s$.

$$A^{sc}(\mathcal{X}^s) = N^{s,db} \sum_{t\in T_{in}\cup T_{out}} C_t^s \tag{2}$$

The additional SRAM accesses are caused by data eviction in register files. Similarly, each eviction of data in output tensors requires one SRAM read and one SRAM write. Equation (3) gives the total number of additional SRAM accesses A^{sa}. $N^{s,db}$ represents the number of SRAM Block Computations and E^s_t is the number of data spilled from registers to SRAM in one SRAM Block Computation of tensor t. Besides $\mathcal{X}^s$, its other parameters $\mathcal{X}^r$ include: (1) PE-level block dimensions $\mathcal{B}^r$; (2) PE Block Computation batch dimensions $\mathcal{P}^s$; (3) the loop ordering $\boldsymbol{I}^s$; (4) $\mathcal{M}^r = \{M^r_t\}_{t \in T_{in} \cup T_{out}}$ describing how each register file is partitioned so tensor t has M^r_t space for its own data.

$$A^{sa}(\mathcal{X}^s, \mathcal{X}^r) = N^{s,db}\left(\sum_{t \in T_{in}} E^s_t + \sum_{t \in T_{out}} 2 \cdot E^s_t\right) \tag{3}$$

Equation (4) gives the objective function for minimizing the number of SRAM accesses A^s, which includes both compulsory SRAM accesses and additional SRAM accesses due to data spilling from register files.

$$A^s(\mathcal{X}^s, \mathcal{X}^r) = A^{sc}(\mathcal{X}^s) + A^{sa}(\mathcal{X}^s, \mathcal{X}^r) \tag{4}$$

To optimize the overall local memory utilization, we aggregate the number of DRAM accesses and SRAM accesses into one optimization objective by calculating their weighted total, as **Problem 0** defined in Equation (5). (The weight w can be determined based on the specific hardware design/technology used.)

$$\begin{aligned} &\textbf{Problem 0:} \quad \underset{\mathcal{X}^s, \mathcal{X}^r}{\text{minimize}} \quad A^d(\mathcal{X}^s) + w \cdot A^s(\mathcal{X}^s, \mathcal{X}^r) \\ &\text{subject to } 1 \le B^r_l \le B^s_l \le B^d_l, \sum_t N^{sb}_t = N^{sb}_{in}, \sum_t M^r_t = M^r \end{aligned} \tag{5}$$

The straightforward method for solving **Problem 0** is sweeping the entire parameter space of $\mathcal{X}_s \cup \mathcal{X}_r$. For each point in this space, we can simulate the memory system during execution and count the total memory accesses at each level. However, this method is not practical for real cases, given the enormous size of the design space. Consider the first convolution layer in AlexNet with input dimension for $f, c, k, x, y, m, n, stride$ as $1, 3, 96, 227, 227, 11, 11, 4$ and a typical setting of memory including 27 4 KB SRAM banks, 260 registers in one PE, and a 12×14 PE array. (We refer to this instance as **AN1**, short for AlexNet layer 1, in the rest of the article.) The size of the design space for **AN1** is about 10^{30}. Sweeping a design space with this size is impractical within a feasible amount of compute time. Further, at each of these design points, we will need an expensive memory system simulation to count the number of memory accesses at each level. In the following sections, we will introduce approaches for addressing both aspects of the search cost: (1) the cost of memory system simulation to evaluate the objective function value for a given design point and (2) the large number of design points.

For ease of explanation, we first look at a simpler version of **Problem 0**, referred to as **Problem 1** that is restricted to optimizing DRAM accesses. As shown in Equation (6), **Problem 1** only optimizes the number of additional DRAM accesses $A^d(\mathcal{X}_s)$. However, solving **Problem 1** involves all the techniques for solving **Problem 0** and thus, we start with this to ease explanation. For the application and architecture example mentioned before, the size of $\mathcal{X}_s$ is about 1.85×10^{14} for **Problem 1**. Figure 4 gives an overview of our solution to **Problem 1** and where each part of the solution is covered in the next two sections.

$$\begin{aligned} &\textbf{Problem 1:} \quad \underset{\mathcal{X}^s}{\text{minimize}} \quad A^d(\mathcal{X}^s) \\ &\text{subject to} \quad 1 < B^s_i < B^d_i, \sum_t N^{sb}_t = N^{sb}_{in} \end{aligned} \tag{6}$$

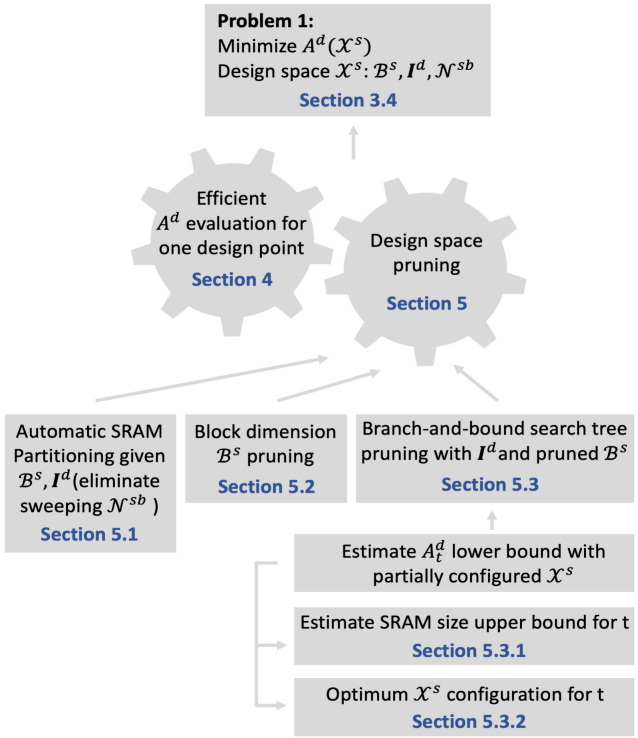


Fig. 4. An overview of the solution to Problem 1.

4 ANALYTICAL COMPUTATION OF DRAM ACCESSES

As shown in Figure 4, in this section, we provide an efficient analytical calculation for the number of additional DRAM accesses A^d for a given set of design parameters $\mathcal{X}^s$ to avoid simulating the memory system. Between DRAM and SRAM, data are moved to provide operand data blocks for SRAM Block Computation. When SRAM is fully occupied by data that still have a future use, a new block used by SRAM Block Computation needs to evict existing data in SRAM to be allocated. Based on the optimum spilling strategy [49], the optimum data to evict are those with furthest next reuse. This result is key to our analytical calculation of memory accesses. For CNN-layer computations, the data use pattern is regular and periodic due to the loop structure, for which we can efficiently determine the optimum dataset to evict in each step.

To better understand this, we will first explain how loops affect the data reuse pattern. From the view of tensor t, loops can be classified into three categories as follows:

(1) **Reuse loop:** The loop does not provide any coordinate to the tensor's element. The data blocks get reused when the loop iterates. For instance, in convolution, loops $\{k\}$, $\{f, x, y\}$, $\{c, m, n\}$ are reuse loops for tensors I, W, O, respectively.
(2) **Related loop:** The loop independently provides a coordinate to the tensor's element. While the loop is iterating, different data blocks get used. For convolution layer, loops $\{f, c\}$, $\{k, c, m, n\}$, $\{f, k, x, y\}$ are related loops for tensors I, W, O, respectively.
(3) **Partial reuse loop:** The loop determines a coordinate of the tensor's element together with other loops. While the loop is iterating, the next data block in use will have partial overlap with the current one. For convolution, loops $\{x, y, m, n\}$ are partial reuse loops for tensor I and tensors W, O do not have partial reuse loops.

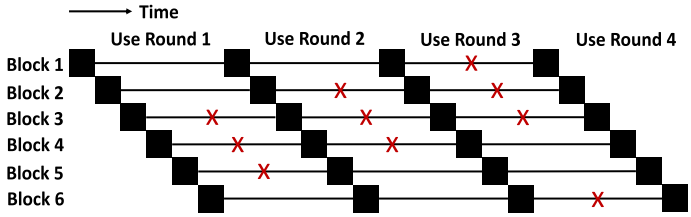


Fig. 5. An example of reuse group (red cross represents data being evicted.)

The loop types and their relative ordering together define the pattern of data reuse in tensor t. In adopting the optimum data spilling strategy, the loop configuration further decides the pattern of data spilling. We explain this by considering different forms of data reuse pattern.

Complete Reuse Group

Complete reuse group is defined as a group of data blocks being reused alternately. This reuse pattern is formed by having one reuse loop followed by a few related loops. Figure 5 shows an example of six data blocks that are alternately reused for four times. We term one round of data block usage as a **use round**. In this example, there are four use rounds, and data blocks 1–6 get used in order in each use round.

To avoid data spilling, SRAM needs to be able to hold all six blocks. Otherwise, data blocks will evict each other. Belady's replacement policy [49] states that the optimal replacement policy is one that evicts the block that will be used furthest in the future. This policy can be used in our context, since the block access pattern is known statically. Assume SRAM can hold up to three data blocks. In the initial use round, the memory is full when we allocate the fourth data block, and the data block we choose to evict is block 3, as its next use is later than that for block 1 and 2. In Figure 5, we use a red cross to represent a data block being evicted. Similarly, allocation of block 5 will evict block 4, and allocation of block 6 will evict block 5, as they both have later next use compared to block 1 and 2. In the following rounds of data use, the data block gets reused in SRAM if SRAM holds its previous copy; otherwise, it will evict the data block being used most recently. The data block eviction forms a periodic pattern that enables the total amount of spilling to be analytically determined. We term one iteration of data block reuse and spilling in the cyclic pattern as a **spilling cycle**. Let the data block size be S^b, the number of data blocks in the group be N^b, the reuse times be R. We only need to consider the spilling between reuse rounds 1 to $R-1$, as data blocks in the last round do not have future use. Let $S^{b,r}$ be the size of the data block being reused. For complete reuse group, $S^{b,r} = S^b$ but we will see other cases in the following. Let D be the size difference between the use round and the spilling cycle. For complete reuse group, $D = S^b$. Let M^{eff} be the effective memory size used to store data with future reuse. For complete reuse group, all data in each use round has the next reuse, thus M^{eff} is equal to the memory size allocated to the tensor. Equation (7) captures the number of spills in the complete reuse group. SP^c, N^c, S^{last_c}, SP^{last_c} represent the number of spills in each spilling cycle, the number of spilling cycles, the length of the last residual spilling cycle, and its number of spills, respectively.

$$
\begin{aligned}
&E(S^{b,r}, N^b, D, R, M^{eff}) = SP^c \cdot N^c + SP^{last\_c} \\
&SP^c = max(0, N^b \cdot S^{b,r} - M^{eff})) \\
&N^c = (R-1) \cdot N^b \cdot S^{b,r} / (N^b \cdot S^{b,r} - D)) \\
&SP^{last\_c} = max(0, S^{last\_c} - (M^{eff} - D)) \\
&S^{last\_c} = (R-1) \cdot N^b \cdot S^{b,r} \% (N^b \cdot S^{b,r} - D)
\end{aligned}
\tag{7}
$$

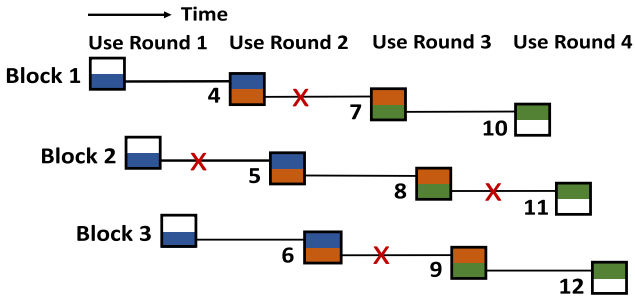


Fig. 6. An example of partial reuse group (Red cross represents data eviction.)

We define **spill-free memory size** as the minimum SRAM size required to eliminate the data spilling in a reuse group, denoted as M^{sf}. For complete reuse group, M^{sf} is equal to $S^b \cdot N^b$.

Partial Reuse Group

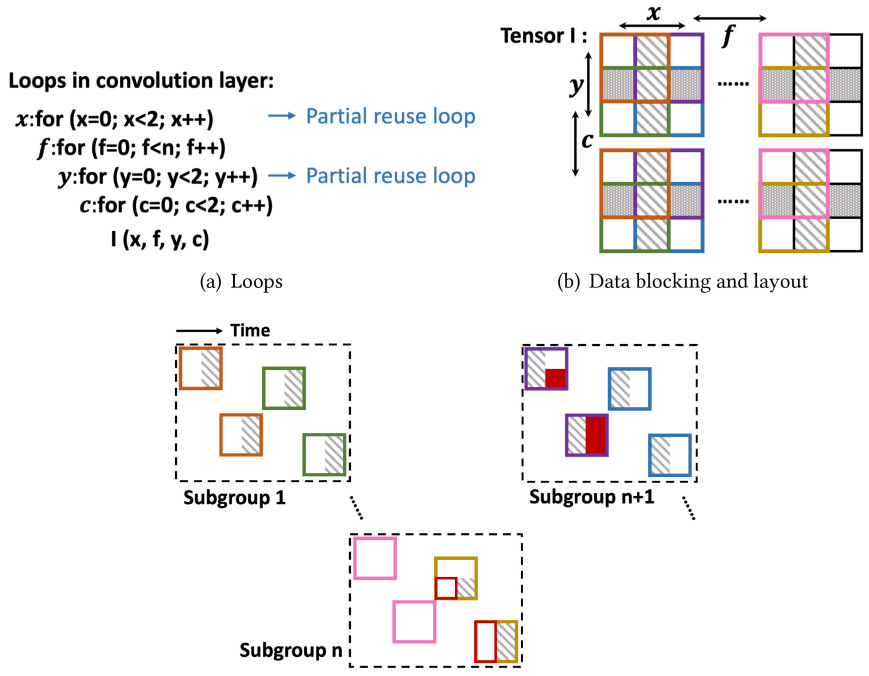
A partial reuse loop followed by a few related loops will define rounds of data block use where each data block only partially overlaps with the one in the previous round. Figure 6 shows an instance of partial reuse group where blocks 4, 5, 6 in the second use round partially overlap with blocks 1, 2, 3. Unlike the complete reuse group case where the exact same set of data are being reused in each round, in the partial reuse group case, the data instances being reused in each round are different. However, the size of the dataset being reused across rounds stays the same. After using each data block, we only need to keep in SRAM the part of data with use in the next round and can erase other data without future use.

We call the portion of a data block that has next reuse as a block's **reuse section** and the other part as its **non-reuse section**. As with the complete reuse group, we use S^b and N^b to represent the data block size and the number of data blocks in the group. $S^{b,r}$ is the size of the block's reuse section. Thus, for partial reuse group, $S^{b,r} < S^b$. To avoid data spilling, SRAM at least needs to hold: (1) all data being reused across use rounds, of which the size is $N^b \cdot S^{b,r}$; (2) the non-reuse section of the current data block, with size $S^b - S^{b,r}$. Therefore, the spill-free memory size M^{sf} for a partial reuse group is equal to $S^{b,r} \cdot (N^b - 1) + S^b$.

When SRAM is smaller than M^{sf}, data spilling occurs. In the example of Figure 6, assume SRAM has size $S^b + S^{b,r}$ and it is able to hold one full data block and one block's reuse section. In the first round, after using block 1 and 2, their reuse sections will stay in SRAM. But the execution of block 3 needs SRAM space to hold the full data block. Thus, the partial block 2 in SRAM will be evicted, as its subsequent reuse is later than block 1's. The eviction is also marked as a red cross in the figure. SRAM space of size $S^b - S^{b,r}$ is allocated for the non-reuse section of the current data block. The rest of SRAM space is for storing data being reused across use rounds. Thus, the effective memory size for resolving data spilling M^{eff} in partial reuse group is equal to $M - (S^b - S^{b,r})$. As there are two blocks' reuse sections that can stay in SRAM, the blocks' reuse section will evict each other. Following the optimal replacement rule, the optimum reuse section to evict is still the one being used most recently. Therefore, the data spilling displays a periodic pattern as with the complete reuse group case except now $S^{b,r} \neq S^b$ and $M^{eff} \neq M$. Accordingly, the size difference between use round and spilling cycle D is equal to $S^{b,r}$. Equation (7) captures the number of data evictions.

Embedded Reuse Group

For deeply nested loops, the reuse, partial reuse, and related loops may appear alternately, resulting in a embedded structure of reuse groups. Instead of a data block as a reuse unit, such as in Figures 5 and 6, a reuse group can be further reused by its higher-level loops.



(a) Loops (b) Data blocking and layout

(c) Block access pattern

Fig. 7. Embedded reuse group.

Figure 7 gives an example of a two-level embedded reuse group. Figure 7(a) shows four innermost loops of a convolution layer as c, y, f, x. Figure 7(b) gives the data layout of tensor I along four dimensions. Figure 7(c) shows the data block reuse pattern when four loops iterate. Loops y and x are I's partial reuse loops. Loops c and f are I's related loops. Therefore, the execution of loops y and c forms a partial reuse group. Different partial reuse groups get invoked as loop f iterates. Then, loop x's next iteration will invoke a new set of partial reuse groups, but each one will partially overlap with the previous one. We name the inner group as the **subgroup** of the outer one. For each subgroup on the outer use rounds 1 to $R-1$, data that are not reused inside the subgroup will be reused in the next outer use round. The data spilling now includes the spilling inside each subgroup and the spilling of the outer reuse group.

We still call the portion of the subgroup that has subsequent reuse across outer rounds as its reuse section and the other portion as its non-reuse section. $S^{b,r}$ and N^b represent the size of the subgroup's reuse section and the number of subgroups, respectively. To eliminate data spilling, SRAM first needs to hold all the data being reused across outer rounds, with size equal to $N^b \cdot S^{b,r}$. Second, SRAM needs an extra space to make each subgroup spilling free. The size of this extra space could be smaller than subgroup's spill-free memory size M^{sf}, denoted as M_b^{sf}, as the set of data being reused in the subgroup and over outer rounds could overlap. We use $\mathcal{D}^{b,sf}$ to represent the initial dataset in a subgroup that fully occupies its spill-free memory size. Thus, the size of $\mathcal{D}^{b,sf}$ is equal to $M^{b,sf}$. We use $S()$ to represent the size of a dataset, then $S(\mathcal{D}^{b,sf}) = M^{b,sf}$. We use $\mathcal{D}^{b,r}$ to represent the set of data in a subgroup that are reused across outer round. Thus, $S(\mathcal{D}^{b,r}) = S^{b,r}$. The extra SRAM space required to eliminate spilling in the subgroup is equal to $S(\mathcal{D}^{b,sf} - \mathcal{D}^{b,r})$. Therefore, the spill-free memory size M^{sf} of the outer reuse group is equal to $N^b \cdot S^{b,r} + S(\mathcal{D}^{b,sf} - \mathcal{D}^{b,r})$.

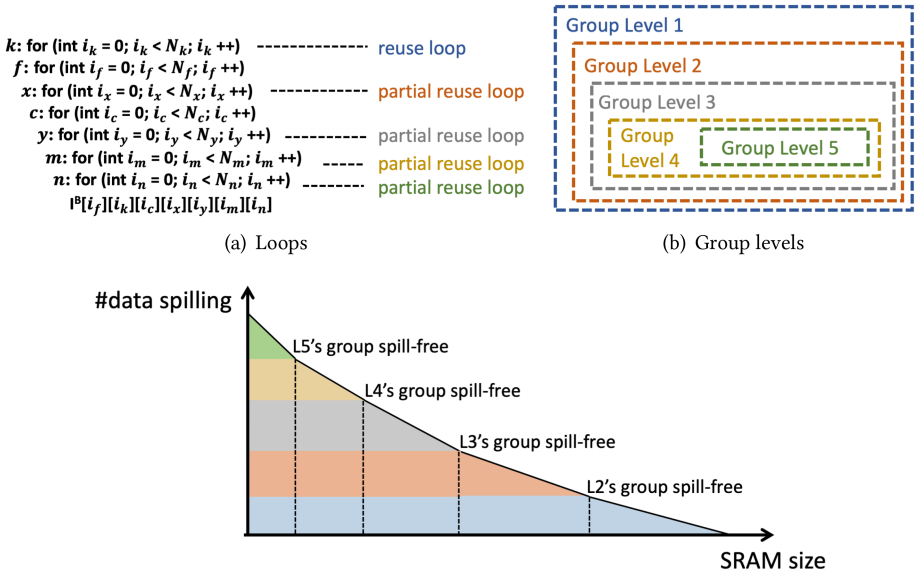
If SRAM size is smaller than M^{sf}, then data spilling happens. In the example of Figure 7, SRAM size is set to $S^{b,r} + M^{b,sf}$. Besides holding one subgroup's reuse section, SRAM can eliminate data spilling inside the subgroup. After the execution of subgroup 1, its reuse section is left in SRAM for use in the second outer round. The execution of subgroup 2 could utilize the remaining SRAM space to eliminate its own spilling. However, subgroup 2's reuse section could not stay in SRAM, as the execution of subgroup 3 will evict space occupied by subgroup 2. Following the optimum replacement strategy, data reused inside a subgroup has a higher priority in taking SRAM space over data only with reuse across outer use rounds. The eviction aims to make the subgroup spill-free first. Only if there is still extra SRAM space, the spilling of the outer group will be handled. In the first outer use round, subgroup i evicts data from subgroup $i-1$ to get the SRAM space of size $M^{b,sf}$ that makes itself spill-free. Subgroup $n+1$ in the second outer round can reuse subgroup 1's reuse section that is in SRAM. And as mentioned earlier, this reuse section overlaps with the dataset being reused inside subgroup $n+1$ by size $S(D^{b,sf} \cap D^{b,r})$. To eliminate its data spilling, subgroup $n+1$ only needs an extra SRAM space of size $S(\mathcal{D}^{b,sf} - \mathcal{D}^{b,r})$. Therefore, subgroup n can still keep in SRAM a portion of its reuse section, of which the size is $S(\mathcal{D}^{b,sf} \cap \mathcal{D}^{b,r})$. Subgroup $n+2$ and following subgroups in the second round will then evict data of size $M^{b,sf}$ from the previous subgroup to make itself spill-free. We see that for the outer group, its data spilling still forms a cyclic pattern that can be captured by Equation (7). The size difference of the use round and the spilling cycle D is equal to $S(\mathcal{D}^{b,sf} \cap \mathcal{D}^{b,r})$. The effective memory used for holding data being reused across outer rounds is equal to $M^{eff} = M - S(\mathcal{D}^{b,sf} - \mathcal{D}^{b,r})$.

When memory is smaller than $M^{b,sf}$, spilling happens inside each subgroup. Each subgroup will fully occupy the SRAM and evict all data from its preceding subgroup. No data can stay in SRAM across the use rounds of the outer group. Thus, the outer group displays full data spilling, with the number of data being spilled equal to $N^b \cdot S^{b,r} \cdot (R-1)$.

To summarize, data spilling of the embedded reuse group consists of spilling inside each subgroup and spilling of the outer group. When memory size is unable to eliminate the subgroup's spilling, all data reused across outer group's use rounds are spilled. The data spilling inside each subgroup decreases as memory size increases. Each subgroup's spilling count is identical and can be calculated with Equation (7). When memory size meets the value of $M^{b,sf}$, the subgroup becomes spill-free. If memory size further increases, then spilling generated in the outer group starts decreasing. The embedded reuse group is entirely free of spilling when SRAM size is equal to its spill-free memory size M^{sf}.

Hierarchical Reuse Groups

More generally, deeply nested loops will deliver a hierarchy of reuse groups. The subgroup may have its subgroups. The spilling numbers in the subgroup further depends on if SRAM can make its subgroup spill-free. Figure 8(a) gives an example loop structure that corresponds to five-level reuse groups, as shown in Figure 8(b). The total number of data spilling includes spilling from each level of reuse groups. Figure 8(c) plots how the total number of data spilling is reduced as the memory size increases. If the memory size is between the spill-free memory size of the group in level $i+1$ and group in level i, then the data spilling are generated from groups in level i and above while all groups of level $i+1$ and below are spill-free. In this case, the group above level i needs to spill all its data reused across its use rounds, which gives its maximum number of data spilling. A group in level i could achieve some data reuse with Equation (7) describing its relation to the memory size. Equation (8) summarizes the calculation of total spilling number A_t^d for a given memory size M_t. N_i^g represents the number of instances for the group in level i. $A_{t,i}^d$ gives the number of data spilling from one group instance in level i. Each group in level i has R_i use rounds and N_i^b level-$(i+1)$ groups as subgroups in each use round that are in level $i+1$. Each subgroup has a set of



(a) Loops (b) Group levels

(c) Data spilling vs. memory size

Fig. 8. Hierarchical reuse groups.

data $\mathcal{D}_i^{b,r}$ being reused across outer use rounds and the dataset size is $S_i^{b,r}$. As subgroup is in level $i+1$, $\mathcal{D}_{i+1}^{sf}$ gives the spill-free dataset of the subgroup.

$$A_t^d(M_t) = \sum_i N_i^g \cdot A_{t,i}^d(M_t)$$

$$A_{t,i}^d(M_t) = \begin{cases} 0, & \text{if } M_t >= M_i^{sf} \\ N_i \cdot S_i^{b,r} \cdot (R_i - 1), & \text{else if } M_t < M_{i+1}^{sf}, \\ E(S_i^{b,r}, N_i^b, S(\mathcal{D}_i^{b,sf} \cap \mathcal{D}_i^{b,r}), R_i, M_t - S(\mathcal{D}_i^{b,sf} - \mathcal{D}_i^{b,r})), & \text{otherwise} \end{cases} \tag{8}$$

Algorithm 1 gives how to evaluate the data spilling in tensor t for given loop ordering $\boldsymbol{I}^d$, block dimensions $\mathcal{B}^s$, and the SRAM size for the tensor M_t. Loop ordering $\boldsymbol{I}^d$ and block dimensions $\mathcal{B}^s$ will define a hierarchy of reuse groups. For each group level i, they specify the type of reuse group and in each group, the number of use rounds R_i, the number of subgroups in each round N_i^b, the dataset of subgroup being reused $\mathcal{D}_i^{b,r}$ and its size $S_i^{b,r}$. The other two pieces of information of group in level i used in Equation (8) are the number of group instances N_i^g and its subgroup's spill-free dataset $\mathcal{D}_i^{b,sf}$, also represented as $\mathcal{D}_{i+1}^{sf}$. N_i^g depends on the upper-level reuse groups. We update N_i^g of each level starting from the top group level. In contrast, $\mathcal{D}_{i+1}^{sf}$ is determined by lower-level reuse groups. We make another pass on the reuse group hierarchy from bottom up to update $\mathcal{D}_{i+1}^{sf}$ of each level as shown in Algorithm 1.

5 PRUNING THE PARAMETER SEARCH SPACE

Algorithm 1 provides an analytical calculation for the number of DRAM accesses, A^d, for a given point in the parameter space $\mathcal{X}^s$. This avoids expensive memory access simulation for each point in this space. However, this solves only part of the problem, as the parameter space is still large,

ALGORITHM 1: Spilling Calculation

function GetSpilling($\boldsymbol{I}^d, \mathcal{B}^s, M_t$)
  Build reuse group hierarchy with $\boldsymbol{I}^d, \mathcal{B}^s$
  $\boldsymbol{I}^d, \mathcal{B}^s$ give N_i^b, R_i, $\mathcal{D}_i^{b,r}$, $S_i^{b,r}$ of group level i
  $i = 1$
  while $i \neq l^{max}$ **do**
    $N_{i+1}^g = N_i^g \cdot N_i^b \cdot R_i$, $i++$
  $\mathcal{D}_i^{sf}$ = one data block, $i--$
  while $i \neq 1$ **do**
    if group_type == "complete" **then**
      $\mathcal{D}_i^{sf} = \mathcal{D}_i^r$
    else
      $\mathcal{D}_i^{sf} = \mathcal{D}_i^r \cup \mathcal{D}_{i+1}^{sf}$
  return $A_t^d(M_t)$

e.g., 1.85×10^{14} for the example instance **AN1** in Section 3. We tackle this by developing a set of techniques for pruning this search space.

5.1 Automatic Memory Partitioning

As shown in Figure 4, in this section, we introduce the first pruning technique. For a given loop ordering $\boldsymbol{I}^d$ and block dimensions $\mathcal{B}^s$, this pruning helps us avoid sweeping the full range of SRAM partitioning parameters $\mathcal{N}^{sb}$. The problem of SRAM partitioning is defined as optimizing the number of SRAM banks each tensor has, denoted as N_t^{sb}, so the total number of DRAM accesses from all tensors is minimized, as summarized in Equation (9). M^{sb} and N_{in}^{sb} are the SRAM bank size and the input number of SRAM banks, respectively.

$$\underset{\mathcal{N}^{sb}}{\text{minimize}} \sum_t A_t^d(N_t^{sb} \cdot M^{sb}) \quad \text{subject to} \sum_t N_t^{sb} = N_{in}^{sb} \tag{9}$$

The principle used here is to allocate SRAM space to the tensor where this piece of memory can hold more data uses. This guarantees memory being best utilized for resolving data spilling. For every tensor, its data spilling decreases as the SRAM size it has increases, but the level of decrease, or the sensitivity, varies with the tensor, the design parameters, and the memory size. We will show the monotonicity of the spilling sensitivity and how we can utilize that for allocating memory.

More formally, we define the sensitivity function of DRAM accesses for tensor t as $F_t^{d,sen}(m)$ in Equation (10). $A_t^d(m)$ is the number of DRAM accesses from tensor t for SRAM size m. As introduced in Section 3.3, each SRAM bank is completely occupied by one tensor, thus, we only consider $F_t^{d,sen}(m)$ values when m is a multiple of SRAM bank size M^{sb}. The initial value for m is $N_t^{sb0} \cdot M^{sb}$, where N_t^{sb0} is the minimum number of SRAM banks required to hold one tensor t's data block. Then, m increases in steps of SRAM bank size M^{sb}. Lemma 5.1 states that $F_t^{d,sen}(m)$ is a non-increasing function.

$$F_t^{d,sen}(m) = \frac{A_t^d(m) - A_t^d(m + M^{sb})}{M^{sb}}, \quad m = i \cdot M^{sb}, N_t^{sb0} \leq i \leq N_{in}^{sb}, i \in \mathbb{N} \tag{10}$$

Lemma 5.1. *For tensor t, the DRAM accesses sensitivity $F_t^{d,sen}(m)$ does not increase as m, the SRAM size available for tensor t, increases.*

ALGORITHM 2: Memory Partitioning

```
function MemoryPartition(F^{d,sen})
    for t in T do N_t^{sb} = N_t^{sb0}, M_t = N_t^{sb0} · M^{sb},
    N^{sb} = 0
    while N^{sb} < N_{in}^{sb} do
        f_c = 0, t_c = NULL
        for t in T do
            if F_t^{sen}(M_t) > f_c then
                f_c = F_t^{d,sen}(M_t), t_c = t
        if t_c! = NULL then M_{t_c} += M_{sb}, N_t^{sb} ++, N^{sb} ++
    return N^{sb}
```

Proof. Assume the spill-free memory sizes for groups in an n-level hierarchy from the bottom to top are $M_n^{sf}, \ldots, M_1^{sf}$ and $M_{i+1}^{sf} < M_i^{sf}$ for $i < n$. When m, the SRAM size allocated for tensor t, falls in the range $[M_{i+1}^{sf}, M_i^{sf})$, the groups located in level $i+1$ and below are spilling free. When m increases in this range, the number of data spilling in level-i reuse groups decreases but that in groups above level i will remain the same. Therefore, the value of $F_t^{d,sen}(m)$ for $m \in [M_{i+1}^{sf}, M_i^{sf})$ is controlled by reuse groups in level i.

From Equations (7) and (8), for data spilling from level-i reuse groups, its sensitivity towards memory size change can be computed as $N_i^g \cdot N_i^c$ as given in Equation (11). N_i^g is the number of reuse group instances in level i. N_i^c is the number of spilling cycles in each reuse group.

$$F_t^{d,sen}(m)_{m\in[M_{i+1}^{sf}, M_i^{sf})} = N_i^g \cdot N_i^c \tag{11}$$

Equation (12) calculates N_{i+1}^g, the number of reuse group instances in level i+1. R_i and N_i^b represent the number of use rounds and the number of subgroups in each round for level-i groups. Therefore, each level-i group has $N_i^b \cdot R_i$ subgroups, which are in level $i+1$. For reuse group in level i, its number of spilling cycles N_i^c is upper bounded by $N_i^b \cdot R_i$, as shown in Equation (13). By combining Equations (12) and (13), we see that the sensitivity in the range $[M_{i+2}^{sf}, M_{i+1}^{sf})$ is greater than that in $[M_{i+1}^{sf}, M_i^{sf})$, as given in Equation (14).

$$N_{i+1}^g = N_i^g \cdot N_i^b \cdot R_i \tag{12}$$

$$N_i^c <= (R_i - 1) \cdot N_i^b / (N_i^b - 1) < N_i^b \cdot R_i \tag{13}$$

$$F_t^{d,sen}(m)_{m\in[M_{i+2}^{sf}, M_{i+1}^{sf})} = N_{i+1}^g \cdot N_{i+1}^c > N_i^g \cdot N_i^c = F_t^{d,sen}(m)_{m\in[M_{i+1}^{sf}, M_i^{sf})} \tag{14}$$

□

This lemma forms the basis of Algorithm 2, which gives the procedure of allocating SRAM space to each tensor. N_t^{sb} and M_t are the number of SRAM banks and the corresponding SRAM size tensor t gets. Initially, each tensor gets several SRAM banks required for holding only one data block, given by N_t^{sb0}. We use N^{sb} to track the number of SRAM banks already being allocated. When N^{sb} is still smaller than N_{in}^{sb}, the total number of SRAM banks available, we pick the tensor t_c that has the greatest $F_t^{d,sen}$ value at M_t and assign an extra SRAM bank to it. The sensitivity value represents the efficiency of utilizing the piece of SRAM space to eliminate the data spills. Algorithm 2 selects the tensor with maximum efficiency to assign the next piece of memory. The non-increasing characteristic of the sensitivity function guarantees that other tensors will not give better efficiency of using this memory, and the selected tensor is the optimum choice.

Theorem 5.2. *Algorithm 2 returns the optimum solution to memory partitioning problem defined in Equation (9).*

Proof. $A_t^d(N_t^{sb} \cdot M^{sb})$ of tensor t can be calculated by subtracting from $A_t^d(N_t^{sb0} \cdot M^{sb})$ the number of DRAM accesses decreased when the SRAM size is expanded from $N_t^{sb0} \cdot M^{sb}$ to $N_t^{sb} \cdot M^{sb}$, as given in Equation (15).

$$A_t^d(N_t^{sb} \cdot M^{sb}) = A_t^d(N_t^{sb0} \cdot M^{sb}) - \sum_{m=N_t^{sb0} \cdot M^{sb}}^{N_t^{sb} \cdot M^{sb}} F_t^{d,sen}(m) \tag{15}$$

$A_t^d(N_t^{sb} \cdot M^{sb})$ of each tensor is constant for given $\boldsymbol{I}^d$ and $\mathcal{B}^s$. The objective function for memory partitioning now becomes maximizing $\sum_t \sum_{m=N_t^{sb0} \cdot M^{sb}}^{N_t^{sb} \cdot M^{sb}} F_t^{d,sen}(m)$. With constraint $\sum_t N_t^{sb} = N_{in}^{sb}$, there are $N^{sen} = N_{in}^{sb} - \sum_t N_t^{sb0}$ number of $F^{d,sen}$ values being accumulated in the new objective. This is maximized if the N^{sen} largest $F_t^{d,sen}$ values from all tensors are selected. Lemma 5.1 essentially states that $F_t^{d,sen}$ values in tensor t are ranked from high to low. If x $F_t^{d,sen}$ values are selected from tensor t, then they must be the first x ones. Thus, we can get the N^{sen} largest $F_t^{d,sen}$ values by merging the ranking from all tensors and picking the first N^{sen} candidates, as done in Algorithm 2. □

After applying memory partitioning, CNNFlow can reduce the size of the parameter space of the specific problem instance **AN1** from 1.8×10^{14} to 5.7×10^{11}.

5.2 Block Dimension (BD) Pruning

The second pruning technique helps with reducing the sweeping of block dimensions $\mathcal{B}^s$. For determining the value of block dimension B_i^s of loop i, we do not need to sweep every integer value between 1 to the input dimension size B_i^d. For B_i^s values that cannot fully divide B_i^d, we pad the last data block. Note that to simplify data movement, a data block with or without padding occupies the same SRAM space. The memory space allocated to padding data is not utilized to resolve data spilling. Thus, block dimensions causing less padding use memory more efficiently. Among a set of block dimensions that result in the same number of iterations, the one that requires minimum padding has the lowest data spilling. Thus, from 1 to B_i^d, we only need to consider each smallest B_i^s value that yields a new number of loop iterations $N_i^s = \lceil \frac{B_i^d}{B_i^s} \rceil$.

Lemma 5.3. *The optimum block dimension for loop i with input dimension B_i^d will be in list $\boldsymbol{V}_i^s = 1, 2, \ldots, \lceil \sqrt{B_i^d} \rceil, \ldots, \lceil \frac{B_i^d}{3} \rceil, \lceil \frac{B_i^d}{2} \rceil, B_i^d$.*

Proof. The number of iterations of loop l, N_i^s, is calculated as $N_i^s = \lceil \frac{B_i^d}{B_i^s} \rceil$. B_i^s with value in the range $[\boldsymbol{V}_i^s[j], \boldsymbol{V}_i^s[j+1])$ results in the same number of loop iterations as $\boldsymbol{V}_i^s[j]$. Assume loop i is a related loop for tensor t. It forms a reuse or partial reuse group together with the closest reuse loop or partial reuse loop above. The number of loop iterations N_i^s of i controls the number of subgroups N_i^b in each group's use round. The block dimension B_i^s controls the size of data being reused in each subgroup $S_i^{b,r}$. Based on Equation (7), the total number of DRAM accesses is increased with the same N_i^b but larger $S_i^{b,r}$. For computation kernels that we target, loop i is a related loop for at least one tensor. Therefore, among B_i^s that give the same N_i^s, the minimum value of B_i^s will give the minimum number of data spilling. The data spilling when $B_i^s = \boldsymbol{V}_i^s[j]$ must be lower than that with $B_i^s \in (\boldsymbol{V}_i^s[j], \boldsymbol{V}_i^s[j+1])$. □

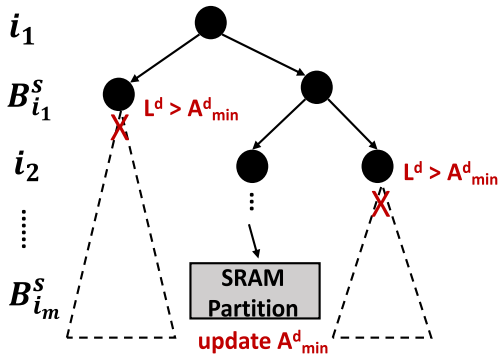


Fig. 9. Pruning search tree with branch and bound.

The number of points to sweep in block dimension B^s_i is reduced to $O(\lfloor 2\sqrt{B^d_i} \rfloor)$ from $O(B^d_i)$. The design space for **AN1** is further reduced from 5.7×10^{11} to 1.9×10^9.

5.3 Branch and Bound Search

As shown in Figure 4, in this section, we further prune the design space formed with loop indicator $\boldsymbol{I}^d$ and pruned blocking dimensions $\mathcal{B}^s$. We can use a search tree to represent the search space such that at each level in the search tree, we explore the values of one parameter. As shown in Figure 9, from top down in the tree, we explore values of loop indicator i_l and block dimension $B^s_{i_l}$ for the lth loop. Instead of fully traversing every path in the tree, we propose using a branch-and-bound algorithm that uses the best solution found thus far to prune the search. At each node, we evaluate the objective function lower bound L^d its subtree could achieve. If L^d is already larger than A^d_{min}, the best objective value seen so far, then we can skip searching in this subtree. Otherwise, the search continues in the subtree, as it may have a better solution. When we reach a leaf node in the tree, we have a specific value for $\boldsymbol{I}^d$ and $\mathcal{B}^s$ for that path. Thus, we can run memory partitioning as given in Algorithm 2 to compute A^d for this path. We update A^d_{min} if A^d is smaller for this path.

Algorithm 3 provides the branch-and-bound algorithm. Function LoopInd(l) explores indicator candidates for the lth loop. For each branch, it estimates the subtree lower bound using Function GetL($\dot{\boldsymbol{I}}^d, \dot{\mathcal{B}}^s$). $\dot{\boldsymbol{I}}^d$ and $\dot{\mathcal{B}}^s$ represent the partially determined loop ordering $\boldsymbol{I}^d$ and block dimensions $\mathcal{B}^s$ for the non-leaf node. If the subtree could possibly update the best objective found so far, then it calls Function LoopBlock(l,i). Function LoopBlock(l,i) explores block dimensions for loop i at the lth location. If the lth loop is already the inner most loop, then it runs memory partitioning given in Algorithm 2 to get the objective value for the current configuration path. Otherwise, it estimates the subtree lower bound and continues exploring indicators for $(l+1)$th loop if the objective could further be improved in the subtree.

The main challenge in the branch-and-bound algorithm is the lower bound estimation for the non-leaf node, given in Function GetL($\dot{\boldsymbol{I}}^d, \dot{\mathcal{B}}^s$). We construct the lower bound of total DRAM accesses $L(\dot{\boldsymbol{I}}^d, \dot{\mathcal{B}}^s)$ as the sum of lower bound of DRAM accesses for each tensor, $L(\dot{\boldsymbol{I}}^d, \dot{\mathcal{B}}^s) = \sum_t L_t(\dot{\boldsymbol{I}}^d, \dot{\mathcal{B}}^s)$. Tensor t can maximally reduce its DRAM accesses if: (1) it gets the maximum SRAM size M^u_t; (2) loops are ordered and blocked in a way that the data spilling is minimized when $M_t = M^u_t$. Therefore, to estimate L_t, the DRAM accesses lower bound for tensor t, we first need to estimate its memory size upper bound M^u_t and then find the optimum loop ordering and blocking for tensor t. Each of these tasks is addressed in one subsection in the following, as shown in Figure 4.

ALGORITHM 3: Branch and bound algorithm

$\dot{I}^d$ = [], $\dot{\mathcal{B}}^s$ = [], $\mathcal{I}_c = \mathcal{I}_{in}$, LoopInd(0)
function LoopInd(l)
  for $i \in \mathcal{I}_c$ **do**
    $\dot{I}^d[l] = i$, $\mathcal{I}_c.erase(i)$
    if GetL($\dot{I}^d, \dot{\mathcal{B}}^s$) $< A_{min}$ **then** LoopBlock(l, i)
function LoopBlock(l, i)
  for b in BD pruned values **do**
    $\dot{\mathcal{B}}^s[i] = b$, UpdateFsenu($i$, t), UpdateFsenl(i, t)
    if l == $\mathcal{I}_{in}$.size() **then** A = MemoryPartition($\mathcal{F}^{d,sen}$), update A^d_{min}
    else if GetL($\dot{I}^d, \dot{\mathcal{B}}^s$) $< A^d_{min}$ **then** LoopInd(l + 1)
function UpdateFsenu(i, t)
  for t in T **do**
    if i is reuse loop **then** $p^u_t = q^u_t, j++$
    else if i is partial reuse loop **then**
      $\mathcal{D}^{b,sf} \rightarrow$ data in first round, $q^u_t = S(\mathcal{D}^{b,sf})$
      for m in $[p^u_t, q^u_t]$ **do** $F^{d,sen\_u}_t[m] = N^g_g \cdot N^c_g$
      $p^u_t = q^u_t, j++$
    else
      $\mathcal{D}^{b,sf} \rightarrow$ data in subgroup, $q^u_t = S(\mathcal{D}^{b,sf})$
      for m in $[p^u_t, q^u_t]$ **do** $F^{d,sen\_u}_t[m] = N^g_g \cdot N^c_g$
function UpdateFsenl(i, t)
  for t in T **do**
    if i is reuse loop **then**
      $j++$, $\mathcal{D}^{sf} \rightarrow$ current group, $p^l_t = S(\mathcal{D}^{sf})$
    else if i is partial reuse loop **then**
      $j++$, adjust q^l_t = min dblk size
      for m in $[0, q^l_t]$ **do** $F^{d,sen\_l}_t[m]$ = MAX_INT
      $\mathcal{D}^{sf} \rightarrow$ first round, $p^l_t = S(\mathcal{D}^{sf})$
    else
      adjust q^l_t = min dblk size
      for m in $[0, q^l_t]$ **do** $F^{d,sen\_l}_t[m]$ = MAX_INT
    for m in $[q^l_t, p^l_t]$ **do** $F^{d,sen\_l}_t[m] = N^g_g \cdot N^c_g$
function GetL($\dot{I}^d, \dot{\mathcal{B}}^s$)
  for t in T **do**
    M^u_t = MemoryPartition($F^{d,sen\_u}_t, \{F^{d,sen\_l}_i\}_{i \neq t}$)
    $\dot{I}^d, \dot{\mathcal{B}}^s$ in subtree configured as $\dot{I}^{t\_opt}, \dot{\mathcal{B}}^{t\_opt}$
    $L_t = \sum_{i \leq i^c} A^d_{t,i}$, i^c is lowest defined group level
    $L+ = L_t$
  return L

5.3.1 Estimate M^u_t - SRAM Size Upper Bound for Tensor t. In memory partitioning, the memory bank is assigned to tensors based on the spilling sensitivity function $F^{d,sen}$ of each tensor. This sensitivity function is fully determined by design parameters of loop ordering $\dot{I}^d$ and block dimensions $\mathcal{B}^s$. $\dot{I}^d$, $\dot{\mathcal{B}}^s$ on a search path can partially define $F^{d,sen}$ for each tensor. For each tensor, we maintain a maximum sensitivity function $F^{d,sen\_u}_t(m)$ and a minimum sensitivity function

$F_t^{d,sen\_l}(m)$ that upper bounds and lower bounds $F_t^{d,sen}$, respectively. To get the maximum memory size M_t^u for tensor t, we use t's maximum sensitivity function and the other tensors' minimum sensitivity function as input for memory partitioning during the search.

With $\dot{\boldsymbol{I}}^d$, $\dot{\mathcal{B}}^s$ being specified in the search tree, the reuse groups are gradually defined from top to bottom in the hierarchy. The group level i determines the value of $F^{d,sen}(m)$ in the scope of m starting from the spill-free memory size of its subgroup M_{i+1}^{sf} to its own M_i^{sf}. We initialize F_t^{d,sen_u} with value +INF and F_t^{d,sen_l} with value 0 over the full memory range. We then update F_t^{d,sen_u} and F_t^{d,sen_l} in each step of determining a loop indicator and its block dimension using the new information we obtained during the search.

Maintaining F_t^{d,sen_u}. A newly declared group level, say, the jth level, updates the sensitivity function in the range of M_{j+1}^{sf} and M_j^{sf}. As we mentioned in Algorithm 1 for the spilling calculation, $\mathcal{D}_j^{sf}$ and its size M_j^{sf} of a level-j group recursively depends on its subgroups. As loops are specified from the outer-most one to the inner ones in the search tree, and thus the group levels are declared from the higher one to the lower ones, we are unable to determine the exact value of $\mathcal{D}_j^{sf}$ and M_j^{sf} based on partial information on the search tree path. Instead, we estimate $M_j^{sf\_u}$, the upper bound of M_j^{sf}, and use it to identify the range of sensitivity functions we could update. For each group level j being declared, we update the sensitivity function in the range of $M_{j+1}^{sf\_u}$ and $M_j^{sf\_u}$. Function UpdateFsenu(i, t) in Algorithm 3 implements the F_t^{d,sen_u} update.

M_j^{sf} of a level-j group is upper bounded by the group data size. $M_1^{sf\_u}$, the upper bound of spill-free memory size for groups in the top level, is estimated as the input tensor size. Assume the most recently declared group level is level j, before the next reuse/partial reuse loop appears to define level $j+1$, we need to estimate $M_{j+1}^{sf\_u}$ and update F_t^{d,sen_u} between the range of $M_{j+1}^{sf\_u}$ and $M_j^{sf\_u}$. $M_{j+1}^{sf\_u}$ is initialized with the value of $M_j^{sf\_u}$ and gets adjusted as the new loop being specified. Assume the next loop is a related loop, its loop blocking reduces the data size of groups in level $j+1$, therefore reduces the value of $M_{j+1}^{sf\_u}$. If the next loop is a partial reuse loop, then groups in level $j+1$ are partial reuse groups. The current $M_{j+1}^{sf\_u}$ is estimated with the data size of the group. However, for partial reuse group, its spill-free memory size is upper bounded by the data size of one use round. Therefore, we can update $M_{j+1}^{sf\_u}$ to obtain a tighter bound estimation. If next loop is a reuse loop, then it declares group level $j+1$ as complete reuse group without changing the estimation of $M_{j+1}^{sf\_u}$. After the appearance of either partial reuse loop or reuse loop, the current group level is switched to level $j+1$, and the task then is to estimate the value of $M_{j+2}^{sf\_u}$. As shown in Function UpdateFsenu(i, t), we use pointer p_t^u to indicate the estimation of $M_j^{sf\_u}$ and pointer q_t^u to indicate the estimation of $M_{j+1}^{sf\_u}$. p_t^u and q_t^u are initialized to be the tensor size and are adjusted accordingly as each new loop is specified.

At each step, the value of sensitivity function between q_t^u and p_t^u is decided by groups in level j. As shown in Equation (11), it is estimated as the product of N_j^g, the number of group instances in level j, and N_j^c, the number of spilling cycles in each level-j group. As given in Equation (12), N_j^g depends on N_{j-1}^g, which further depends on upper group levels. As the group levels are declared from top down, we can obtain the exact value of N_j^g. However, we can only obtain the upper bound estimate of N_j^c. From Equation (7), $N_j^c = N_j^b \cdot S_j^{b,r} \cdot (R_j - 1)/(N_j^b \cdot S_j^{b,r} - D_j)$. D_j is the size of intersection set between $\mathcal{D}_{j+1}^{sf}$ and $\mathcal{D}_j^{b,r}$, $D_j = S(\mathcal{D}_{j+1}^{sf} \cap \mathcal{D}_j^{b,r})$. $\mathcal{D}_j^{b,r}$ represents the dataset

in a subgroup that are being reused across current use rounds and its size is $S_j^{b,r}$. Their values can be fully determined by the specified loops on the search path. $\mathcal{D}_{j+1}^{sf}$ is the spill-free dataset of the subgroup. It depends on inner reuse groups that are not specified yet. Thus, $\mathcal{D}_{j+1}^{sf}$ can only be estimated with its superset, just as its size $\mathcal{M}_{j+1}^{sf}$ being estimated with an upper-bound value. Therefore, we can only obtain the upper bound value of D_j and N_j^c. Thus, the sensitivity value we estimate between q_t^u and p_t^u is the upper bound of the real sensitivity value that is given by the current group level.

Maintaining $F_t^{d,sen\_l}(m)$. To maintain $F_t^{d,sen\_l}(m)$, for a new group level being declared, say, level j, we update the sensitivity function between $M_{j+1}^{sf\_l}$ and $M_j^{sf\_l}$. $M_j^{sf\_l}$ is the lower bound estimation of M_j^{sf}. Function UpdateFsenl(i, t) in Algorithm 3 implements the F_t^{d,sen_l} update.

To obtain $M_{j+1}^{sf\_l}$, we assume the current group level is the lowest group level such that its subgroup is the data block. The data block size is estimated with its minimum value assuming the block value for all undetermined related loop and partial reuse loops is equal to 1. The corresponding spill-free dataset $\mathcal{D}_{j+1}^{sf}$ is then a data block instance. $M_j^{sf\_l}$ is estimated as the size of data being reused in level-j group S_j^r. In Function UpdateFsenl(i, t), we use pointer p_t^l to indicate $M_j^{sf\_l}$ and pointer q_t^l for $M_{j+1}^{sf\_l}$, which essentially estimates the minimum data block size. Pointer p_t^l and q_t^l are updated as each new loop is specified. For a related loop, if its blocking dimension is greater than 1, then the minimum data block size increases, thus, we expand both p_t^l and q_t^l. For a reuse loop, it does not change the estimation of data block size, but it declares a new level of reuse group, level $j+1$. $M_{j+1}^{sf\_l}$ is not estimated as the data block size, as it is lower bounded by S_{j+1}^r, the size of data being reused in level $j+1$. We then estimate $M_{j+2}^{sf\_l}$ with data block size and update $M_{j+1}^{sf\_l}$ accordingly. For partial reuse loop, it expands the data block and also declares a new group level.

As SRAM needs to hold at least one data block, the spilling sensitivity value within q_t^l is set as +INF. The sensitivity value between q_t^l and p_t^l is decided by the current group level. The estimation of sensitivity value is lower than its real value, as we use the subset of $\mathcal{D}_{j+1}^{sf}$, which yields an underestimation of N_j^c.

Lemma 5.4. *$F_t^{d,sen\_u}(m)$ and $F_t^{d,sen\_l}(m)$ give an upper and lower bound, respectively, for $F_t^{d,sen}(m)$ of tensor t. M_t^u is the upper bound of memory size tensor t can have.*

Proof. Assume the partially defined $\dot{\mathcal{I}}^d$, $\dot{\mathcal{B}}^s$ specifies a partial hierarchy of reuse groups with j levels. In the range $[M_{j+1}^{sf\_u}, M_j^{sf\_u}]$, the value of F_t^{d,sen_u} is updated by Equation (11). $M_j^{sf\_u}$ is the upper bound of M_j^{sf} for groups in level j. The calculation of $F^{d,sen}()$ utilizes $D_j = S(\mathcal{D}_{j+1}^{sf} \cap \mathcal{D}_j^{b,r})$. With $\mathcal{D}_{j+1}^{sf}$ being estimated with a superset of its real set, the value of the sensitivity function for reuse group in level j is overestimated, as shown in Equation (16).

$$F_t^{d,sen\_u}(m1)_{m1\in[M_{j+1}^{sf\_u}, M_j^{sf\_u}]} >= F_t^{d,sen}(m2)_{m2\in[M_{j+1}^{sf}, M_j^{sf}]}, j \in [1, n] \tag{16}$$

Further, Lemma 5.1 states the value of $F^{d,sen}$ does not increase with the memory size, as described in Equation (17).

$$F_t^{d,sen}(m2)_{m2\in[M_{j+1}^{sf}, M_j^{sf}]} >= F_t^{d,sen}(m2)_{m2\in[M_{j+1}^{sf\_u}, M_j^{sf\_u}]}, j \in [1, n] \tag{17}$$

Combining Equations (16) and (17), we show that F^{d,sen_u} gives the upper bound of $F^{d,sen}$ for each of its step regions starting with $M_{n+1}^{sf\_u}$. Within $M_{n+1}^{sf\_u}$, F^{d,sen_u} is set with +INF, which should upper bound $F^{d,sen}$ as well.

Similarly, F^{d,sen_l} estimates the sensitivity value of group level j between $M_{j+1}^{sf\_l}$ and $M_j^{sf\_l}$. $M_j^{sf\_l}$ is the lower bound estimation of M_j^{sf}. As D_j is estimated with its lower bound, the calculation of F^{d,sen_l} underestimates the real sensitivity value of the current group level. Combined with the non-increasing property of the sensitivity function given in Lemma 5.1, F^{d,sen_l} provides the lower bound of $F^{d,sen}$. □

5.3.2 Optimum Loop Ordering and Blocking for Tensor t. Equation (18) gives the configuration of unexplored $\boldsymbol{I}^d$ and $\mathcal{B}^s$ in the subtree with which tensor t will have the lowest number of DRAM accesses. The loops from candidate sets are ranked as related loops followed by partial reuse loops and then followed by reuse loops. The loop-blocking dimension is one for related and partial reuse loops and is equal to the input size for reuse loops.

$$\begin{aligned} \boldsymbol{I}^{t\_opt} &= [\mathcal{I}_c^{t\_related}, \mathcal{I}_c^{t\_partial}, \mathcal{I}_c^{t\_reuse}] \\ \mathcal{B}^{t\_opt}[l] &= \begin{cases} 1 & \text{if } l \in \mathcal{I}_c^{t\_related} \cup \mathcal{I}_c^{t\_partial} \\ B_l^d & else \end{cases} \end{aligned} \tag{18}$$

Lemma 5.5. *Given partially defined $\dot{\boldsymbol{I}}^d$ and $\dot{\mathcal{B}}^s$ for tensor t with local memory size M_t^u, selecting the rest of $\boldsymbol{I}^d$ and $\mathcal{B}^s$ as $\boldsymbol{I}^{t\_opt}$ and $\mathcal{B}^{t\_opt}$ gives the minimum number of data spilling in tensor t.*

Proof. **Blocking and ordering of reuse loops.** Assume loop i is a reuse loop for tensor t and its input dimension is B_i^d. Loop i declares a level of complete reuse groups. Its blocking dimension B_i^s only affects the number of use rounds the complete reuse group has. Having $B_i^s = B_i^d$ only gives one use round and eliminates the data spilling. Loops with just one iteration have the same effect on the data use pattern regardless of their ordering relative to other loops. We can put all reuse loops as the innermost loops.

Blocking of related loops. Assume partial reuse loops i^p and related loops i^r are ordered as $i^{p_1}, i^{r_1}, \ldots, i^{r_3}, i^{p_2}, i^{r_4}, \ldots, i^{r_j}, i^{p_3} \ldots, i^{p_n}$ from the outer most one to the inner ones. i^{p_k}, $k \in [1, n]$, and its following related loops form the kth group level with the type being partial reuse group.

$$A_{t,i}^d(M_t) = \begin{cases} 0, & \text{if } M_t >= M_i^{sf} \\ N_i^g \cdot N_i^b \cdot S_i^{b,r} \cdot (R_i - 1), & \text{else if } M_t < M_{i+1}^{sf}, \\ N_i^g \cdot N_i^c \cdot SP_i, & \text{otherwise} \end{cases} \tag{19}$$

$$N_i^c = \frac{N_i^b \cdot S_i^{b,r} \cdot (R_i - 1)}{N_i^b \cdot S_i^{b,r} - S(\mathcal{D}_i^{b,r} \cap \mathcal{D}_{i+1}^{sf})}, SP_i = N_i^b \cdot S_i^{b,r} + S\left(\mathcal{D}_{i+1}^{sf} - \mathcal{D}_i^{b,r}\right) - M_t^u$$

The blocking dimensions of level k's related loops affect the data spilling generated in levels k and below. From the spilling calculation given in Equations (7) and (8), the number of data spilling from groups in level i, represented by $A_{t,i}^d$, is given in Equation (19). N_i^g is the number of group instances in level i, and it is determined by blocking of loops above level i. N_i^b is the number of subgroups in one use round of the level-i group, and it is decided by the blocking of level i's related loops. $\mathcal{D}_i^{b,r}$ is the dataset from one subgroup that is being reused in level-i group, and $S_i^{b,r}$ gives its size. $\mathcal{D}_i^{b,r}$ and $S_i^{b,r}$ are decided by blocking dimension of loops in the current level and levels above. $\mathcal{D}_{i+1}^{sf}$ represents the spill-free dataset of the subgroup, and its size is also affected by blocking dimension of loops in the current level and levels above.

In the example with $i^{p_1}, i^{r_1}, \ldots, i^{r_3}, i^{p_2}, i^{r_4}, \ldots, i^{r_j}, i^{p_3} \ldots, i^{p_n}$, related loop i^{r_j} controls the reuse groups in the second level. Assume loop i^{r_j} is blocked with block size $B^s_{r_j}$ and yields loop iterations $N^s_{r_j}$. In the following, we show how this loop's blocking affects reuse groups in level 2, in levels below and levels above. This proof can be applied to related loops other than i^{r_j}.

(1) Effect of $B^s_{r_j}$ on group level 2: $B^s_{r_j}$ adjusts the subgroup size and the number of subgroups. If $B^s_{r_j}$ increases by x times, then the number of subgroups N^b_2 decreases x times while the items related to subgroup size, including $S^{b,r}_2$, $S(\mathcal{D}^{b,r}_2 \cap \mathcal{D}^{sf}_3)$ and $S(\mathcal{D}^{sf}_3 - \mathcal{D}^{b,r}_2)$, all increase by x times. If $M^u_t < M^{sf}_3$, then $A^d_{t,2}$ does not change with $B^s_{r_j}$. But if M^u_t is between M^{sf}_3 and M^{sf}_2, then $A^d_{t,2}$ increases with $B^s_{r_j}$ as both N^c_2 and SP_2 increase. Thus, data spilling from level-2 reuse groups benefits from $B^s_{r_j} = 1$.

(2) Effect of $B^s_{r_j}$ on group levels below: $B^s_{r_j}$ controls the number of group instances and data block size for group levels below. We use group level 3 as example for explanation. If $B^s_{r_j}$ increases by x times, then the instance number of level-3 group N^g_3 is reduced by x times. However, the block dimension increasing x times makes $S^{b,r}_3$, $S(\mathcal{D}^{b,r}_3 \cap \mathcal{D}^{sf}_4)$ and $S(\mathcal{D}^{sf}_4 - \mathcal{D}^{b,r}_3)$ all increase by x times. If $M^u_t < M^{sf}_4$, then $A^d_{t,3}$ does not change with $B^s_{r_j}$. However, if M^u_t is between M^{sf}_4 and M^{sf}_3, then SP_2 increases by more than x times, and even with N^g_3 decreasing x times, $A^d_{t,3}$ still increases with $B^s_{r_j}$. Thus, data spilling from levels below also benefits from $B^s_{r_j} = 1$.

(3) Effect of $B^s_{r_j}$ on group levels above: $B^s_{r_j} = 1$ gives a smaller data block size than $B^s_{r_j} > 1$. With the dependency $\mathcal{D}^{sf}_i = \mathcal{D}^r_i \cup \mathcal{D}^{sf}_{i+1}$, $B^s_{r_j} = 1$ may yield a smaller $\mathcal{D}^{sf}_i$ for group level i. Both N^c_i and SP_i benefit from smaller $\mathcal{D}^{sf}_{i+1}$. Thus, $B^s_{r_j} = 1$ also benefits the reuse groups in levels above.

By considering the effects of $B^s_{r_j}$ on reuse groups in level 2 itself, levels below and above, we can conclude $B^s_{r_j} = 1$ results in fewer data spilling than $B^s_{r_j} > 1$. Thus, related loops are blocked with blocking size equal to 1.

Ordering of related loops. We want to prove that moving i^{r_j} from following i^{p_2} to preceding i^{p_2} can benefit both groups in level 2 and above.

(1) Effect of i^{r_j} ordering on group level 2: By moving i^{r_j} from following i^{p_2} to preceding i^{p_2}, the number of level-2 reuse groups N^g_2 increases, but the number of subgroups in the use round N^b_2 decreases. The number of loop iterations of i^{r_j} is represented as $N^s_{r_j}$. Thus, N^g_2 increases by $N^s_{r_j}$ times and N^b_2 decreases by $N^s_{r_j}$ times. We can prove that with $M^u_t - S(\mathcal{D}^{sf}_3 - \mathcal{D}^{b,r}_2) > S(\mathcal{D}^{sf}_3 \cap \mathcal{D}^{b,r}_2)$, equivalently $M^u_t > M^{sf}_3$, $N^c_3 \cdot SP_3$ decreases by more than $N^s_{r_j}$ times. Thus, the total data spilling of group level 2 benefits from moving i^{r_j} from following i^{p_2} to preceding i^{p_2}.

(2) Effect of i^{r_j} ordering on group above: Having i^{r_j} belonging to level 1 increases its number of subgroups N^b_1 but decreases its subgroup size. Both changes are by a factor of $N^s_{r_j}$. This has the same effect as reducing the block dimension of a level-1 related loop by $N^s_{r_j}$ times. We have shown that a group level benefits from its related loop having a smaller block dimension. Thus, groups above level 2 also benefit from moving loop i^{r_j}.

Therefore, groups in level 2 and above benefit from moving related loop i^{r_j} from level 2 to level 1. Groups in levels below are not affected by this loop movement. The same proof can be applied to moving other related loops. We have all related loops preceding partial loops in $\boldsymbol{I}^{t\_opt}$. Their relative order can be arbitrarily defined, as they are within the same reuse group.

Blocking and ordering of partial reuse loops. After moving all related loops upward, partial reuse loops $i^{p_1}, i^{p_2}, \ldots, i^{p_n}$ are ordered as the inner-most loops. Assume partial reuse loop i^{p_1} defines reuse group level j and partial reuse loop i^{p_k} defines group level $j+k-1$. Without related loops, there is only one subgroup in each use round. The partial loop's blocking dimension $B^r_{p_k}$ determines the number of reuse times R_{j+k-1} and the data block size.

The partial reuse groups are spill-free if the memory size M^u_t is greater than the spill-free memory size of top-level partial groups M^{sf}_j. In this case, the data spilling are from group levels above j, which are defined by specified loops on the search path. Configuring partial reuse loops in the subtree to give the minimum value of M^{sf}_j can provide the minimum number of data spilling. However, if M^u_t is smaller than the minimum value of M^{sf}_j, then data spilling will occur in partial group level j. Reuse groups above level j require full data spilling. Thus, their total number is independent of M^{sf}_j. The minimum number of data spillings from level j and its lower levels requires exploration towards both ordering and blocking dimension of partial loops in the subtree. To simplify lower bound estimation, we only include data spilling above level j that are led by specified reuse/partial reuse loops on the search path. Its lowest number is reached at M^{sf}_j getting its minimum value.

The spill-free memory size M^{sf}_j of the top partial reuse group in the subtree is determined by both the ordering and blocking of partial loops $i^{p_1}, i^{p_2}, \ldots, i^{p_n}$. For each loop ordering, having the block dimension of all partial loops as 1, $B^r_{p_k} = 1$, decreases M^{sf}_j. But it still requires exploring the loop ordering of $i^{p_1}, i^{p_2}, \ldots, i^{p_n}$ to search for the minimum value of M^{sf}_j. □

Theorem 5.6. *Given partially defined $\boldsymbol{I}^d$ and $\boldsymbol{I}^d$ as $\dot{\boldsymbol{I}}^d$ and $\dot{\mathcal{B}}^s$, $L = \sum_t L_t(\dot{\boldsymbol{I}}^d, \dot{\mathcal{B}}^s)$ gives the lower bound of $A^d(\boldsymbol{I}^d, \mathcal{B}^s)$. The branch-and-bound algorithm in Algorithm 3 is complete.*

Proof. For each tensor t, with any configuration, its data spilling decreases as the memory size allocated to it increases. Having memory size upper bound M^u_t for tensor t gives the spilling lower bound for any configuration. Lemma 5.5 shows $\boldsymbol{I}^{t\_opt}$ and $\mathcal{B}^{t\_opt}$ give the optimum configuration for any value M^u_t may be. Using $\boldsymbol{I}^{t\_opt}$, $\mathcal{B}^{t\_opt}$ and M^u_t to calculate the tensor spilling gives the spilling lower bound.

The subtree is pruned only when the spilling lower bound is already higher than the best objective value found. This pruning will not miss the optimum solution. Thus, Algorithm 3 is a complete algorithm for searching the configuration space. □

For the scenario mentioned before, with only memory partitioning and BP pruning, the number of design points for **AN1** is 1.9×10^9, which is also the number of leaves in the search tree. The total number of nodes in the search tree (tree size) is 2.7×10^{12}. The degree of pruning is instance-specific, as it depends on how the objective function values are distributed in the design space. For this specific example, after applying the branch-and-bound algorithm, the pruned search tree size is only 2.1×10^4.

Therefore, after applying the memory partitioning, BD pruning, and brand-and-bound search tree pruning, we reduced the search space from a computationally impractical 1.85×10^{14} points to a very manageable 2.1×10^4 points.

6 INTEGRATED OPTIMIZATION

This section shows how we apply the same set of optimization techniques for solving **Problem 0** but with the analytic calculation of memory accesses and the lower bound estimation being updated accordingly. Figure 10 shows how the solutions used for **Problem 1** are updated to handle **Problem 0** and where each part of the updates is addressed in this section.

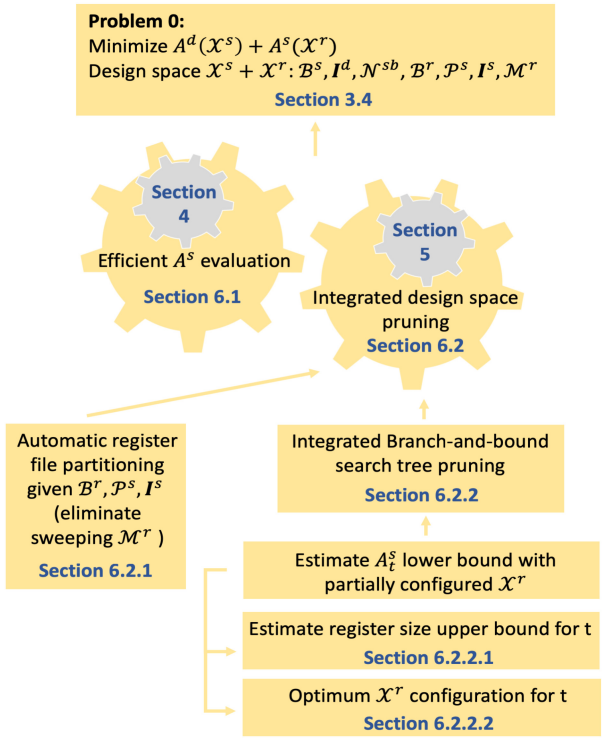


Fig. 10. An overview of solution for Problem 0.

6.1 Analytic Calculation for SRAM Accesses

As shown in Figure 10, to efficiently evaluate the objective function of one design point, in addition to the analytical calculation of A^d introduced in Section 4, we need analytical evaluation of A^s, the number of SRAM accesses, as well. The SRAM accesses include compulsory ones and also additional ones caused by data spilling from registers. Equation (2) in Section 3.4 gives the calculation of the number of compulsory SRAM accesses A^{sc}. Next, we consider the number of additional SRAM accesses, A^{sa}.

The number of additional SRAM accesses from each PE Block Computation is the same, given that they are controlled by the same $\boldsymbol{I}^s$ and $\mathcal{B}^r$ and that the PEs have uniform register partitioning. However, A_t^{sa} is not simply equal to the number from one PE, denoted as $A_t^{sa\_pe}$, times the total number of PEs. In CNNFlow, PEs can spatially share data blocks. We only need to re-fetch one copy of evicted data being shared by PEs from SRAM, and broadcast them to all PEs that use them. As introduced in Section 3, a batch of PE Block Computations is physically mapped to the PE array, and the batch dimension is given by the design parameter $\mathcal{P}^s$. With the batch dimension j, if each PE needs a data block with dimension B_j^{pe} but the datapath only needs to access SRAM for a block of dimension B_j^{dp} due to data sharing across PEs, then the ratio between A_t^{sa} and $A_t^{sa\_pe}$ given by dimension j is equal to $\frac{B_j^{dp}}{B_j^{pe}}$. Considering all loops, we use $f_{t,i}$ to represent the ratio between the number of data spilling in group level i generated from all PEs and that from one PE. As each loop can contribute to the batch dimension, $f_{t,i} = \prod_j f_{t,i}^j$ where $f_{t,i}^j$ represents the ratio contributed by batch dimension j. We list the computation of $f_{t,i}^j$ for reuse loop, related loop, and partial reuse loops in the following items:

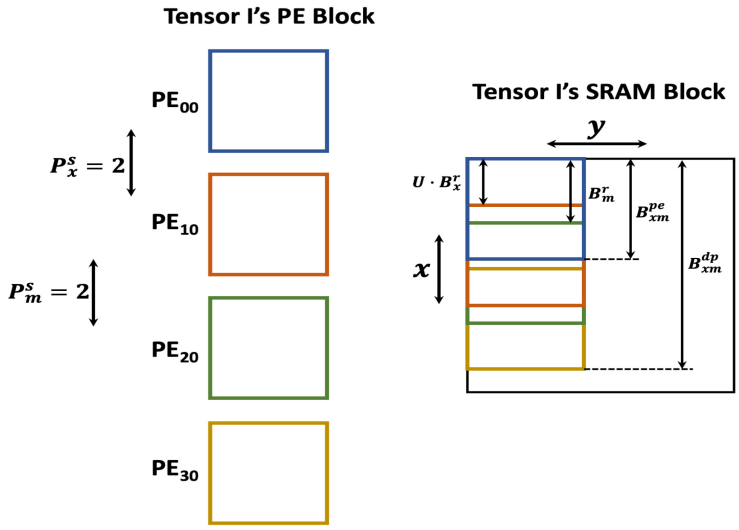
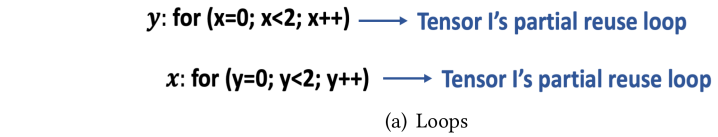
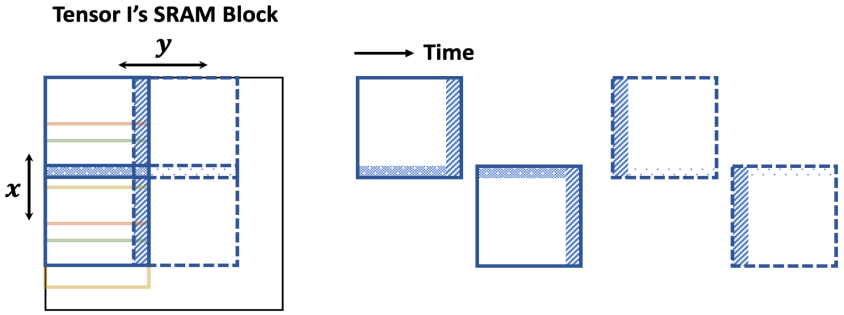


Fig. 11. Example of partial spatial data sharing.

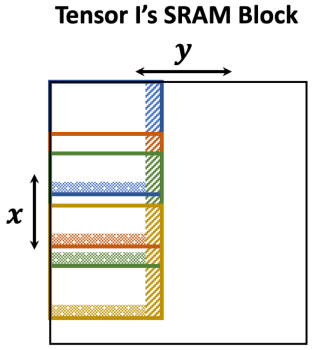
- $f_{t,i}^j$ of reuse loop: If the batch dimension j is formed by a reuse loop, then all PEs on this direction share the same data. Therefore, $f_{t,i}^j = 1$.
- $f_{t,i}^j$ of related loop: P_j^s PEs in the direction of related loop j do not share their evicted data. Therefore, $f_{t,i}^j = P_j^s$.
- $f_{t,i}^j$ of partial reuse loop: For the targeted CNN layers given in Figure 1, only tensor I in the convolution layer and the pooling layer have partial reuse loops x, y, m, and n. Figure 11 shows an example of spatial data sharing on the batch dimensions of x and m, with batch size equal to 2 on both dimensions ($P_x^s = 2$, $P_m^s = 2$). The adjacent PEs on the batch dimension of x overlap their data blocks by dimension $B_m^r - U$, as blocks in PE_{00} and PE_{10} in the figure. Adjacent PEs on batch dimension m overlap their data blocks by dimension $U \cdot (B_x^r - 1)$, as blocks in PE_{00} and PE_{20}. We use B_i^{rsp} to represent the spatial overlapping dimension on batch dimension i. Then, $B_x^{rsp} = B_m^r - U$, $B_m^{rsp} = U \cdot (B_x^r - 1)$.
  Tensor I's block size on the third dimension is given by loop blocking of x and m. We label the tensor I's third dimension as xm. We use B_{xm}^{pe} to represent the third dimension size of each data block inside PE. Similarly, we use B_{xm}^{dp} to represent the third dimension size of the data block formed by all PEs in the datapath. Then, we have $B_{xm}^{pe} = U \cdot (B_x^r - 1) + (B_m^r - 1)$ and $B_{xm}^{dp} = U \cdot (P_x^s \cdot B_x^r - 1) + (P_m^s \cdot B_m^r - 1)$.
  To distinguish from the spatial data reuse occurring across multiple physical PEs, we name the data reuse across use rounds inside each PE as temporal data reuse. We use B_j^{rti} to represent the dimension that data blocks in two adjacent iterations overlap inside one PE (temporal reuse dimension). For loop x, $B_x^{rti} = B_{xm}^{pe} - P_x^s \cdot (B_{xm}^{pe} - B_x^{rsp})$. If B_j^{rti} is smaller than zero, then the temporal data reuse effect is eliminated inside the PE. Loop j behaves as a related loop of the reuse group above. Otherwise, loop j still leads a partial reuse group level inside PE.
  Data blocks overlap with each other on the batch dimension of x and m. x and m together will give a magnifying factor $f_{t,i}^{xm}$ for groups in level i. The value of $f_{t,i}^{xm}$ depends on group level i.



(a) Loops



(b) Temporal data sharing inside PE



(c) Spatial data sharing across PEs

Fig. 12. Magnifying factor of partial reuse loop.

Figure 12 explains the choice of magnifying factor. Figure 12(a) gives the loop configuration that defines the temporal reuse group hierarchy inside each PE. Figure 12(b) shows the temporal data reuse pattern inside the first PE. There are two levels of partial reuse groups that are led by loop y and x. Data marked with the dot pattern are reused in the group level of x, while data marked in the strip pattern are reused in group level of y. Figure 12(c) combines data being reused in two group levels from each PE.

- For reuse groups formed by loop x or m, assuming they are in level i_x and level i_m, data blocks temporally overlap over tensor I's third dimension in each PE. As shown in Figure 12(c), data blocks from different PEs do not share the temporally used data within each PE. Therefore, $f^{xm}_{t,i_x} = f^{xm}_{t,i_m} = P^s_x \cdot P^s_m$.
- For reuse groups formed by other reuse or partial reuse loops, temporally reused data inside each PE can be spatially shared. As in Figure 12(c), each PE needs additional SRAM accesses for data with dimension B^{pe}_{xm}. With spatial sharing, the datapath only needs to access SRAM for data with dimension B^{dp}_{xm}. $f^{xm}_{t,i} = \frac{B^{dp}_{xm}}{B^{pe}_{xm}}$ for level i except i_x and level i_m.

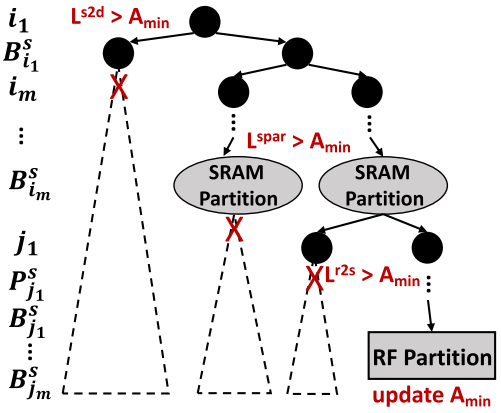


Fig. 13. Pruning integrated search tree with branch and bound.

6.2 Integrated Optimization

Problem 0 optimizes the weighted total of the number of DRAM accesses A^d and the number of SRAM accesses A^s. Besides parameters in $\mathcal{X}^s$ that affect A^d, A^s depends on parameters in $\mathcal{X}^r$, which includes the loop ordering $\boldsymbol{I}^s$, block dimensions $\mathcal{B}^r$, and step sizes $\mathcal{P}^s$ for PE blocking loops as well as the partitioning of the register file in each PE $\mathcal{M}^r$. Therefore, in this section, we explore $\mathcal{X}^s$ and $\mathcal{X}^r$ in an integrated way to optimize the objective function of **Problem 0**. Algorithm 2, used for SRAM partitioning, is updated to address the register file partitioning. The same BD pruning given in Section 5.2 can be applied to prune redundant sweeping for values of $\mathcal{B}^r$ and $\mathcal{P}^s$. The branch-and-bound approach introduced in Section 5.3 is also updated to prune the integrated search tree. As shown in Figure 10, the following two subsections provide updates to the memory partitioning and branch-and-bound pruning, respectively.

6.2.1 Register File Partitioning. We adopt the same memory partitioning in Algorithm 2 that is used for SRAM to partition the register file in each PE. It utilizes a function describing how the data spilling sensitivity changes versus local memory size change for each tensor t. We define the sensitivity function of SRAM accesses for tensor t as $F_t^{r,sen}(m)$ in Equation (20). As all PEs have the same register file partitioning, m in Equation (20) represents the register file size tensor t has in one PE. Nevertheless, A_t^s gives the total number of data spilling generated in all PEs from tensor t.

$$F_t^{r,sen}(m) = A_t^s(m) - A_t^s(m+1), 1 \le m \le M_{in}^r, m \in \mathbb{N} \tag{20}$$

Similar to $F_t^{d,sen}$ in Section 5, the sensitivity value of $F_t^{r,sen}$ in the range between M_{i+1}^{sf} and M_i^{sf} is determined by reuse groups in level i. For only one PE, $F_t^{r,sen}$ in this range is equal to $N_i^g \cdot N_i^c$, where N_i^g is the number of group instances in level i and N_i^c is the number of spilling cycles each level-i group has. With the effect of a PE batch, $F_t^{r,sen}$ is magnified by factor $f_{t,i}$, as given in Equation (21). $f_{t,i}$ gives the ratio between the number of data spilling in level-i groups generated from all PEs and from one PE. As introduced in Section 6.1, $f_{t,i}$ equals to the product of $f_{t,i}^j$, which represents the magnifying factor of loop j on data spilling from group level i.

$$F_t^{r,sen}(m)_{m \in [M_{i+1}^{sf}, M_i^{sf}]} = f_{t,i} \cdot N_i^g \cdot N_i^c \tag{21}$$

6.2.2 Integrated Branch and Bound. The integrated search tree is built up by sweeping values for one parameter in each layer. As sketched in Figure 13, from root to the leaves, the parameters

are ordered as follows: the loop indicator i_l for the lth SRAM blocking loop ($\boldsymbol{I}^d[l] = i_l$) and its blocking dimension $B^s_{i_l}$ for l from 1 to m, with SRAM partitioning done after $\boldsymbol{I}^d$ and $\mathcal{B}^s$ are fully specified, then followed by the loop indicator j_l for the lth PE blocking loop ($\boldsymbol{I}^s[l] = j_l$), its PE batch dimension $P^s_{j_l}$ and blocking dimension $B^r_{j_l}$ for l from 1 to m, with RF partitioning applied at the leaf node. The tree is also pruned using branch and bound. The lower bound estimation needs to be modified from what we used in solving **Problem 1**, since the objective function has changed. For nodes above the SRAM partitioning level, at the SRAM partitioning level, and below SRAM partitioning level, we evaluate their subtree lower bounds as L^{s2d}, L^{spar}, and L^{r2s}, respectively.

L^{s2d}: L^{s2d} is estimated as the weighted sum of DRAM accesses lower bound L^d and the compulsory SRAM accesses lower bound L^{sc}, as given in Equation (22). L^d is obtained with the algorithm given in Section 5. The compulsory SRAM accesses A^{sc}_t of tensor t is given in Equation (2). $N^{s,db}$, the number of SRAM Block Computation used in Equation (2), is computed as $\prod_{l \in \mathcal{I}_{in}} N^s_l$, where N^s_l is the number of iterations of SRAM blocking loop l. A^{sc}_t is lower bounded by the input tensor size S_t times the product of tensor t's reuse loop iterations, as shown in Equation (23). For partially defined loop blocking, having all unspecified loops are blocked with its input dimension can guarantee L^{sc} as the A^{sc} lower bound.

$$L^{s2d} = L^d + w \cdot L^{sc} \tag{22}$$

$$L^{sc} = \sum_t \left(S_t \cdot \prod_{t\text{'s reuse loop } l} N^s_l \right) \leq \prod_{l \in \mathcal{I}_{in}} N^s_l \cdot \sum_{t \in T_{in} \cup T_{out}} C^s_t = A^{sc} \tag{23}$$

L^{spar}: At the SRAM partition node where $\boldsymbol{I}^d$ and $\mathcal{B}^s$ are fully defined, the objective lower bound is estimated as L^{spar} given in Equation (24). The exact value of DRAM accesses and compulsory SRAM accesses can be obtained at this point. L^{spar} is estimated by simply assuming the number of additional SRAM accesses is zero.

$$L^{spar} = A^d + w \cdot A^{sc} \tag{24}$$

L^{r2s}: At nodes when not all parameters of the PE Block Computation are determined, the lower bound is estimated as L^{r2s}, as shown in Equation (25). Besides exact A^d and A^{sc}, it also estimates the lower bound of additional SRAM accesses L^{sa} based on the partial information $\boldsymbol{I}^s$, $\mathcal{B}^r$, and $\mathcal{P}^s$.

$$L^{r2s} = A^d + w \cdot (A^{sc} + L^{sa}) \tag{25}$$

L^{sa} is the lower bound number of additional SRAM accesses. As we obtained L^d in Section 5, L^{sa} is estimated by $\sum_t L^{sa}_t$, where L^{sa}_t is the lower bound of the number of addition SRAM accesses generated by tensor t. Tensor t can have the least number of data spilling if it can occupy the maximum possible size of the register file in PEs, represented as $M^{r,u}_t$, and the loops are configured in a way that benefits tensor t the most. Therefore, as shown in Figure 10, we use two steps to obtain the value of L^{sa}_t, with each step being addressed by one subsection below.

6.2.2.1 Estimate $M^{r,u}_t$ - Register File Size Upper Bound for Tensor t. To estimate the upper bound of register size for tensor t, as in estimating the SRAM size upper bound, we use an upper bound sensitivity function F^{r,sen_u}_t for tensor t and lower bound sensitivity function $F^{r,sen\_l}_{t'}$ for other tensors t'. Therefore, we maintain an upper bound sensitivity function F^{r,sen_u}_t and a lower bound sensitivity function F^{r,sen_l}_t for each tensor t. The algorithm for maintaining F^{r,sen_u}_t and F^{r,sen_l}_t is modified from that of maintaining F^{d,sen_u}_t and F^{d,sen_l}_t to include the effect of PE batch dimensions $\mathcal{P}^s$. $\mathcal{P}^s$ controls the size of data being reused inside each PE and the sensitivity value of data spillings from all PEs. As the sensitivity function takes the register size of one PE as its x-axis but the sensitivity value of data spillings from all PEs as its y-axis, $\mathcal{P}^s$ adjusts its ranges on both x-axis

and y-axis. For related loops and partial reuse loops, the step size greater than one reduces the tensor size being reused inside one PE but increases the number of PEs generating data spilling. The former is reflected on the x-axis, while the latter gets reflected on the y-axis.

Maintaining F_t^{r,sen_u}. For maintaining F_t^{r,sen_u}, we assume, for each unspecified loop i in the subtree, its block size B_i^r equals to its input dimension B_i^s. Compared to the actual quantities $B_i^r \cdot P_i^s = B_i^s$, this may overestimate the size of reuse groups in each PE. After the actual values of B_i^r and P_i^s are determined, we adjust the range of F_t^{r,sen_u} along the x-axis to obtain a tighter sensitivity bound. We use $B_i^{s,pe}$ to represent the total data dimension over the direction of loop i for all the data blocks being used in one PE. Its size depends on the actual value of B_i^r and P_i^s. With evaluating B_i^r as B_i^s, the reuse groups disclosed so far also evaluate $B_i^{s,pe}$ as B_i^s in maintaining F_t^{r,sen_u}. Therefore, after specifying B_i^r and P_i^s in each step, we then shrink F_t^{r,sen_u} along its x-axis by the ratio between B_i^s and $B_i^{s,pe}$. The value of $B_i^{s,pe}$ is evaluated as below.

- $B_i^{s,pe}$ of related loop: $B_i^{s,pe} = \lceil \frac{B_i^s}{P_i^s} \rceil$.
- $B_i^{s,pe}$ of partial reuse loop: B_i^{pe} is the dimension of data block in one PE. B_i^{rsp} and B_i^{rti} represent the dimension shared spatially and temporally, respectively. $B_i^{s,pe} = B_i^{pe} + (N_i^r - 1)(B_i^{pe} - B_i^{rti})$. N_i^r is the number of loop iterations, $N_i^r = \lceil \frac{B_i^s}{B_i^r \cdot P_i^s} \rceil$.

For the specified group level j that is above level i, as in Equation (21), its sensitivity value is estimated as $f_{t,j} \cdot N_j^g \cdot N_j^c$. We assume the PE batch magnifying factor $f_{t,j}$ for every group level j is equal to its maximum value limited by the number of physical PEs $D_1^{pe} \cdot D_2^{pe}$. The number of group instances N_j^g is determined by group levels above j, which is not affected by B_i^r and P_i^r of its lower group level i. B_i^r determines the size of subgroup being reused $S_j^{b,r}$ and the spill-free memory size of subgroup M_j^{sf}. Based on Equation (7), N_j^c does not change with B_i^r as well. Thus, after knowing the actual value of B_i^r and P_i^s, we do not need to adjust the magnitude of sensitivity values that have been updated.

Maintaining F_t^{r,sen_l}. For F_t^{r,sen_l}, setting both block dimension and step size to 1 guarantees the lower bound. For related loop and partial reuse loop, we need to correct the underestimation towards their input dimension in disclosed reuse groups. In contrast to the F_t^{r,sen_u} case, we now expand the function along the x-axis based on group size underestimation and increase the function value along the y-axis based on sensitivity value underestimation.

Lemma 6.1. *$F_t^{r,sen\_u}(m)$ and $F_t^{r,sen\_l}(m)$ give an upper and lower bound, respectively, for $F_t^{r,sen}(m)$ of tensor t. $M_t^{r,u}$ is the upper bound of memory size tensor t can have.*

Proof. Assume the partially defined $\boldsymbol{I}^s$, $\mathcal{B}^r$, and $\mathcal{P}^s$ specify a partial hierarchy of reuse groups with i levels. In the range $[M_{i+1}^{sf\_u}, M_i^{sf\_u}]$, the value of F_t^{r,sen_u} is updated by $f_{t,i} \cdot N_i^g \cdot N_i^c$. $f_{t,i}$ is the aggregated PE batch magnifying factor for level-i group. Similar to the proof for Lemma 5.4, we only need to show $M_i^{sf\_u}$ is the upper bound of M_i^{sf} and the sensitivity value we updated for level i is the upper bound of its real value. Then, together with non-decreasing property of the sensitivity function, we can prove $F_t^{r,sen\_u}(m)$ is the upper bound of $F_t^{r,sen}(m)$.

Estimating B_l^r as B_l^s for unknown loop l makes $M_i^{sf\_u}$ the upper bound value of its actual value. $f_{t,i} = D_1^{pe} \cdot D_2^{pe}$ is also the upper bound of $f_{t,i}$'s real value. Similar to the case of estimating F_t^{d,sen_u}, N_i^c uses $D_i = S(\mathcal{D}_{i+1}^{sf} \cap \mathcal{D}_i^{b,r})$. With $\mathcal{D}_{i+1}^{sf}$ being the superset of its actual dataset, N_i^c also gives the upper bound of its actual value. Therefore, the sensitivity value for level-i reuse groups is overestimated.

Similarly, $F_t^{r,sen\_l}(m)$ estimates the sensitivity value of group level i between $M_{i+1}^{sf\_l}$ and $M_i^{sf\_l}$. $M_i^{sf\_l}$ is the lower bound estimation of M_i^{sf}. As D_i is estimated with its lower bound, the calculation of F_t^{r,sen_l} underestimates the actual sensitivity value of the current group level. With proof similar to that used for F_t^{d,sen_l}, F_t^{r,sen_l} gives the lower bound of $F_t^{r,sen}$. □

6.2.2.2 Optimum Loop Configuration for Tensor t.

$$
\begin{aligned}
\boldsymbol{I}_t^{opt} &= [\mathcal{I}_c^{t\_related}, \mathcal{I}_c^{t\_partial}, \mathcal{I}_c^{t\_reuse}] \\
\mathcal{B}_t^{opt}[l] &= \begin{cases} 1 & \text{if } l \in \mathcal{I}_c^{t\_related} \cup \mathcal{I}_c^{t\_partial} \\ B_l^d & else \end{cases} \\
\mathcal{P}_t^{opt}[l] &= \begin{cases} 1 & \text{if } l = \mathcal{I}_c^{t\_partial} \cup \mathcal{I}_c^{t\_reuse} \\ max(D_1^{pe}, D_2^{pe}) & else \end{cases}
\end{aligned} \tag{26}
$$

For tensor t, $\boldsymbol{I}^s$ and $\mathcal{B}^r$ in the subtree define the remaining reuse group hierarchy inside each PE. $\mathcal{P}^s$ in the subtree determines the spilling magnifying factor caused by the PE batch. For any given $\mathcal{P}^s$, we can estimate L_t^{sa}, the lower bound number of data spilling from tensor t, by accumulating data spillings from level i^c and above, when loops in the subtree are configured as (1) loops in subtree are ordered as related loops above partial reuse loop and above reuse loop; (2) blocking dimensions of related loop and partial reuse loop set to 1 and the relative ordering of partial reuse loops is explored. Level i^c is the lowest group level led by specified reuse/partial reuse loops on the search path. To obtain L_t^{sa} with considering $\mathcal{P}^s$, for each partial loop ordering, we need to obtain the optimum $\mathcal{P}^s$ to minimize L_t^{sa}. In the following, we show how $\mathcal{P}^s$ could affect the lower bound estimation of L_t^{sa}.

PE batch dimension for L_t^{sa}. With related loops and partial reuse loops having block dimension as 1, if their batch dimension is 1, then their loop iteration number is equal to the SRAM block dimension. For loop j, having PE batch dimension $P_j^s > 1$ decreases the number of loop iterations N_j^r that may reduce data being temporally reused inside PE but increases PE batch magnifying factor f_t^j as multiple PEs generate data spilling on its batch dimension.

If loop j is related loop, then having $P_j^s > 1$ increases its magnifying factor f_t^j by P_j^s. Its number of loop iterations N_j^r is decreased by P_j^s. Loop j is located in reuse group level i^c, the lowest group level defined by specified loop on the search path. We investigate how the change of N_j^r affects data spilling generated in level i^c and its upper levels. For group level i^c, N_j^r controls its number of subgroups $N_{i^c}^b$. If the memory size is between $M_{i^c+1}^{sf}$ and $M_{i^c}^{sf}$, based on Equation (19), then data spilling in level i^c is reduced by more than P_j^s times if $N_{i^c}^b$ decreases P_j^s times. Therefore, in this case, level i^c benefits from having $P_j^s > 1$. If level i^c requires full data spilling, then its number of data spilling reduces by P_j^s times. In this case, having $P_j^s > 1$ does not affect spilling in level i^c. For the upper group level $i < i^c$, N_j^r controls its subgroup size $S_i^{b,r}$. Its number of data spilling is decreased by more than P_j^s times if it is the lowest group level requiring data spilling. If it requires full spilling, then its number of data spilling will then decrease by just P_j^s times. In summary, for any loop configuration, having $P_j^s > 1$ if loop j is a related loop could give smaller value of L_t^{sa}.

If loop j is partial reuse loop, then having $P_j^s > 1$ magnifies the number of data spilling generated in level i^c and above by B_j^{dp}/B_j^{pe} times. Having $P_j^s > 1$ decreases the number of loop iterations N_j^r by P_j^s times. For groups in level $i \le i^c$, N_j^r controls the subgroup size $S_i^{b,r}$ through adjusting $B_j^{s,pe}$, the effective input dimension for all data used in one PE. As mentioned before, $B_j^{s,pe} = B_j^{pe} + (N_j^r - 1)(B_j^{pe} - B_j^{rti})$. $B_j^{rti} = max(0, B_j^{pe} - P_j^s(B_j^{pe} - B_j^{rsp}))$. We could prove the effective input

Table 1. Memory Accesses in AlexNet

Layer	Tensor size (MB)	#MAC ops (G)	Total DRAM acc			DRAM acc due to spilling			Total SRAM acc			Opt time (mins)
			Eyeriss (MB)	CNNFlow (MB)	↓ %	Eyeriss (MB)	CNNFlow (MB)	↓ %	Eyeriss (MB)	CNNFlow (MB)	↓ %	
Conv1	3.58	0.42	6	3.58	40	2.42	0.00	100	18.5	16.48	11	16.8
Conv2	2.36	0.90	5.4	4.13	24	3.04	1.77	42	77.6	29.35	62	7.6
Conv3	2.62	0.60	5	4.78	4	2.38	2.16	9	50.2	37.69	25	4.7
Conv4	2.09	0.45	3.9	3.69	5	1.81	1.60	12	37.4	28.27	24	4.8
Conv5	1.50	0.30	2.6	2.48	5	1.10	0.98	11	24.9	18.85	24	3.5
Total	12.15	2.67	22.9	18.66	19	10.75	6.51	40	208.6	130.64	37	-

↓ Represents Reduction.

dimension reduces less than B_j^{dp}/B_j^{pe} times by having $P_j^s > 1$. Therefore, the optimum value of P_j^s to obtain the minimum L_t^{sa} is equal to 1 if loop j is a partial reuse loop.

7 RESULTS

We have implemented the CNNFlow framework in C++. The framework contains (i) the tool for analytically calculating the memory accesses and (ii) an optimizer incorporating the pruning techniques.

We demonstrate the efficacy of our methods through comparing with other state-of-the-art accelerator designs and tools. We choose AlexNet [50] and VGG-16 [51] as benchmarks, as they are two popular CNN architectures with results reported by other works. They have 5 and 13 convolution layers, respectively, so, effectively, we are running CNNFlow on 18 test cases. To provide a fair comparison, we set the hardware resources in CNNFlow, such as on-chip SRAM size, PE array dimensions, and register file size, to be the same, as they are in the competing works.

7.1 AlexNet

We compare the solution generated by CNNFlow with Eyeriss[1] for AlexNet. Eyeriss has a 108 KB global buffer, 12×14 PE array, and a scratchpad with size 24 B, 448 B, and 48 B for Ifmap, filter, and partial sum in each PE. The data bit width is 16 b. The memory settings of CNNFlow are 108 KB on-chip SRAM organized in 27 banks with 4 KB for each bank, 12×14 PE array, and 520 B register file in each PE. With 16-bit data width, each SRAM bank can hold 2k data and each register file can hold 260 data. Table 1 shows the results of the convolution layers in AlexNet given by Eyeriss and CNNFlow. As in Eyeriss, the number of batches in CNNFlow, B_f^d, is set to 4. In total, we reduce about 19% DRAM accesses and 37% SRAM accesses. Subtracting the compulsory DRAM accesses that equal the input tensor size, CNNFlow can reduce 40% DRAM accesses caused by data spills.

The optimization for the layers is run in parallel, with each layer optimized with a single core. The max optimization time CNNFlow used is about 17 minutes.[1]

7.2 VGG16

We also compare VGG-16 with Eyeriss, as shown in Table 2. The hardware setting is the same as in the AlexNet experiment but the number of batches is 3, as used in Eyeriss. The solutions given by CNNFlow can reduce 86% SRAM accesses compared to Eyeriss. In Eyeriss, data are stored using a sparse representation in DRAM and are only decoded into full length before they are loaded into on-chip memory. Thus, the DRAM accesses they report in MB are with compressed data size. We focus on dataflow optimization and do not consider sparsity in this work. The DRAM accesses

[1]CNNFlow is run on a server with Intel Xeon CPU E5-2643 v3 at 3.40 GHz.

Table 2. Memory Accesses in VGG16

Layer	Tensor size (MB)	#MAC ops (G)	Total DRAM acc		Total SRAM acc			Opt time (mins)
			Eyeriss (MB) /Ifmap zero%	CNNFlow (MB)	Eyeriss (MB)	CNNFlow (MB)	↓ %	
Conv11	19.25	0.26	15.4/1.6	19.25	112.6	38.54	66%	3.3
Conv12	37.14	5.55	54.0/47.4	37.99	2,402.8	135.35	94%	10.1
Conv21	14.08	2.77	33.4/24.8	19.19	1,201.4	85.87	93%	7.4
Conv22	18.98	5.55	48.5/38.7	47.55	2,402.8	171.75	93%	18.8
Conv31	7.62	2.77	20.2/39.7	24.17	607.4	78.75	87%	12.6
Conv32	10.64	5.55	32.2/58.1	45.47	1,214.8	209.50	83%	53.8
Conv33	10.64	5.55	30.8/58.7	45.57	1,214.8	209.50	83%	53.8
Conv41	5.86	2.77	17.8/64.3	24.52	321.8	91.14	72%	22.0
Conv42	9.43	5.55	28.6/74.7	46.91	643.7	186.08	71%	45.4
Conv43	9.43	5.55	22.8/85.4	46.91	643.7	186.08	71%	46.3
Conv51	5.82	1.39	6.3/79.4	11.81	90	50.30	44%	7.71
Conv52	5.82	1.39	5.7/87.4	11.81	90	50.30	44%	7.71
Conv53	5.82	1.39	5.6/88.5	11.81	90	50.30	44%	7.71
Total	160.53	46.04	-	392.96	11,035.80	1,543.46	86%	-

↓ Represents Reduction.

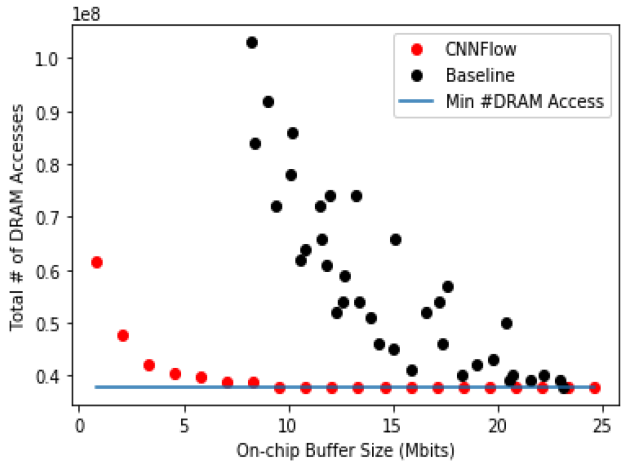


Fig. 14. DRAM accesses in VGG-16.

CNNFlow reports assumes each data is in its full length, i.e., 16 bits. Given this difference, we cannot fairly compare the decrease in the percentage of DRAM accesses but, overall, we still do better even considering the compression due to the sparse representation in Eyeriss.

For VGG-16, we also provide a comparison with the results in Reference [29], which also optimizes loop unrolling, tiling, and interchanging for reducing memory accesses. However, they couple the block dimension with the on-chip buffer size and only randomly sample the design space for optimization.

Figure 6 in Reference [29] shows how its total number of DRAM accesses changes with the on-chip buffer size. In Figure 14, we plot DRAM access points in Reference [29]'s Figure 6 as black dots. We also show the solution generated by CNNFlow as red dots. The blue line at the bottom is the

sum of tensor size, which is the minimum number of DRAM accesses that can be achieved. Overall, CNNFlow requires fewer DRAM accesses for a given SRAM size than Reference [29]. Further, CNNFlow realizes the minimum DRAM accesses possible with 10 Mbits SRAM, while the design in Reference [29] requires about 20 Mbits SRAM.

The results shown above demonstrate that the exact optimal solutions found by CNNFlow have fewer DRAM/SRAM accesses than competing designs where the results are obtained using heuristic methods and/or sampling. Further, they show that CNNFlow's optimal algorithm runs in reasonable time for practical instances.

8 CONCLUSION

This work addresses the problem of automatically mapping the CNN computation on a CNN architecture such that remote memory access is minimized and data reuse in local memory is maximized. This is done through optimally controlling the dataflow through hierarchical data blocking, loop ordering, as well as local memory partitioning for the CNN tensors. We provide a precise mathematical formulation for optimizing the memory access in terms of these parameters. We then provide a solution to this problem that has two key contributions: (1) it uses an analytical method to calculate the number of DRAM and SRAM accesses for a given value of the above parameters; (2) it prunes the search space of these parameters using a set of techniques to make the solution search manageable. We demonstrate the efficacy of this exact solution method in both computational efficiency and quality of results. It provides exact solutions in reasonable compute time (tens of mins). In comparison with the state-of-the-art methods for CNN mapping, CNNFlow can realize 20% fewer DRAM accesses and 40%–80% fewer SRAM accesses for well-known CNN applications.

REFERENCES

[1] Yu-Hsin Chen, Tushar Krishna, Joel S. Emer, and Vivienne Sze. 2016. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE J. Solid-state Circ.* 52, 1 (2016).

[2] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. 2014. DaDianNao: A machine-learning supercomputer. In *MICRO*. 609–622.

[3] Song Han, Huizi Mao, and William J. Dally. 2015. Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding. *arXiv preprint arXiv:1510.00149* (2015).

[4] Jiantao Qiu, Jie Wang, Song Yao, Kaiyuan Guo, Boxun Li, Erjin Zhou, Jincheng Yu, Tianqi Tang, Ningyi Xu, Sen Song, Yu Wang, and Huazhong Yang. 2016. Going deeper with embedded FPGA platform for convolutional neural network. In *FPGA*. 26–35.

[5] Ping Chi, Shuangchen Li, Cong Xu, Tao Zhang, Jishen Zhao, Yongpan Liu, Yu Wang, and Yuan Xie. 2016. PRIME: A novel processing-in-memory architecture for neural network computation in ReRAM-based main memory. In *ACM SIGARCH Computer Architecture News*, Vol. 44. IEEE, 27–39.

[6] Google. 2022. XLA: Optimizing compiler for tensorflow. Retrieved from https://www.tensorflow.org/performance/xla.

[7] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Garret Catron, Summer Deng, Roman Dzhabarov, Nick Gibson, James Hegeman, Meghan Lele, Roman Levenstein, Jack Montgomery, Bert Maher, Satish Nadathur, Jakob Olesen, Jongsoo Park, Artem Rakhov, Misha Smelyanskiy, and Man Wang. 2018. Glow: Graph lowering compiler techniques for neural networks. *arXiv preprint arXiv:1805.00907* (2018).

[8] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An automated end-to-end optimizing compiler for deep learning. In *OSDI*. 578–594.

[9] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. 2014. OpenTuner: An extensible framework for program autotuning. In *PACT*. 303–316.

[10] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. 2016. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *ACM SIGARCH Computer Architecture News*, Vol. 44. IEEE.

[11] Hardik Sharma, Jongse Park, Divya Mahajan, Emmanuel Amaro, Joon Kyung Kim, Chenkai Shao, Asit Mishra, and Hadi Esmaeilzadeh. 2016. From high-level deep neural models to FPGAs. In *MICRO*. 1–12.

[12] Zidong Du, Robert Fasthuber, Tianshi Chen, Paolo Ienne, Ling Li, Tao Luo, Xiaobing Feng, Yunji Chen, and Olivier Temam. 2015. ShiDianNao: Shifting vision processing closer to the sensor. In *ISCA*.

[13] Abhinandan Majumdar, Srihari Cadambi, Michela Becchi, Srimat T. Chakradhar, and Hans Peter Graf. 2012. A massively parallel, energy efficient programmable accelerator for learning and classification. *ACM Trans. Archit. Code Optim.* 9, 1 (2012).
[14] Utku Aydonat, Shane O'Connell, Davor Capalija, Andrew C. Ling, and Gordon R. Chiu. 2017. An OpenCL™deep learning accelerator on Arria 10. In *FPGA*. 55–64.
[15] Xuechao Wei, Cody Hao Yu, Peng Zhang, Youxiang Chen, Yuxin Wang, Han Hu, Yun Liang, and Jason Cong. 2017. Automated systolic array architecture synthesis for high throughput CNN inference on FPGAs. In *DAC*.
[16] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W. Keckler, and William J. Dally. 2017. SCNN: An accelerator for compressed-sparse convolutional neural networks. In *ISCA*. 27–40.
[17] Yufei Ma, Yu Cao, Sarma Vrudhula, and Jae-sun Seo. 2017. An automatic RTL compiler for high-throughput FPGA implementation of diverse deep convolutional neural networks. In *FPL*. 1–8.
[18] Lukas Cavigelli and Luca Benini. 2016. Origami: A 803-GOp/s/W convolutional network accelerator. *IEEE Trans. Circ. Syst. Vid. Technol.* 27, 11 (2016), 2461–2475.
[19] Yu Ting Chen and Jason H. Anderson. 2017. Automated generation of banked memory architectures in the high-level synthesis of multi-threaded software. In *FPL*. 1–8.
[20] Xiaofan Zhang, Junsong Wang, Chao Zhu, Yonghua Lin, Jinjun Xiong, Wen-mei Hwu, and Deming Chen. 2018. DNNBuilder: An automated tool for building high-performance DNN hardware accelerators for FPGAs. In *ICCAD*.
[21] Yufei Ma, Naveen Suda, Yu Cao, Jae-sun Seo, and Sarma Vrudhula. 2016. Scalable and modularized RTL compilation of convolutional neural networks onto FPGA. In *FPL*. 1–8.
[22] Huimin Li, Xitian Fan, Li Jiao, Wei Cao, Xuegong Zhou, and Lingli Wang. 2016. A high performance FPGA-based accelerator for large-scale convolutional neural networks. In *FPL*. 1–9.
[23] Murugan Sankaradas, Venkata Jakkula, Srihari Cadambi, Srimat Chakradhar, Igor Durdanovic, Eric Cosatto, and Hans Peter Graf. 2009. A massively parallel coprocessor for convolutional neural networks. In *ASAP*. 53–60.
[24] Srihari Cadambi, Abhinandan Majumdar, Michela Becchi, Srimat Chakradhar, and Hans Peter Graf. 2010. A programmable parallel accelerator for learning and classification. In *PACT*. 273–284.
[25] Srimat Chakradhar, Murugan Sankaradas, Venkata Jakkula, and Srihari Cadambi. 2010. A dynamically configurable coprocessor for convolutional neural networks. *ACM SIGARCH Comput. Archit. News* 38, 3 (2010), 247–257.
[26] Mohammad Motamedi, Philipp Gysel, Venkatesh Akella, and Soheil Ghiasi. 2016. Design space exploration of FPGA-based deep convolutional neural networks. In *ASP-DAC*. 575–580.
[27] Joo-Young Kim, Minsu Kim, Seungjin Lee, Jinwook Oh, Kwanho Kim, and Hoi-Jun Yoo. 2009. A 201.4 GOPS 496 mW real-time multi-object recognition processor with bio-inspired neural perception engine. *IEEE J. Solid-state Circ.* 45, 1 (2009), 32–45.
[28] Maurice Peemen, Arnaud A. A. Setio, Bart Mesman, and Henk Corporaal. 2013. Memory-centric accelerator design for convolutional neural networks. In *ICCD*. 13–19.
[29] Yufei Ma, Yu Cao, Sarma Vrudhula, and Jae-sun Seo. 2017. Optimizing loop operation and dataflow in FPGA acceleration of deep convolutional neural networks. In *FPGA*.
[30] Wenyan Lu, Guihai Yan, Jiajun Li, Shijun Gong, Yinhe Han, and Xiaowei Li. 2017. FlexFlow: A flexible dataflow accelerator architecture for convolutional neural networks. In *HPCA*. 553–564. DOI: http://dx.doi.org/10.1109/HPCA.2017.29
[31] Qi Nie and Sharad Malik. 2018. MemFlow: Memory-driven data scheduling with datapath co-design in accelerators for large-scale inference applications. In *ASP-DAC*.
[32] Q. Nie and S. Malik. 2020. MemFlow: Memory-driven data scheduling with datapath co-design in accelerators for large-scale inference applications. *IEEE Trans. Comput.-Aid. Des. Integ. Circ. Syst.* 39, 9 (2020), 1875–1888. DOI: 10.1109/TCAD.2019.2925377
[33] Hanqing Zeng, Ren Chen, Chi Zhang, and Viktor Prasanna. 2018. A framework for generating high throughput CNN implementations on FPGAs. In *FPGA*. 117–126.
[34] Stylianos I. Venieris and Christos-Savvas Bouganis. 2016. fpgaConvNet: A framework for mapping convolutional neural networks on FPGAs. In *FCCM*. 40–47.
[35] Naveen Suda, Vikas Chandra, Ganesh Dasika, Abinash Mohanty, Yufei Ma, Sarma Vrudhula, Jae-sun Seo, and Yu Cao. 2016. Throughput-optimized OpenCL-based FPGA accelerator for large-scale convolutional neural networks. In *FPGA*. 16–25.
[36] Jialiang Zhang and Jing Li. 2017. Improving the performance of OpenCL-based FPGA accelerator for convolutional neural network. In *FPGA*. ACM, 25–34.
[37] Roberto DiCecco, Griffin Lacey, Jasmina Vasiljevic, Paul Chow, Graham Taylor, and Shawki Areibi. 2016. Caffeinated FPGAs: FPGA framework for convolutional neural networks. In *FPT*.
[38] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. 2015. Optimizing FPGA-based accelerator design for deep convolutional neural networks. In *FPGA*.

[39] Chen Zhang, Guangyu Sun, Zhenman Fang, Peipei Zhou, Peichen Pan, and Jason Cong. 2018. Caffeine: Towards uniformed representation and acceleration for deep convolutional neural networks. In *ICCAD*.
[40] Ying Wang, Jie Xu, Yinhe Han, Huawei Li, and Xiaowei Li. 2016. DeepBurning: Automatic generation of FPGA-based learning accelerators for the neural network family. In *DAC*.
[41] Kaiyuan Guo, Lingzhi Sui, Jiantao Qiu, Jincheng Yu, Junbin Wang, Song Yao, Song Han, Yu Wang, and Huazhong Yang. 2017. Angel-Eye: A complete design flow for mapping CNN onto embedded FPGA. *IEEE Trans. Comput.-aid. Des. Integ. Circ. Syst.* 37, 1 (2017).
[42] 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. 2019. Timeloop: A systematic approach to DNN accelerator evaluation. In *ISPASS*.
[43] Liqiang Lu, Naiqing Guan, Yuyue Wang, Liancheng Jia, Zizhang Luo, Jieming Yin, Jason Cong, and Yun Liang. 2021. TENET: A framework for modeling tensor dataflow based on relation-centric notation. In *ISCA*. 720–733. DOI: http://dx.doi.org/10.1109/ISCA52012.2021.00062
[44] Jiajun Li, Guihai Yan, Wenyan Lu, Shuhao Jiang, Shijun Gong, Jingya Wu, and Xiaowei Li. 2018. SmartShuttle: Optimizing off-chip memory accesses for deep learning accelerators. In *DATE*. 343–348. DOI: http://dx.doi.org/10.23919/DATE.2018.8342033
[45] Qijing Huang, Minwoo Kang, Grace Dinh, Thomas Norell, Aravind Kalaiah, James Demmel, John Wawrzynek, and Yakun Sophia Shao. 2021. CoSA: Scheduling by constrained optimization for spatial accelerators. In *ISCA*. 554–566. DOI: http://dx.doi.org/10.1109/ISCA52012.2021.00050
[46] Sheng-Chun Kao and Tushar Krishna. 2020. GAMMA: Automating the HW mapping of DNN models on accelerators via genetic algorithm. In *ICCAD*. 1–9.
[47] Fengbin Tu, Weiwei Wu, Shouyi Yin, Leibo Liu, and Shaojun Wei. 2018. RANA: Towards efficient neural acceleration with refresh-optimized embedded DRAM. In *ISCA*.
[48] Liu Ke, Xin He, and Xuan Zhang. 2018. NNest: Early-stage design space exploration tool for neural network inference accelerators. In *ISLPED*. ACM, 4:1–4:6.
[49] Laszlo A. Belady. 1966. A study of replacement algorithms for a virtual-storage computer. *IBM Syst. J.* 5, 2 (1966), 78–101.
[50] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. ImageNet classification with deep convolutional neural networks. In *NIPS*. 1097–1105.
[51] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. In *arXiv preprint arXiv:1409.1556*.

Received 2 February 2022; revised 26 November 2022; accepted 7 December 2022