I/O Lower Bounds for Auto-tuning of Convolutions in CNNs

Xiaoyang Zhang, Junmin Xiao*, and Guangming Tan
State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences
University of Chinese Academy of Science
zhangxiaoyang@ncic.ac.cn xiaojunmin@ict.ac.cn tgm@ict.ac.cn

Abstract

Convolution is the most time-consuming part in the computation of convolutional neural networks (CNNs), which have achieved great successes in numerous practical applications. Due to the complex data dependency and the increase in the amount of model samples, the convolution suffers from high overhead on data movement (i.e., memory access). This work provides comprehensive analysis and methodologies to minimize the communication for the convolution in CNNs. With an in-depth analysis of the recent I/O complexity theory under the red-blue game model, we develop a general I/O lower bound theory for a composite algorithm which consists of several different sub-computations. Based on the proposed theory, we establish the data movement lower bound results for two main convolution algorithms in CNNs, namely the direct convolution and Winograd algorithm, which represents the direct and indirect implementations of a convolution respectively. Next, derived from I/O lower bound results, we design the near I/O-optimal dataflow strategies for the two main convolution algorithms by fully exploiting the data reuse. Furthermore, in order to push the envelope of performance of the near I/O-optimal dataflow strategies further, an aggressive design of auto-tuning based on I/O lower bounds, is proposed to search an optimal parameter configuration for the direct convolution and Winograd algorithm on GPU, such as the number of threads and the size of shared memory used in each thread block. Finally, experiment evaluation results on the direct convolution and Winograd algorithm show that our dataflow strategies with the auto-tuning approach can achieve about 3.32× performance speedup on average over cuDNN. In addition, compared with TVM, which represents the state-of-the-art technique for auto-tuning, not only our auto-tuning method based on I/O lower bounds can find the optimal parameter configuration faster, but also our solution has higher performance than the optimal solution provided by TVM.

CCS Concepts: • **Theory of computation → Communication complexity**; • **Computing methodologies → Parallel algorithms**.

Keywords: I/O lower bounds, red-blue pebble game, dataflow design, auto-tuning, convolutional neural network.

1 Introduction

Convolutional neural networks (CNNs) are commonly applied to numerous computer vision and machine learning applications, which have achieved great successes because the complex layer structures could produce high-quality results based on a large number of data. Specifically, the convolution layer is an important structure in many state-of-the-art modern CNN models, such as MobileNet [14], ResNet [28], ShuffleNet [33], SqueezeNet [15], VggNet[26] and so on. The wide adoption of convolution and its huge cost have led to a high demand to optimize convolution operations for high performance. From the hardware perspective, GPUs have been demonstrated to be able to provide tremendous computation power for accelerating convolution operations. Furthermore, many specific accelerators for convolutions in CNNs are designed based on field-programmable gate arrays (FPGA) and application-specific integrated circuits (ASIC). From the software perspective, a variety of optimization techniques have been developed from algorithm level [8] to compilation level [34]. Many optimization efforts have also been incorporated into the widely used software libraries, such as NVIDIA cuDNN [9] and AMD MIOpen [19].

For convolution operations in CNNs, multiple convolution algorithms have been developed and classified into two categories: direct and indirect approaches. Typical direct and indirect representatives are the direct convolution and Winograd convolution algorithms respectively, each of which involves a large amount of memory accesses due to the complex computational workflow and massive data in convolution operations. For example, all inputs and weights are typically stored in the off-chip memory of CNN accelerators, such as global memory in GPUs. During computation, partial inputs and weights are loaded from the off-chip memory into the

* Corresponding author, who established all theoretical results of this work.

PPoPP '21, February 27-March 3, 2021, Virtual Event, Republic of Korea

ACM ISBN 978-1-4503-8294-6/21/02…$15.00
https://doi.org/10.1145/3437801.3441609

on-chip buffer to produce portions of outputs. Meanwhile, each processor could use its own registers to read some inputs and weights which have been in the on-chip buffer. Consequently, the frequent data movement in the memory hierarchy commonly dominates the energy consumption of convolution operations [6]. Therefore, optimizing the data transmission of convolutions is the key to improve the performance of convolutions.

To minimize data movement, the most works focus on how to reduce the model size, such as quantifying weights [35]. On the other hand, another effective way for reducing communication is to increase data reuse based on the dataflow design. In recent years, a variety of dataflow approaches have been proposed [7, 18, 24], most of which mainly focus on the computation efficiency. However, the data movement of convolutions has not been taken a full account. This work would try to consider the communication-optimal strategies for different convolution algorithms based on the I/O lower bound analysis.

Since I/O lower bound analysis is important for evaluating the optimality of a proposed algorithmic solution, it is widely concerned to establish appropriate lower bounds of the data movement of application codes [29, 30]. Under the red-blue pebble game model [17] for data transmission in memory hierarchy, past work on I/O lower bounds has found bounds for specific algorithms, such as matrix-matrix multiplication and FFT. As the recent methodology mainly focuses on the workflow's specific properties which do not translate across different computational patterns, the recent lower bound theory seems hard to be applied to arbitrary computations such as convolutions, in which different sub-computations involve different computational patterns. How to establish a systematic I/O lower bound theory for convolutions based on the red-blue pebble game model is a big challenge [31]. Even if the lower bounds could be obtained, the theoretical minimum of I/O complexity is not easy to directly yield an efficient dataflow strategy. There is a very large space to explore. How to determine the dataflow with the help of I/O lower bound is another challenge.

To solve the above challenges, this work considers to quantify the contribution of each sub-computation to the total computation, and then generalizes the recent I/O lower bound theory to establish I/O lower bound results for convolutions under the red-blue pebble game model. Next, through a deeper investigation of the highest order term in the lower bound results, we determine which data reuse should be fully exploited, and propose the near I/O-optimal dataflow strategy for maximizing such data reuse to minimize the memory access in convolutions. Furthermore, by comparing the lower bound result with I/O cost of our dataflow strategy, the optimality condition for implementation of convolutions is deduced. Based on the optimality condition, a fine-grained auto-tuning optimization is designed to find the optimal implementation with high performance effectively.

In this work, we make the following key contributions.

- Develop a general I/O lower bound theory under the red-blue pebble game model for any composite algorithm which involves different sub-computations and different computational patterns.
- Establish I/O lower bound results for two typical representatives of direct and indirect convolution algorithms, which are the direct convolution and Winograd convolution algorithms.
- Design near I/O-optimal dataflow strategies respectively for the direct convolution and Winograd convolution algorithms.
- Propose an auto-tuning engine to achieve excellent implementations of our dataflow strategies.

2 Background

2.1 Red-blue Pebble Game

The red-blue pebble game is a two-level memory access model which is proposed by Hong & Kung [17]. The game is played on a directed acyclic graph (DAG), which describes the operation of the algorithm. Let $G(V, E)$ be a DAG. V is the vertex set representing operations of algorithm, and E is the edge set representing the dependency of two operations. A partition on G is called an S-partition if the following four properties hold.

- Property 1: V is partitioned into h subsets $V_1, V_2, \cdots, V_h$ such that V_i's are disjoint but their union is V.
- Property 2: There is a dominator set D_i for each V_i that contains at most S vertices. A dominator set D_i for V_i is a set of nodes in V such that any path from an input of G to a node in V_i contains some nodes in D_i.
- Property 3: There is a minimum set M_i for each V_i that contains at most S vertices. The minimum set of V_i is defined to be the set of vertices in V_i that do not have any successor vertex belonging to V_i.
- Property 4: No cyclic dependence is among $V_1, \cdots, V_h$.

Let $P(S)$ be the minimum number of subsets that any S-partition of a DAG must have. The following theorem describes the communication lower bound based on the S-partition model (the proof is provided in [17]).

Theorem 2.1. *Any complete calculation of a red-blue pebble game on DAG $G = (V, E)$ with at most S red pebbles needs the minimum I/O time Q such that*

$$Q \geq S \cdot (P(2S) - 1). \tag{1}$$

2.2 Direct Convolution

Figure 1 illustrates a direct convolution. We have an input image of size $W_{in} \times H_{in} \times C_{in}$ and C_{out} kernels of weights, producing a $W_{out} \times H_{out} \times C_{out}$ output image. For the convolution, the channels of input image C_{in} are the number of channels in each kernel, and the channels of output image C_{out} are equal to the number of kernels, and each channel

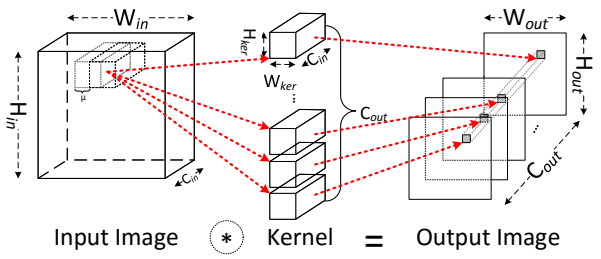

Figure 1. Direct Convolution.

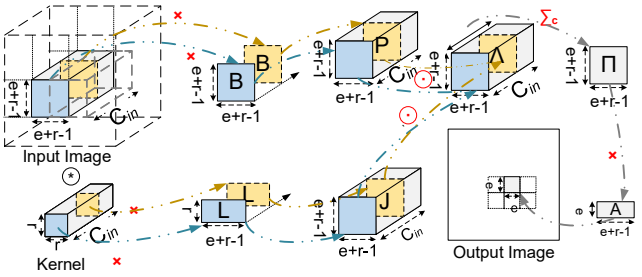

Figure 2. Winograd Algorithm.

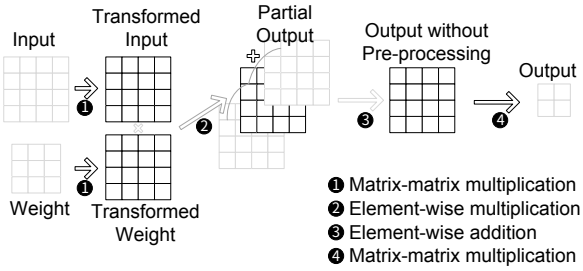

Figure 3. Different Patterns in Winograd Algorithm.

of output image is a $H_{out} \times W_{out}$ matrix. The kernel is a $W_{ker} \times H_{ker} \times C_{in}$ tensor. Each output is computed by an inner product between a kernel tensor and a sliding input tensor with the size of $W_{ker} \times H_{ker} \times C_{in}$ from an input image by using a sliding window. The stride size μ is the position difference between two adjacent sliding windows.

2.3 Winograd Algorithm

Winograd algorithm for convolution is shown in Figure 2. This algorithm changes the characteristics of time-domain convolution calculations, and reduces the number of multiplication operations between input images and kernels through mathematical transformation. In order to perform the mathematical transformation, several parameter matrices are introduced. Matrix A, B and L are three transformation matrices for output images, input images and kernels respectively. Furthermore, as Winograd algorithm requires $W_{ker} = H_{ker}$, we denote r as W_{ker} or H_{ker} briefly. Winograd algorithm can calculate multiple output results at once. Here, we denote $F(e \times e, r \times r)$ as a calculation process to deduce e^2 outputs in Winograd algorithm. Theoretically, the value of e is arbitrary, but in practice e usually is chosen as 2, 3 or 4. To compute every e^2 outputs at a fixed channel of an output image, $F(e \times e, r \times r)$ requires a sliding input tensor with the size of $(e+r-1) \times (e+r-1) \times C_{in}$ from input images using a sliding window and a kernel tensor with the size of $e \times e \times C_{in}$. Then the input tensor and kernel are transformed by B and L into P and J which have the same size of $(e+r-1) \times (e+r-1) \times C_{in}$. Next, the corresponding element product of P and J results in a new $(e+r-1) \times (e+r-1) \times C_{in}$ tensor Λ, and the summation of elements in Λ along channel direction generates the $(e+r-1) \times (e+r-1)$ matrix Π. Finally, Π is transformed by A into some $e \times e$ matrix which are e^2 outputs.

3 Challenges

In this section, we elaborate specific challenges that need to be addressed in order to build I/O lower bound theory and design I/O-optimal dataflow strategies, and present our basic idea to address these challenges.

3.1 Challenges for Building I/O Lower Bounds

In real application, most numerical algorithms, such as convolutions, are typically constructed from a number of sub-computations. For instance, Figure 3 shows that Winograd algorithm has 4 sub-computations, which involve 4 different patterns: (1) matrix-matrix multiplication, (2) element-wise multiplication, (3) element-wise addition, (4) matrix-matrix multiplication. Although the red-blue pebble game model has been proposed for many years, it is still difficult to use this model to establish I/O lower bounds of composite algorithms which involve several different kinds of computational patterns [13]. It is not even possible to deduce a suitable I/O lower bound of the DAG only focusing on each sub-computation of the composite algorithms, due to the following two main reasons. Firstly, at the beginning of the red-blue pebble game, all DAG vertices without predecessors have blue pebbles, and all vertices without successors would get blue pebbles at the end of the game. Based on this assumption, the calculation for each sub-DAG will require at least one load operation for each input and one store operation for each output. However, when the red-blue pebble game is played on the full DAG, the data could pass from a previous sub-computation to a later one directly through fast memory. Secondly, when a composite computation is assigned into several sub-computations, the total DAG is partitioned into several relevant sub-graphs. Under a common constraint that previous sub-computation must be totally finished before the later sub-computation starts, the partition way of the total DAG usually impacts the data movement complexity due to the limited size of fast memory. The two reasons above

describe the essential difficulties to develop the general I/O lower bound theory for any composite algorithm. To get around these difficulties, the red-blue-white pebble game model has been proposed recently to analyze composite algorithms, which uses some restrictions on models, such as the limitation of disallowing re-computation of values on vertices of DAG [13]. However, the convolution algorithms, such as Winograd algorithm, allow re-computation to reduce the number of I/O operations.

3.2 Challenges for Optimal Implementations

If I/O lower bounds could be obtained, another challenge is how to design the optimal implementations for convolutions. First of all, the theoretical minimum of I/O complexity seems not to directly yield efficient implementations. Although the dataflow design of increasing the local data reuse is useful for reducing the number of I/O operations, the recent lower bound theory could not tell us which data should be reused in the on-chip memory prior to the others. Hence, it is necessary to develop a new methodology for using insights from I/O lower bounds to design I/O-optimal strategies. Next, even if we know which data has higher reuse priority, it is also not easy to determine the optimal implementation for maximizing such data reuse. For the implementation design of convolutions, the combinatorial choices of memory access, threading pattern, specific input shape and layout create a huge configuration space, such as loop tiling, ordering, unrolling, and so on. For 4 sub-computations in Winograd algorithm, the size configuration space is usually larger than 10^6. This fact indicates that it is hard to manually design an efficient implementation for a convolution. Although NVIDIA proposes excellent implementations for different convolution algorithms in cuDNN library [10], these implementations mainly focus on general optimization on GPUs. Directly using the convolution API in cuDNN sometimes can not satisfy the real-time demand of inference applications. Recently, auto-tuning methods have been proposed for the fine-grained optimization of convolutions. The common way is to adopt a predefined cost model to guide the search, but building an accurate cost model is difficult due to the increasing complexity of modern hardware. As the state-of-the-art framework for auto-tuning convolutions, TVM proposes a new auto-tuning method based on ML-model [5]. However, it still needs a large search cost due to the huge search space.

3.3 Motivation

In this work, we explore the red-blue pebble game. The analysis on each sub-computation could not accurately estimate the data movement complexity, which is because the contribution of each sub-computation to total computation is ignored. Through the quantification of such contribution, all sub-computations can be viewed as a whole, which provides an opportunity to build I/O lower bound of composite algorithms.

Besides, to addresses the challenges for determining the optimal implementations, this work combines both coarse-grained design and fine-grained optimization. In the coarse-grained design, we develop a methodology of using the highest order term in I/O lower bounds to determine which data reuse should be fully exploited, and design I/O-optimal dataflow strategies for maximizing such data reuse to minimize the memory access in convolutions. By comparing the I/O volumes of the dataflow strategies with the lower bounds, we discover the optimality condition for the I/O-optimal design. In the fine-grained optimization, we use this optimality condition to reduce the size of search space, and propose an effective parallel searching method to find the optimal implementation, which leads to an auto-tuning engine.

4 Lower Bound Theory

4.1 Red-Blue Pebble Game Re-exploration

4.1.1 Basic Idea. In order to build I/O lower bounds of convolutions, we revisit the red-blue pebble game. First of all, it is not easy for a composite algorithms to deduce the value of $P(S)$ indeed, while we could try to estimate a valid lower bound of $P(S)$. In a DAG $G(V, E)$, we use the notation $|\cdot|$ to represent the number of vertices in any set, such as $|V|$ being the number of vertices in V. Denote $\mathbb{P}_S$ as the set containing all possible options of S-partitions for DAG $G(V, E)$, and each element in $\mathbb{P}_S$ represents some S-partition of $G(V, E)$. Let

$$H(S) = \min_{\{V_1,\cdots,V_h\}\in\mathbb{P}_S} \frac{|V|}{\max_{1\le i\le h}|V_i|}. \tag{2}$$

It is clear that $H(S)$ represents a lower bound of the number of sets in any S-partitions of $G(V, E)$. By the definition of $P(S)$ in Section 2.1, we have $P(S) \ge H(S)$. This fact, together with Equations (1) and (2), implies that Q satisfies

$$Q \ge S \cdot (P(2S) - 1) \ge S \cdot (H(2S) - 1). \tag{3}$$

Hence, we only need to estimate $H(2S)$ instead of $P(2S)$. Secondly, from Equation (2), $H(S)$ depends on the value of $\max_{1\le i\le h}|V_i|$, which means that the fine-gained analysis on V_i is the key. Thirdly, if we can find out the relationship between V_i and all sub-computations of $G(V, E)$, it would become possible to estimate the number of vertices in V_i. Before deducing the upper bound of $|V_i|$, we formalize the notation of multi-step partition of a DAG.

Definition 4.1. Assume that a DAG $G(V, E)$ is decomposed into n sub-DAGs $G_1(U_1, E_1), G_2(U_2, E_2), \cdots, G_n(U_n, E_n)$ where $G_j(U_j, E_j)$ is corresponding to a sub-computation. $\{G_1(U_1, E_1), \cdots, G_n(U_n, E_n)\}$ is called as *a multi-step partition* of $G(V, E)$, if and only if any input vertex of $G_j(U_j, E_j)$ must be an output vertex of $G_{j-1}(U_{j-1}, E_{j-1})$, and the internal vertex sets of all U_j's are disjoint from each other.

It is clear that any sequence of sub-computations can be represented as a multi-step partition of the DAG for the total

computation. Assume that $\{G_1(U_1, E_1), \cdots, G_n(U_n, E_n)\}$ is a multi-step partition of $G(V, E)$. If we are able to estimate all the upper bounds of $|V_i \cap U_j|$ $(j = 1, 2, \cdots, n)$ by using Property 2 and Property 3 in the definition of S-partition, it is possible to obtain the maximum of $|V_i|$.

In the following, we study the feasibility on the derivation of the upper bound of $|V_i|$ based on recursive analysis. For some j-th sub-computation, assume that the upper bound of $|V_i \cap U_j|$ has been obtained successfully. The next problem is how to estimate $|V_i \cap U_{j+1}|$. Since $|V_i \cap U_j|$ seems not to be associated with the upper bound of $|V_i \cap U_{j+1}|$, we have to focus on how the output set of U_j affects the $(j+1)$-th sub-computation. Denote $\widetilde{O}_j$ as the output set of U_j, and D_i as a dominator set of V_i. Further, we apply a new concept of *vertex generation* to determine the vertices in $\widetilde{O}_j$ which are associated with $V_i \cap U_{j+1}$.

Definition 4.2. In a DAG $G(V, E)$, *a vertex set U can generate another vertex set U'*, if and only if every path from an input of V to a vertex in U' contains some vertex in U. Furthermore, $\Theta(U)$ represents a set containing all vertices which can be generated by U.

Definition 4.2 presents a new concept of *vertex generation* which describes the dependency relationship between vertices. For example, the dominator set D_i is just one of sets which can generate V_i. It is obvious that all inputs of $V_i \cap U_{j+1}$ are included in $\Theta(D_i) \cap V_i \cap \widetilde{O}_j$. Hence, $|\Theta(D_i) \cap V_i \cap \widetilde{O}_j|$ could be used to deduce the upper bound of $|V_i \cap U_{j+1}|$. In conclusion, if we can deduce the upper bounds of $|V_i \cap U_j|$ and $|\Theta(D_i) \cap V_i \cap \widetilde{O}_j|$, it seems possible to obtain $|V_i \cap U_{j+1}|$ and $|\Theta(D_i) \cap V_i \cap \widetilde{O}_{j+1}|$ by using $\Theta(D_i) \cap V_i \cap \widetilde{O}_j$ as the inputs for $V_i \cap U_{j+1}$. Based on recursive analysis, all upper bounds of $|V_i \cap U_{j+1}|$ $(j = 1, 2, \cdots, n)$ can be established, which would lead to the upper bound of $|V_i|$.

After the feasibility analysis above on the derivation of the upper bound of $|V_i|$, *we use a simple example to show the intuition of how to obtain the the upper bound of* $|V_i|$. Assume DAG $G(U, E)$ of a composite algorithm has two sub-computations $(G(U, E) = G_1(U_1, E_1) + G_2(U_2, E_2))$. Denote k^i as the number of vertices in the dominator D_i of V_i. According to the definition of S-partition, we have $|D_i| = k^i \leq S$, and divide k^i into $k^i = k_1^i + k_2^i$ where k_1^i is the number of input vertices for $V_i \cap U_1$ and k_2^i is the number of a part of input vertices for $V_i \cap U_2$. For any integer k, we find two functions $\varphi_j(k)$ and $\psi_j(k)$, where $\varphi_j(k)$ represents the maximum of vertices in U_j generated by using k input vertices, and $\psi_j(k)$ represents the maximum of vertices in $\widetilde{O}_j$ generated by using k input vertices. Hence, $|\Theta(D_i) \cap V_i \cap U_1|$ is not larger than $\varphi_1(k_1^i)$, and at most $\psi_1(k_1^i)$ vertices are generated as the inputs for $V_i \cap U_2$. Hence, there are at most $k_2^i + \psi_1(k_1^i)$ input vertices for $V_i \cap U_2$. Further, $|\Theta(D_i) \cap V_i \cap U_2| \leq \varphi_2(k_2^i + \psi_1(k_1^i))$ is valid. Hence, we have

$$\begin{aligned} |V_i| &\leq |D_i| + |\Theta(D_i) \cap V_i \cap U_1| + |\Theta(D_i) \cap V_i \cap U_2| \\ &\leq S + \varphi_1(k_1^i) + \varphi_2(k_2^i + \psi_1(k_1^i)) \\ &\leq S + \max_{k_1+k_2 \leq S} (\varphi_1(k_1) + \varphi_2(k_2 + \psi_1(k_1))) . \end{aligned}$$

Letting $T(S) = S + \max_{k_1+k_2 \leq S}(\varphi_1(k_1) + \varphi_2(k_2 + \psi_1(k_1)))$, we achieve $|V_i| \leq T(S)$.

According to the discussion above, we deduce the general I/O lower bound result of any composite algorithm by three steps. Firstly, find two functions which can determine the numbers of vertices generated by D_i in $V_i \cap U_j$ and $V_i \cap \widetilde{O}_j$ respectively (Section 4.1.2). Secondly, deduce the upper bound of $|V_i|$ by using the upper bounds of the two functions (Section 4.1.3). Finally, establish general I/O lower bound result by substituting the upper bound of $|V_i|$ into Equations (2) and (3) (Section 4.1.4). Due to the page limit, the proofs of all lemmas and theorems, and a detailed discussion on relevant derivations, are in an extended technical report [1].

4.1.2 Maximum Vertex Generation Functions.

For any integer k and a vertex set U with any dominator set D satisfying $|D \cap U_j| + |\Theta(D) \cap \widetilde{O}_{j-1}| \leq k$, define *two vertex generation functions* for the j-th sub-computation, as follows

$$\widetilde{\varphi}_j(U, k) = |\Theta(D) \cap U \cap U_j| \text{ and } \widetilde{\psi}_j(U, k) = |\Theta(D) \cap U \cap \widetilde{O}_j|.$$

Here, $\widetilde{\varphi}_j$ and $\widetilde{\psi}_j$ represent the numbers of vertices generated by D in two vertex sets $U \cap U_j$ and $U \cap \widetilde{O}_j$ respectively. Furthermore, for any given k and the j-th sub-computation, we define *maximum vertex generation functions* as

$$\varphi_j(k) = \max_U \widetilde{\varphi}_j(U, k) \text{ and } \psi_j(k) = \max_U \widetilde{\psi}_j(U, k). \quad (4)$$

It is clear that φ_j and ψ_j provide the upper bound estimation on the number of vertices in U_j and $\widetilde{O}_j$, which can be generated by a vertex D satisfying $|D \cap U_j| + |\Theta(D) \cap \widetilde{O}_{j-1}| \leq k$.

4.1.3 Estimation of Upper Bound of $|V_i|$.

The analysis in Section 4.1.1 inspires us that I/O lower bound establishment is equivalent to finding the upper bound of $|V_i|$. Further, two kinds of maximum vertex generation functions φ_j and ψ_j in Section 4.1.2, provide us a powerful tool to respectively estimate the numbers of vertices generated by D_i in $V_i \cap U_j$ and $V_i \cap \widetilde{O}_j$. In the following, we deduce the upper bound of $|V_i|$, where V_i is any set of an arbitrary S-partition of DAG.

First of all, we deduce two auxiliary results. Let $\widetilde{O}_j^i$ be the subset of $\widetilde{O}_j$ such that for any $v \in \widetilde{O}_j^i$, any path from the input set of V to v has at least one vertex which belongs in $\cup_{k=1}^{j}(D_i \cap U_k)$.

Lemma 4.3. *$\widetilde{O}_j^i \cup (D_i \cap U_{j+1})$ is a dominator set of $\widetilde{O}_{j+1}^i$.*

Lemma 4.4. *$\widetilde{O}_j^i \cup (D_i \cap U_{j+1})$ is also a dominator of $V_i \cap U_{j+1}$.*

[1] Available at https://arxiv.org/abs/2012.15667

Next, by Lemma 4.3 and Lemma 4.4, we can obtain an upper bound of $|V_i|$, which is important for I/O complexity analysis under the red-blue pebble game model.

Theorem 4.5. *Assume that $\{G_1(U_1, E_1), \cdots, G_n(U_n, E_n)\}$ is a multi-step partition of a DAG $G(V, E)$. For any S-partition $\{V_1, \cdots, V_h\}$ of $G(V, E)$, $|V_i|$ has an upper bound*

$$T(S) = S + \max_{\sum_{j=1}^{n} k_j \le S} (\varphi_1(k_1) + \varphi_2(k_2 + \psi_1(k_1)) + \cdots\cdots + \varphi_n(k_n + \psi_{n-1}(k_{n-1} + \psi_{n-2}(k_{n-2} \cdots + \psi_1(k_1))))). \quad (5)$$

4.1.4 I/O Lower Bounds of Composite Algorithms.

Theorem 4.6. *Assume that a DAG $G(V, E)$ describes an algorithm with n steps. All sub-computations in n steps are corresponding to a multi-step partition of the DAG. Given a fast memory of size S, to finish the algorithm, the number Q of I/O operations between the fast memory and the slow memory satisfies*

$$Q \ge S \cdot \left(\frac{|V|}{T(2S)} - 1 \right). \quad (6)$$

Although Equation (6) is similar to Equation (1), it is easier to estimate T for a composite algorithm than obtaining P. Theorem 4.6 concludes how the I/O lower bound depends on the upper bounds of φ_j and ψ_j. It not only presents a general theoretical result, but also provides an I/O lower bound proof approach for a composite algorithm which consists of several stages with different kinds of computational patterns. This proof approach can be described as three steps.

- Step 1: Estimate the total number of vertices $|V|$ in DAG $G(V, E)$ of a composite algorithm.
- Step 2: For the j-th step in the multi-step partition of $G(V, E)$, estimate φ_j and ψ_j respectively. Based on Equation (5), compute the value of $T(S)$.
- Step 3: By Equation (6), deduce I/O lower bound.

Based on the above approach, we establish the I/O lower bounds of the direct convolution and Winograd algorithm respectively in the following sections.

4.2 I/O Lower Bound of Direct Convolution

Figure 4 shows a DAG $G(V, E)$ of the direct convolution. It is clear that the direct convolution consists of two steps. The first step is to generate a lot of product terms by using inputs in the input images and kernels. In DAG $G(V, E)$, we call the product vertices as the vertices which are corresponding to the product terms generated by the first step in the direct convolution. Denote I_i as the i-th sliding input tensor with the size of $W_{ker} \times H_{ker} \times C_{in}$ from an input image by using a sliding window. Let K_j be the j-th kernel whose size is also $W_{ker} \times H_{ker} \times C_{in}$. For each I_i and K_j, the first step generates $W_{ker}H_{ker}C_{in}$ product terms by executing the corresponding element product of I_i with K_j. The second step is to sum the product terms generated by I_i and K_j to form one of final outputs based on a summation tree. The summation tree is a

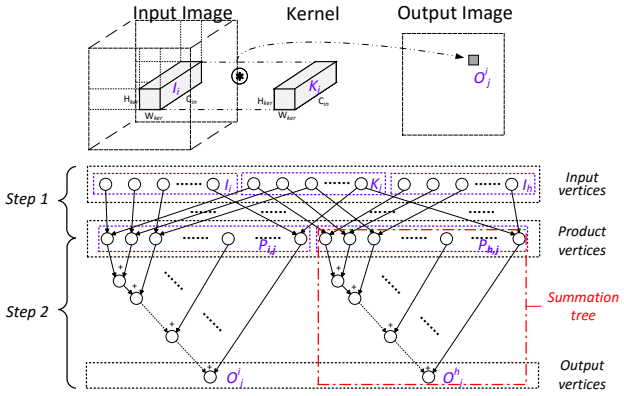

Figure 4. DAG of Direct Convolution.

sub-DAG with the tree structure in which, except for all input vertices, the in-degree of other vertices is at most two, and all inputs of the tree would be summed together to the only one output (Figure 4). After the summation process, the direct convolution is finished. Hence, the multi-step partition of $G(V, E)$ can be written as $G(V, E) = G_1(U_1, E_1) \cup G_2(U_2, E_2)$ where the sub-DAG $G_i(U_i, E_i)$ is corresponding to the i-th step of the direct convolution.

Lemma 4.7. *In DAG of a direct convolution, the total number of vertices is*

$$|V| = (2W_{ker}H_{ker}C_{in} - 1)W_{out}H_{out}C_{out} + W_{in}H_{in}C_{in} + W_{ker}H_{ker}C_{in}C_{out}. \quad (7)$$

Denote R as the maximum reuse number of each input (element) in an input image by different sliding windows. Its value is $R = W_{ker}H_{ker}/\mu^2$ where μ is the stride size. For the two steps in the direct convolution, we can prove that $\psi_1 = \varphi_1$, $\varphi_1(k_1) \le 2S\sqrt{Rk_1}$, and $\varphi_2(k_2) \le k_2 - 1$ are valid for any integers k_1 and k_2, which leads to an estimation of $T(S)$.

Lemma 4.8. *For a direct convolution, $T(S) \le 4S\sqrt{RS} + S - 1$.*

Based on Theorem 4.6, Lemma 4.7 and Lemma 4.8, we can establish I/O lower bound of the direct convolution.

Theorem 4.9. *The I/O lower bound of a direct convolution (DC) is*

$$Q_{lower\ DC} = \Omega\left(\frac{W_{ker}H_{ker}C_{in}W_{out}H_{out}C_{out}}{4\sqrt{2RS}} \right). \quad (8)$$

In recent years, the minimum memory access of direct convolution has been obtained [6, 11]. Based on the red-blue pebble game model, we avoid to solve the intricate optimization problem in [11]. Besides, based on the proposed general theoretical result (Theorem 4.6), we analyze the two steps of direct convolution directly, instead of transforming direct convolution into matrix-matrix multiplication [6]. Furthermore, The I/O lower bound in Equation (8) is equivalent to

the I/O lower bounds of direct convolution in [6, 11], However, our proposed result on the direct convolution is tighter with the more precise coefficient.

4.3 I/O Lower Bound of Winograd Algorithm

In Winograd algorithm, since the size of three transformation matrices A, B and L is small (Figure 2), we assume that they can be always stored in fast storage, and their volume can be ignored compared with the size S of the fast memory. Furthermore, as Winograd algorithm requires $W_{ker} = H_{ker}$, we denote r as W_{ker} or H_{ker} briefly.

Lemma 4.10. *In DAG of Winograd algorithm, the total number of vertices is*

$$|V| = O\left(\frac{W_{out}H_{out}C_{out}C_{in}(e+r-1)^4}{e^2}\right). \tag{9}$$

For 4 sub-computations of Winograd algorithm (Figure 3), we can deduce the upper bounds of φ_j and ψ_j $(1 \le j \le 4)$. $\varphi_1(k_1) \le 6k_1(e+r-1)^4/er$, $\psi_1(k_1) \le 3k_1(e+r-1)^2/er$, $\psi_2 = \varphi_2$, $\varphi_2(k_2) \le k_2\sqrt{k_2} + (e+r-1)^2 S\sqrt{k_2}/e^2$, $\varphi_3(k_3) \le k_3 - 1$, $\psi_3(k_3) \le \min\{k_3/2, S(e+r-1)^2/e^2\}$, and $\varphi_4(k_4) \le \min\{(2k_4-1)e^2, (2(e+r-1)^2-1)S\}$ are valid for any integers k_1, k_2, k_3 and k_4.

Lemma 4.11. *For Winograd algorithm,*

$$T(S) = O\left(2\frac{(e+r-1)^3}{er}S\sqrt{S} + \frac{6(e+r-1)^2}{er}S\right). \tag{10}$$

So far, the lower bound of I/O complexity of Winograd algorithm can be established according to the proof approach in Section 4.1.4.

Theorem 4.12. *The I/O lower bound of Winograd algorithm (WA) is*

$$Q_{lower\ WA} = \Omega\left(\frac{W_{out}H_{out}C_{out}C_{in}(e+r-1)r}{e\sqrt{S}}\right). \tag{11}$$

It is worth mentioning that, if R is 1 in Equation (8) and e is 1 in Equation (11), the I/O lower bounds of the direct convolution and Winograd algorithm are similar to that of matrix-matrix multiplication [17]. In this case, the lower bound of data movement is inversely proportional to $\sqrt{S}$, which is consistent with the communication-optimal implementation of matrix-matrix multiplication [20].

5 Near I/O-optimal Strategy

5.1 Methodology for Near I/O-optimal Strategy

In the proposed general I/O lower bound theory, the highest order term in I/O lower bound result (6) must be determined by some φ_j due to the definition (5) of T. Specifically, for the direct convolution, the maximum vertex generation function φ_2 for the last step determines the highest order term in I/O lower bound (Equation (8)). For Winograd algorithm, the highest order term in I/O lower bound (Equation (11)) comes from φ_3 for the third step, rather than φ_4 for the last step. As the highest order term of I/O lower bound result represents the main part of I/O number, the relevant φ_i points to the major process which involves most of I/O operations.

By the function φ_j which determines the highest order term in I/O lower bound result of a composite algorithm, we are able to find which data should be fully reused in the on-chip memory, and focus on reducing I/O operations in the j-th step of the composite algorithm. In detail, for the direct convolution, φ_2 which determines the highest order term in Equation (8), indicates that minimizing the number of I/O operations needs to maximize the output data reuse. For Winograd algorithm, φ_3 inspires us to maximize the data reuse of two temporary arrays which are involved during the third step.

After determining which data reuse should be exploited, the dataflow strategy can be designed to maximize the reuse of such data. For the direct convolution and Winograd algorithm, we propose different schedules by exploiting the reuse of relevant data respectively.

5.2 Dataflow Design for Direct Convolution

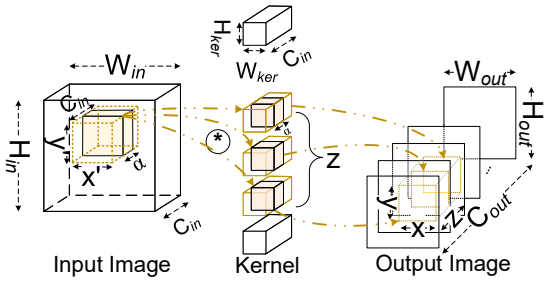

Figure 5. Dataflow Design for Direct Convolution.

For the direct convolution, the highest order term of I/O lower bound (Equation (8)) comes from φ_2 for the last step. φ_2 indicates that the output data reuse should be fully exploited, which implies that we need to use the least inputs to produce the most outputs. Hence, the dataflow design should assign most of the effective on-chip memory to portions of outputs. Figure 5 shows a sub-block of the output image with the dimension of $x \times y \times z$. Based on the fundamental principle above, to reach the minimum off-chip memory access, we tend to choose $xyz \approx S/N_p$ where N_p is the total number of active processors.

To compute the output sub-block $x \times y \times z$, we need the inputs in the corresponding $x' \times y'$ locations from all input channels (the yellow sub-block in an input image) and z kernels associated with the partial output channels (the yellow kernels), as shown in Figure 5. Since the on-chip memory is limited and tends to be used for storing the most outputs, it is necessary to load the required inputs and kernels by a

series of stages, rather than at a time. During each stage, a portion of inputs $x' \times y' \times \alpha$ and the corresponding weights $H_{ker} \times W_{ker} \times \alpha$ of z kernels are loaded into the on-chip memory (Figure 5). Since each input of the i-th channel only can be reused by the weights of the i-th channel, rather than other channels. In order to put the larger output sub-block in the limited on-chip memory, we set $\alpha = 1$, which indicates that our dataflow design is to load a $x' \times y'$ tile with a fixed channel index firstly and then slide the tile along the channel direction.

After loading a $x' \times y'$ input tile and the corresponding weights of z kernels into the on-chip memory, a partial sum can be performed on the output sub-block. To update the whole output sub-block, we continuously slide the $x' \times y'$ input tile along the channel direction, and load the corresponding inputs and weights (in the yellow blocks), and perform partial updates. Consequently, updating each output sub-block only needs to load the required inputs and weights from the off-chip memory to on-chip memory exactly once. Meanwhile, different output sub-blocks are updated by N_p processors in parallel.

In our dataflow design, there are $(W_{out}H_{out}C_{out})/(xyz)$ output sub-blocks in total. To update each sub-block, we need $x'y'C_{in}$ inputs from an input image and $W_{ker}H_{ker}C_{in}z$ weights from z kernels. As $R = W_{ker}H_{ker}/\mu^2$, $x' \approx \mu x$ and $y' \approx \mu y$, the I/O volume of reading data is

$$Q_{DC\ reading} \approx \frac{H_{out}W_{out}C_{out}}{xyz}\left(H_{ker}W_{ker}C_{in}(z + \frac{xy}{R})\right)$$
$$\geq H_{out}W_{out}C_{out}H_{ker}W_{ker}C_{in}\left(2\sqrt{\frac{1}{Rxyz}}\right), \quad (12)$$

where the final equality holds if and only if $xy = Rz$. By the fact $R = W_{ker}H_{ker}/\mu^2$, $x' = \mu x$ and $y' = \mu y$ again, the requirement of $xy = Rz$ leads to $x'y' = zW_{ker}H_{ker}$, which determines the optimal size of each $x' \times y'$ tile. Further, the I/O volume of storing outputs is $W_{out}H_{out}C_{out}$. When we choose $xyz \approx S/N_p$ and $xy = Rz$, the total I/O volume is

$$Q_{DC} \approx \frac{2H_{out}W_{out}C_{out}H_{ker}W_{ker}C_{in}}{\sqrt{RS/N_p}} + H_{out}W_{out}C_{out}. \quad (13)$$

If $N_p = 1$ and $\frac{H_{ker}W_{ker}C_{in}}{\sqrt{SR}} \gg 1$ which is easily satisfied in CNN applications due to S usually being equal to or less than KB level, Q_{DC} reaches the I/O lower bound (Theorem 4.9). This fact indicates that, sequentially executing the dataflow and assigning most of the effective on-chip memory to the outputs can reach the minimum off-chip memory access. Otherwise, if we perform the dataflow in parallel, Equation (13) means that fully utilizing the on-chip memory owned by each processor to produce the partial sum could maximize the output data reuse and reduce the data transmission in the memory hierarchy.

In order to view the proposed dataflow at a high level, we conclude the details of this design as follows:

- The input data reuse is fully considered. In fact, one input is reused by weights of z kernels, and one weight is reused by $x \times y$ outputs. On the other hand, one input is also reused by at most R sliding windows on each $x' \times y'$ tile.
- The output data reuse is fully exploited. In fact, the partial sum can always stay into the on-chip memory during the update process, and they are just written back to the off-chip memory only once. To make sure the larger output sub-block can be loaded in the on-chip memory, the optimal tiling is designed to slide the $x' \times y'$ tile along the channel direction, which reveals that the loading of inputs along the width and height directions should be considered prior to the channel direction.
- In order to achieve the I/O lower bound, the $x \times y \times z$ output sub-block needs to satisfy $xy = Rz$, which is called as the optimality condition in this work. Under this condition, $x'y' = zW_{ker}H_{ker}$, which determines the optimal size of each input tile.

5.3 Dataflow Design for Winograd Algorithm

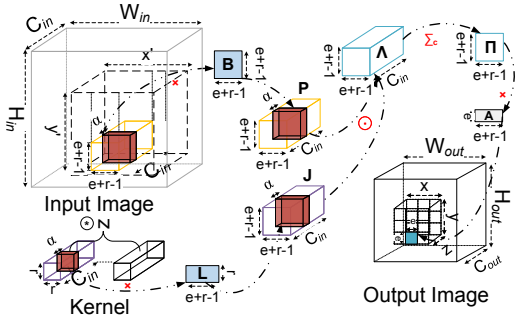

Figure 6. Dataflow Design for Winograd Algorithm.

Similar to the analysis in the dataflow design for direct convolution, φ_3 determining the highest order term in I/O lower bound of Winograd algorithm (Equation (11)), leads us to maximize the data reuse of temporary arrays involved during the third step.

To compute each $x \times y \times z$ output sub-block, Winograd algorithm needs to partition further sub-block into xy/e^2 smaller sub-blocks each of which has the size of $e \times e \times z$. Each $e \times e \times z$ small sub-block is computed by using the corresponding $(e+r-1) \times (e+r-1)$ locations from all input channels of the input images (i.e., the yellow block in the input image) and z kernels associated with the partial output channels (Figure 6), which are loaded into on-chip memory by a series of stages due to the limited on-chip memory. Based on the same discussion in the dataflow design, each stage loads a $(e+r-1) \times (e+r-1)$ input tile at an input channel (which means $\alpha = 1$) and the corresponding r^2 weights at the same channel of a kernel, and then produce a partial sum Λ

(Figure 6). We allocate two $(e+r-1)\times(e+r-1)$ temporary arrays in the on-chip memory for the summation of all partial sums along the channel direction. The first array is used to save the last summation result, and the second one is for the generation of a new partial sum. When a new partial sum is created in the second array, it would be added to the first array. After collecting all partial sums along the channel direction, the $(e+r-1)\times(e+r-1)$ summation matrix Π naturally generates in the first array (Figure 6), which would be multiplied with a transform matrix to deduce $e\times e$ outputs in the same channel of the small output sub-block. To complete the update of each small sub-block with the size of $e\times e\times z$, each processor continuously loads the required inputs and weights (the red blocks in Figure 6), and performs partial updates. In order to exploit the parallelism of the computation of $x\times y\times z$ outputs, each processor could use serval threads to execute the computation of all $e\times e$ tiles in a channel in parallel. For the update of each $x\times y\times z$ sub-block, every e^2 outputs rely on two $(e+r-1)\times(e+r-1)$ temporary arrays at a time. To maximize the data reuse of temporary arrays, we should use the most on-chip memory to store the $2xyz/e^2$ required temporary arrays. Hence, our design chooses $2\frac{(e+r-1)^2}{e^2}xyz \approx S/N_p$.

In the dataflow above, an output image is divided into $(W_{out}H_{out}C_{out})/(xyz)$ sub-blocks. For each sub-block, we need to load $x'y'C_{in}$ inputs from an input image and zr^2C_{in} weights from z kernels. As $\mu=1$ is only valid in Winograd algorithm, we have $x'\approx x$ and $y'\approx y$. The I/O volume of reading data can be estimated as follows

$$\begin{aligned} Q_{WA\ reading} &\approx \frac{H_{out}W_{out}C_{out}}{xyz}\left(xyC_{in}+zr^2C_{in}\right) \\ &\geq H_{out}W_{out}C_{out}C_{in}\left(2\frac{r}{\sqrt{xyz}}\right), \end{aligned} \tag{14}$$

where the final equality holds if and only if $xy=r^2z$. Due to $R=r^2$ in Winograd algorithm, $xy=r^2z$ leads to $xy=Rz$, which is similar to the optimality condition for the dataflow of direct convolution. In addition, the I/O volume of writing outputs is $W_{out}H_{out}C_{out}$. As $2\frac{(e+r-1)^2}{e^2}xyz \approx S/N_p$, the total I/O volume is

$$Q_{WA} \approx \frac{2H_{out}W_{out}C_{out}C_{in}r(e+r-1)}{e\sqrt{S/N_p}} + H_{out}W_{out}C_{out}.$$

For our dataflow design of Winograd algorithm, we list two specific details as follows:

- The dataflow design of direct convolution mainly focuses on the output data reuse, while the dataflow design of Winograd algorithm is to exploit the data reuse of temporary arrays and combine input data reuse in the best way. In addition, each $(e+r-1)^2$ inputs are reused by weights of z kernels, and each r^2 weights are reused by e^2 outputs.
- The parallelism of the computation of $x\times y\times z$ outputs is fully considered. The update of every $e\times e$ tiles at an output channel is performed in parallel. To achieve a high parallelism and data reuse, the most on-chip memory is for loading the temporary arrays.

6 Auto-tuning for Implementation

6.1 Auto-tuning Engine

The dataflow design above just provides a coarse-grained strategies to minimize the off-chip memory access. In order to achieve an optimal implementation, fine-grained computational schedule and memory access schedule are still needed. In this section, we mainly consider the optimal implementation on accelerators, such as GPU. Similar optimization can be used for other hardware backends.

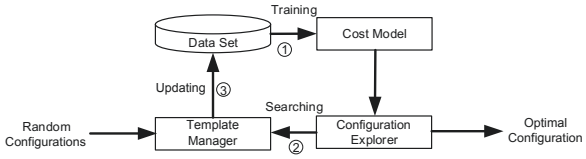

Figure 7. Auto-tuning Engine.

For a given coarse-grained schedule, we define the configuration as a group of key performance parameters, including specific input shape and layout, number of threads in each thread block, tiling size, the shared memory size allocated to each thread block. Each configuration provides the description of an implementation way. All possible configurations constitute a configuration space whose size usually is over billions. In order to rapidly find the optimal choice in the huge space, we built an auto-tuning engine based on the learning-based cost modeling method. Figure 7 shows the overview of our auto-tuning engine, which consists of three main components: a template manager that measures the execution time of any given configuration, and a cost model that predicts the cost of any given configuration, and a configuration explorer that searches promising new configurations.

Template Manager: In the low-level implementation, the proposed dataflow schedules are described as a template. Template manager is in charge of all schedule template, and generates various configurations for each template.

Cost Model: We use XGBoost method [4] to train a gradient tree boosting model as the cost model to predict the runtime of any configuration. The model is trained using measurement data, which is consisted of a configuration and its execution time. During the auto-tuning process, the cost model would be updated periodically as the configuration explorer finds more configurations and updates the training dataset.

Configuration Explorer: During the configuration searching, the configuration explorer uses the trained cost model to

predict the cost of any configuration, and searches the potential optimal configuration in the search space. Although the cost model could reduce the time to evaluate configurations, the searching process is still expensive due to a huge search space with over billions of size.

6.2 Searching Based on Optimality Condition

In order to improve the search efficiency, we construct a searching domain based on the optimality condition, which is helpful for significantly reducing the size of search space. Besides, we use a heuristic method to rapidly search promising configurations.

Searching Domain: Table 1 presents the searching domain. According to the dataflow design, the tile is loaded into on-chip memory as a whole, which implies that $xyz \leq S_b$, S_b is the shared memory size for each block. Furthermore, the optimality condition $xy = Rz$ leads to $z \leq \sqrt{S_b/R}$ and $xy \leq \sqrt{S_b R}$. In order to achieve a high level parallelism, at least two thread blocks are guaranteed to concurrently run on one streaming multiprocessor (SM), resulting in $S_b \leq S_{sm}/2$.

Table 1. Searching Domain.

Parameters	Definition and Constrains
H_{in}, W_{in}, C_{in}	Input shape
$H_{out}, W_{out}, C_{out}$	Output shape
H_{ker}, W_{ker}	Kernel shape
CHW, CWH, HWC	Layout
S_{sm}	Shared memory size of SM
S_b	Shared memory size for each block $S_b \leq S_{sm}/2$
x, y, z	Tile sizes being the factor of $H_{out}, W_{out}, C_{out}$, $xyz \leq S_b$, $z \leq \sqrt{S_b/R}$, and $xy \leq \sqrt{S_b R}$
N_{xt}, N_{yt}, N_{zt}	Thread numbers being the factor of x, y, z

Searching Process: To find many promising configurations, the configuration explorer performs a searching process to select configurations from the searching domain. At the beginning of the searching process, n_s random configurations are chosen as initial guesses. During each searching step, the configuration explorer randomly walks from each initial guess to its nearby configuration in the searching domain. Each random walk tends to converge on a configuration that has lower predicted costs. Consequently, the n_s parallel random walks generate n_s promising configurations, which are saved as the initial guesses for the next searching step. Until all predicted costs of the n_s selected configurations are lower than a threshold, they are outputted as a solution.

6.3 Auto-tuning Process

The proposed auto-tuning engine searches the optimal implementation iteratively. Each iteration consists of three stages: (1) *Model Training* that trains the cost model, (2) *Configuration Searching* that applies the cost model to select multiple promising configurations, (3) *Dataset Updating* that measures the new configurations and updates the dataset. Until the measurement runtime of the selected configurations does not decrease for hundreds of iterations, the auto-tuning process would end. The parallel strategy corresponding the best selected configuration is the implementation of our near I/O-optimal dataflow.

7 Evaluation

In this section, we evaluate our proposed near I/O-optimal dataflow designs for the direct convolution and Winograd algorithm respectively. We first evaluate the optimal dataflow implementations derived from the proposed auto-tuning engine, and then compare the speeds of different automation searching methods, and finally demonstrate our implementation can achieve performance speedup in end-to-end cases. Our evaluation is mainly performed in the NVIDIA 1080Ti and V100 GPUs.

To evaluate our work from a broad scale, we use synthetic convolution cases with different W_{ker} H_{ker} and the stride μ. On the one hand, in cuDNN library, the direct implementation of convolutions mainly has two approaches: direct convolution and image2col method [16], where the direct convolution occasionally fails for some different input shapes, and the image2col method are usually better than the direct convolution. In order to present the superior of our implementations, we compare with the best one of two direct implementations in cuDNN. On the other hand, the indirect implementation of convolutions in cuDNN mainly is Winograd algorithm. The following evaluation compares the runtime of different convolution kernels of ours and cuDNN, where CUDA-9.0 and cuDNN-7.0.3 are used.

To evaluate the auto-tuning engine, we first compare the searching performance of our proposed searching method with different searching strategies in TVM, which represents the state-of-the-art technique for auto-tuning a convolution operation, and then compare our searched implementation with the optimal solution provided by TVM.

7.1 Performance Comparison with cuDNN

Figure 8 shows the performance comparison on the implementations of the direct convolution and Winograd algorithm respectively. We can find that our I/O-optimal dataflow implementations can achieve 3.32× performance speedup on average. We have three important observations from the results.

Firstly, the benefit from the dataflow is consistent as the H_{in} and W_{in} increase, and our methodology can have significant performance improvement. This mainly owes to the design of exploiting input and output data reuse. I/O dataflow design maximizes the data reuse of the $x' \times y'$ tile at a given channel. When H_{in} and W_{in} become larger, the more data reuse can be achieved.

Secondly, when C_{out} is small, the dataflow contribution is always higher for the direct convolution. Conversely, when

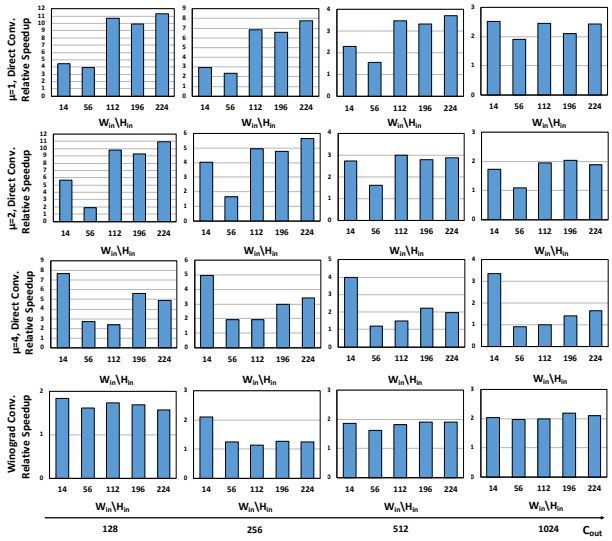

Figure 8. Performance Comparison of Dataflow Design over cuDNN for Direct Convolution and Winograd Algorithm on 1080Ti GPU. For all convolutions, $H_{ker} \times W_{ker} = 3 \times 3$ and $C_{in} = 256$.

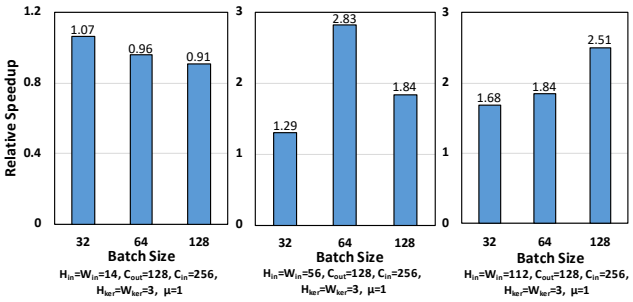

Figure 9. Performance Comparison of Dataflow Design over cuDNN for Batched Direct Convolution Test on 1080Ti GPU.

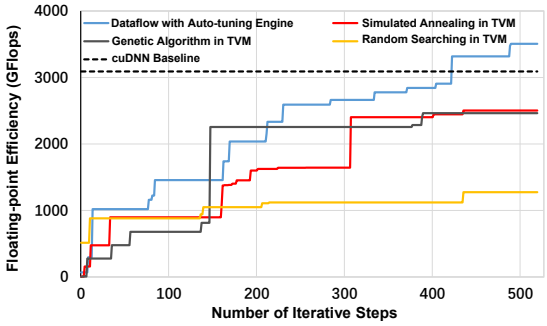

Figure 10. Comparison of Different Automation Methods.

C_{out} is large, the benefit from the dataflow is always higher for Winograd algorithm.

Thirdly, on the whole, the dataflow benefits decrease as the stride μ increase. This is because the motivation of the I/O dataflow design is to minimize the off-chip memory access. When the stride μ is larger, more off-chip memory accesses gradually become independent with each other.

Furthermore, Figure 9 shows the batched convolution test. It is clear that, compared with scaling the batch size of cuDNN, our I/O-optimal dataflow still achieves 1.51× performance speedup on average. On the one hand, For a given batch-size, when H_{in} and W_{in} increases, the performance improvement from our dataflow design gradually becomes apparent. On the other hand, when H_{in} and W_{in} are small, the dataflow contribution is small. However, when H_{in} and W_{in} become larger, the convolution needs more I/O operations, and the benefit from the dataflow becomes greater. When H_{in} and W_{in} are 112, the speedup becomes larger with the batch size increasing.

7.2 Performance Comparison with TVM

Table 2 presents the detail information about configuration space, the number of iterations and the best solution's runtime of the auto-tuning engine and TVM during searching the optimal implementations of different convolution layers in AlexNet on V100 GPU. We have three important observations from the experiment results. Firstly, the constraints for the templates and the proposed searching domain can successfully reduce the size of configuration space to about $20\% - 50\%$ for the direct convolution and 50% for Winograd algorithm. Secondly, the proposed auto-tuning engine finds the final solution faster than TVM, thanks to the proposed searching domain. Thirdly, the final configuration found by the auto-tuning engine usually has a shorter runtime than the best solution in TVM. The three facts above demonstrate that the auto-tuning engine has the strong scaling efficiency for searching optimal configuration.

Figure 10 shows the comparison of different automation methods for searching an optimal direct convolution implementation of the conv1 in Table 2 on V100 GPU. The ML-based model in TVM starts with no training data and uses the collected data to improve itself. The X-axis is the number of iterative steps and the Y-axis is the floating-point arithmetic efficiency in GFlops. From Figure 10, we observe a similar trend for all automation methods. During the iterations, each automation method gradually finds the better configuration with higher floating-point arithmetic efficiency. It should be noted that the proposed auto-tuning engine is able to find better configurations much faster than the others. This mainly owes to two reasons. On the one hand, the I/O optimality condition is used to prune configuration search space, which leads to the proposed searching domain. On the other hand, the parallel searching method effectively improves the searching process in the searching domain.

Table 2. Comparison of TVM with Auto-tuning Engine (ATE).

Convolution	Parameter						Size of Search Space			Iterations			Performance of Solution (GFlops)		
	C_{in}	H_{in}/W_{in}	C_{out}	H_{ker}/W_{ker}	stride	padding	TVM	ATE	ATE/TVM	TVM	ATE	TVM/ATE	TVM	ATE	ATE/TVM
conv1	3	227	96	11	4	0	9.29×10^6	4.81×10^6	51.78%	142	197	0.72	2927.30	5377.06	1.84
conv2	96	27	256	5	1	2	2.25×10^8	4.76×10^7	21.16%	762	449	1.53	5909.73	6426.83	1.09
conv3	256	13	384	3	1	1	1.87×10^7	4.51×10^6	24.12%	877	389	2.25	2107.68	2555.93	1.21
conv4	384	13	256	3	1	1	1.54×10^7	5.23×10^6	33.96%	784	407	1.93	2040.57	2040.92	1.00
conv3_wino	256	13	384	3	1	1	2.59×10^5	1.36×10^5	52.51%	352	202	1.74	6700.77	6726.17	1.01
conv4_wino	384	13	256	3	1	1	1.58×10^5	8.06×10^4	51.01%	587	286	2.05	7121.57	7118.23	1.00

7.3 Performance Comparison on CNN Models

The convolution layer is important and popular in many modern CNN models. In the following, we demonstrate that our proposed auto-tuning engine can help for accelerating CNN inference. In the experiment on end-to-end models, the runtime is the time for inference (or forward computation).

Figure 11 shows the performance comparison of the dataflow design and cuDNN on different CNN models. For SqueezeNet, Vgg-19, ResNet-18, ResNet-34 and Inception-v3, our optimal implementation can achieve 2.67×, 1.09×, 1.02×, 1.09× and 1.23× performance speedup respectively compared with using cuDNN. The performance benefits come from two aspects. The different kinds of convolutions take up the main part of CNN models. Besides, for each convolution layer, the proposed auto-tuning engine could find a better implementation than cuDNN.

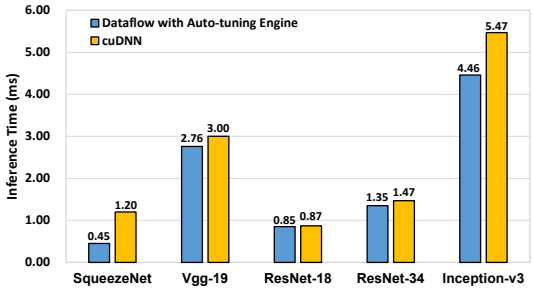

Figure 11. Performance Comparison of Dataflow Design over cuDNN on different CNN Models on V100 GPU.

7.4 Sensitivity for GPU Architecture

To demonstrate the scalability on GPU architecture, we evaluate the proposed dataflow with auto-tuning engine on Pascal and Maxwell architectures. We use one kind of Pascal architectures: 1080Ti, and one kind of Maxwell architecture: GTX Titan X. Figure 12 shows the evaluation results on the above two architectures. The proposed dataflow is much faster than cuDNN. Compared with the solution of TVM, for the direct convolution, the improvement of our implementation on these architectures can achieve about 1.05× and 1.27× respectively. For Winograd algorithm, the speedups of our dataflow are 1.12× and 1.01× respectively on these architectures.

In addition, we compare the dataflow design with MIOpen library on AMD GFX906 platform (Pre-Wukong GPU), and

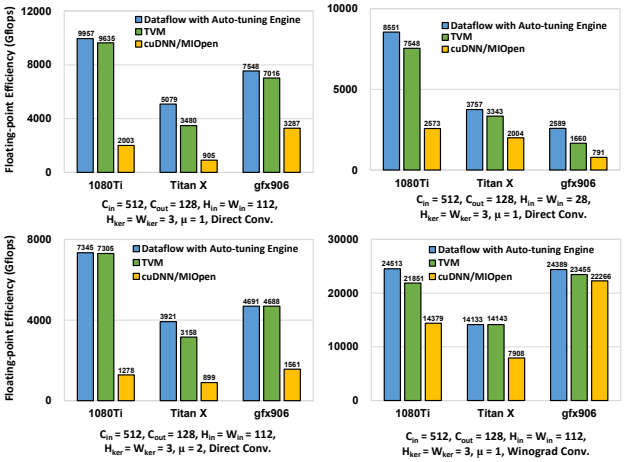

Figure 12. Sensitivity on different GPU architectures.

use ROCm-2.9 and MIopen-2.1 in this evaluation. On average, the performance improvement is up to 2.86× and 1.10× for direct convolution and Winograd algorithm respectively. Besides, compared with the solution of TVM, our optimal implementation achieves 1.21× speedup for the direct convolution and 1.03× speedup for Winograd algorithm. We find that our optimal implementation is well ported to different architectures and achieve a consistent performance speedup.

8 Related Work

The red-blue pebble game is widely used in theory analysis of communication lower bound to guide optimal communication strategy. After Hong & Kung established the I/O complexity theory [17], Savage developed the notion of S-span to derive Hong-Kung style lower bounds [22]. Kwasniewski et al. provided a new proof of I/O complexity of matrix-matrix multiplication and designed a parallel algorithm to reach its lower bound [20]. Although the red-blue pebble game model has been proposed for many years [1–3, 12, 23, 27], it is still difficult to use this model to establish I/O lower bounds of composite algorithms which involve several different kinds of computational patterns [13]. To get around the essential difficulties, the lower bound of composite algorithms was considered by modifying the red-blue pebble game model into a red-blue-white pebble game model [13], which uses some restrictions on models, such as the limitation of disallowing re-computation of values on the DAG [13]. However,

such restrictions seem inappropriate for the lower bound analysis of some convolution algorithms. For example, Winograd algorithm allows re-computation of values to decrease the number of I/O operations. In order to solve the difficulties, this work at first establishes a general I/O lower bound theory for any composite algorithm based on the red-blue pebble game model without introducing the limitation of disallowing re-computation of values on the DAG.

For convolutions in DNN, Demmel et al. estimated the minimum memory access of direct convolution by solving an intricate optimization problem [11]. Furthermore, Chen et al. transformed the direct convolution into Matrix-matrix multiplication, and successfully deduced the lower bound of the off-chip communication of direct convolution in CNN accelerators [6]. However, our work is the first time to perform a systematic analysis of diverse convolution algorithms in deep learning by developing a general I/O lower bound theory for any composite algorithm. It is worth mentioning that the I/O lower bound in Equation (8) is equivalent to the I/O lower bound of direct convolution in [6, 11], while our proposed result on direct convolution is the tighter lower bound with a more precise coefficient. Besides, the previous works [6, 11] mainly focus on the direct convolution, and seem not easy to adapt to Winograd algorithm. However, to the best of our knowledge, this work at first establishes the I/O lower bound of Winograd algorithm.

To fully exploit the research efforts of convolution algorithm and micro-architecture optimizations, many software libraries, such as cuDNN, are launched to pack these optimizations together in order to reduce programming difficulty. However, due to the increasing demand on performance, directly using the software libraries sometimes is not satisfactory. In recent years, the convolution optimization is widely concerned. Some excellent implementations are proposed for different convolution algorithms [7, 18, 21, 24, 25]. However, most of the studies mainly focus on the optimization from experience [32]. In this work, we try to propose the I/O-optimal dataflow based on the lower bound theoretical analysis. By comparing the I/O volume of the dataflow with the lower bound, we find the optimality condition for I/O-optimal design. On the other hand, in the convolution optimization, the combinatorial choices of memory access, threading pattern, and novel hardware primitives creates a huge configuration space. A common way is to adopt a predefined cost model to guide the search, but building an accurate cost model is difficult due to the increasing complexity of modern hardware. To addresses these challenges, some searching strategies based on the learning-based cost models are proposed, in which TVM represents the state-of-the-art auto-tuning technique. However, it still needs a large search cost due to the huge search space. In this work, this work firstly considers to use the deduced optimality condition to fully reduce the size of search space, and proposes an effective parallel searching method to find the optimal implementation, which leads to an effective auto-tuning engine. Compared with TVM, it could faster find a better final solution.

9 Conclusion

In this paper, we have tackled the challenge of building I/O lower bound theory and designing I/O-optimal dataflow implementations for convolutions. By fine-grain viewing the recent lower bound theory developed under the red-blue pebble game model, we fully consider the influence of sub-computations to each other, and propose a general I/O lower bound theory for composite algorithms. Based on the proposed theory, we establish the communication lower bound results for the typical representatives of direct and indirect convolution methods, which are the direct convolution and Winograd convolution algorithms. Furthermore, for each approach, we design the near I/O-optimal dataflow strategy based on the lower bound analysis. By developing an auto-tuning engine for searching the optimal configuration, we push the envelope of performance of our dataflow designs further.

Acknowledgments

The authors would like to express their gratitude to all anonymous reviewers for their valuable comments and helpful suggestions. This work is supported by The National Key Research and Development Program of China under Grant No. (2018AAA0103302, 2016YFC1401706, 2016YFB0200800), National Natural Science Foundation of China under Grant No. (61802369, 61972377, 62032023) and Huawei Technologies Co., Ltd.. The authors also thank Dr. Long Wang and the group of Huawei Technologies Co., Ltd. for their help to this research.

References

[1] Alok Aggarwal and S. Vitter, Jeffrey. 1988. The Input/Output Complexity of Sorting and Related Problems. *Commun. ACM* 31, 9 (Sept. 1988), 1116–1127. https://doi.org/10.1145/48529.48535

[2] Grey Ballard, James Demmel, Olga Holtz, and Oded Schwartz. 2011. Minimizing Communication in Numerical Linear Algebra. *SIAM J. Matrix Anal. Appl.* 32, 3 (2011), 866–901. https://doi.org/10.1137/090769156

[3] Grey Ballard, James Demmel, Olga Holtz, and Oded Schwartz. 2013. Graph Expansion and Communication Costs of Fast Matrix Multiplication. *J. ACM* 59, 6, Article 32 (Jan. 2013), 23 pages. https://doi.org/10.1145/2395116.2395121

[4] Tianqi Chen and Carlos Guestrin. 2016. XGBoost: A Scalable Tree Boosting System. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (San Francisco, California, USA) *(KDD '16)*. Association for Computing Machinery, New York, NY, USA, 785–794. https://doi.org/10.1145/2939672.2939785

[5] 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 *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 578–594.

https://www.usenix.org/conference/osdi18/presentation/chen

[6] Xiaoming Chen, Yinhe Han, and Yu Wang. 2020. Communication Lower Bound in Convolution Accelerators. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 529–541. https://doi.org/10.1109/HPCA47549.2020.00050

[7] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. 2016. Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks. *SIGARCH Comput. Archit. News* 44, 3 (June 2016), 367–379. https://doi.org/10.1145/3007787.3001177

[8] Yu Cheng, Duo Wang, Pan Zhou, and Tao Zhang. 2017. A Survey of Model Compression and Acceleration for Deep Neural Networks. (2017). http://arxiv.org/abs/1710.09282

[9] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cuDNN: Efficient Primitives for Deep Learning. (2014). http://arxiv.org/abs/1410.0759

[10] Jee W. Choi, Amik Singh, and Richard W. Vuduc. 2010. Model-Driven Autotuning of Sparse Matrix-Vector Multiply on GPUs. *SIGPLAN Not.* 45, 5 (Jan. 2010), 115–126. https://doi.org/10.1145/1837853.1693471

[11] James Demmel and Grace Dinh. 2018. Communication-Optimal Convolutional Neural Nets. (2018). http://arxiv.org/abs/1802.06905

[12] James Demmel, Laura Grigori, Mark Hoemmen, and Julien Langou. 2012. Communication-optimal Parallel and Sequential QR and LU Factorizations. *SIAM Journal on Scientific Computing* 34, 1 (2012), A206–A239. https://doi.org/10.1137/080731992

[13] Venmugil Elango, Fabrice Rastello, Louis-Noël Pouchet, J. Ramanujam, and P. Sadayappan. 2014. On Characterizing the Data Movement Complexity of Computational DAGs for Parallel Execution. In *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures* (Prague, Czech Republic) *(SPAA '14)*. Association for Computing Machinery, New York, NY, USA, 296–306. https://doi.org/10.1145/2612669.2612694

[14] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. (2017). http://arxiv.org/abs/1704.04861

[15] Forrest N. Iandola, Matthew W. Moskewicz, Khalid Ashraf, Song Han, William J. Dally, and Kurt Keutzer. 2016. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <1MB model size. (2016). http://arxiv.org/abs/1602.07360

[16] Yangqing Jia. 2014. *Learning semantic image representations at a large scale*. Ph.D. Dissertation. UC Berkeley. https://escholarship.org/uc/item/64c2v6sn

[17] Hong Jia-Wei and Hsiang-Tsung Kung. 1981. I/O Complexity: The Red-Blue Pebble Game. In *Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing* (Milwaukee, Wisconsin, USA) *(STOC '81)*. Association for Computing Machinery, New York, NY, USA, 326–333. https://doi.org/10.1145/800076.802486

[18] Jihyuck Jo, Suchang Kim, and In-Cheol Park. 2018. Energy-Efficient Convolution Architecture Based on Rescheduled Dataflow. *IEEE Transactions on Circuits and Systems I: Regular Papers* 65, 12 (2018), 4196–4207. https://doi.org/10.1109/TCSI.2018.2840092

[19] Jehandad Khan, Paul Fultz, Artem Tamazov, Daniel Lowell, Chao Liu, Michael Melesse, Murali Nandhimandalam, Kamil Nasyrov, Ilya Perminov, Tejash Shah, Vasilii Filippov, Jing Zhang, Jing Zhou, Bragadeesh Natarajan, and Mayank Daga. 2019. MIOpen: An Open Source Library For Deep Learning Primitives. (2019). http://arxiv.org/abs/1910.00078

[20] Grzegorz Kwasniewski, Marko Kabić, Maciej Besta, Joost VandeVondele, Raffaele Solcà, and Torsten Hoefler. 2019. Red-Blue Pebbling Revisited: Near Optimal Parallel Matrix-Matrix Multiplication. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '19)*. Association for Computing Machinery, New York, NY, USA, Article 24, 22 pages. https://doi.org/10.1145/3295500.3356181

[21] Maurice Peemen, Arnaud A. A. Setio, Bart Mesman, and Henk Corporaal. 2013. Memory-centric accelerator design for Convolutional Neural Networks. In *2013 IEEE 31st International Conference on Computer Design (ICCD)*. 13–19. https://doi.org/10.1109/ICCD.2013.6657019

[22] John E. Savage. 1995. Extending the Hong-Kung Model to Memory Hierarchies. In *Proceedings of the First Annual International Conference on Computing and Combinatorics (COCOON '95)*. Springer-Verlag, Berlin, Heidelberg, 270–281.

[23] John E. Savage. 1997. *Models of Computation: Exploring the Power of Computing* (1st ed.). Addison-Wesley Longman Publishing Co., Inc., USA.

[24] Nimish Shah, Paragkumar Chaudhari, and Kuruvilla Varghese. 2018. Runtime Programmable and Memory Bandwidth Optimized FPGA-Based Coprocessor for Deep Convolutional Neural Network. *IEEE Transactions on Neural Networks and Learning Systems* 29, 12 (2018), 5922–5934. https://doi.org/10.1109/TNNLS.2018.2815085

[25] Runbin Shi, Zheng Xu, Zhihao Sun, Maurice Peemen, Ang Li, Henk Corporaal, and Di Wu. 2015. A Locality Aware Convolutional Neural Networks Accelerator. In *Proceedings of the 2015 Euromicro Conference on Digital System Design (DSD '15)*. IEEE Computer Society, USA, 591–598. https://doi.org/10.1109/DSD.2015.70

[26] Karen Simonyan and Andrew Zisserman. 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. http://arxiv.org/abs/1409.1556

[27] Edgar Solomonik, Aydın Buluç, and James Demmel. 2013. Minimizing Communication in All-Pairs Shortest Paths. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. 548–559. https://doi.org/10.1109/IPDPS.2013.111

[28] Christian Szegedy, Sergey Ioffe, and Vincent Vanhoucke. 2016. Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning. (2016). http://arxiv.org/abs/1602.07261

[29] Junmin Xiao, Shigang Li, Baodong Wu, He Zhang, Kun Li, Erlin Yao, Yunquan Zhang, and Guangming Tan. 2018. Communication-Avoiding for Dynamical Core of Atmospheric General Circulation Model. In *Proceedings of the 47th International Conference on Parallel Processing* (Eugene, OR, USA) *(ICPP 2018)*. Association for Computing Machinery, New York, NY, USA, Article 12, 10 pages. https://doi.org/10.1145/3225058.3225140

[30] Junmin Xiao and Jian Peng. 2019. Trade-offs between computation, communication, and synchronization in stencil-collective alternate update. *CCF Transactions on High Performance Computing* 1 (07 2019). https://doi.org/10.1007/s42514-019-00011-x

[31] Xiaoyang Zhang, Junmin Xiao, and Guangming Tan. 2020. Communication Lower Bounds of Convolutions in CNNs. In *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures* (Virtual Event, USA) *(SPAA '20)*. Association for Computing Machinery, New York, NY, USA, 591–593. https://doi.org/10.1145/3350755.3400267

[32] Xiaoyang Zhang, Junmin Xiao, Xiaobin Zhang, Zhongzhe Hu, Hongrui Zhu, Zhongbo Tian, and Guangming Tan. 2019. Tensor Layout Optimization of Convolution for Inference on Digital Signal Processor. 184–193. https://doi.org/10.1109/ISPA-BDCloud-SustainCom-SocialCom48970.2019.00036

[33] Xiaoyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. 2018. ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 6848–6856. https://doi.org/10.1109/CVPR.2018.00716

[34] Jie Zhao and Peng Di. 2020. Optimizing the Memory Hierarchy by Compositing Automatic Transformations on Computations and Data. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 427–441. https://doi.org/10.1109/MICRO50266.2020.00044

[35] Aojun Zhou, Anbang Yao, Yiwen Guo, Lin Xu, and Yurong Chen. 2017. Incremental Network Quantization: Towards Lossless CNNs with Low-Precision Weights. (2017). http://arxiv.org/abs/1702.03044

A Artifact Appendix

A.1 Abstract

A.2 Artifact check-list (meta-information)

- **Program:** Python and C/C++ code
- **Run-time environment:** NVIDIA GPU run-time environment.
- **Hardware:** Any GPU with the compute capability of no less than 3.0.
- **Output:** The time results of auto-tuning engine.
- **How much disk space required (approximately)?:** 20GB
- **How much time is needed to prepare workflow (approximately)?:** 1 hour.
- **How much time is needed to complete experiments (approximately)?:** One week.

A.3 Description

A.3.1 How to access. The artifact provides the source code and installation instructions. The environment can be deployed with them, and then we provide several scripts to verify different experiments with commands.

A.3.2 Hardware dependencies. Any GPU with the compute capability of no less than 3.0

A.3.3 Software dependencies. Generally speaking, the dependencies which mainly come from the official reference of various tools we used can satisfy the basic operating environment, but in order to meet the operating environment better, we also recommend the environment version we are using.

g++ $\geq$ 5 (tested 9.3)
CMAKE $\geq$ 3.5 (tested 3.19.1)
LLVM $\geq$ 4 (tested 8.0)
CUDA toolkit $\geq$ 8
Python $\geq$ 3.6 (tested 3.8)
cuDNN 7.0.3

A.4 Installation

After installed the gcc, LLVM and CUDA toolkit. Run the following command to install the fundamental dependencies:

apt-get install -y python3 python3-dev python3-setuptools gcc libtinfo-dev zlib1g-dev build-essential cmake libedit-dev libxml2-dev

Then, type the command to install the python dependencies:

pip3 install --user typed_ast numpy decorator tornado psutil xgboost attrs

After the dependencies are set up, unpack the source code to the server. Then the following script in source code is helpful for configuring the experimental environment:

./install.sh

A.5 Experiment workflow

For the convenience of the artifact evaluation, we provide a series of shell scripts which run the different tests we have described in the paper and print the result after parsing on screen. After the artifact is properly installed, the experiments can be run as follows.

1. Enter the test directory:
   cd $Testdir/test/
2. Using scripts to run tests, the file *test_all.py* contains different test located in *test_src/scripts* . To run different test, using the command as follows:
   python3 test_all.py --options=[param]

Corresponding to the data of the paper, there are the following options for *param*:

Figure 8: *paper* **or** *paper_all*
Figure 9: *batch* **or** *batch_all*
Figure 11: *end2end* **or** *end2end_all*
Figure 12: *multi_platform*
Table 2: *comp* **or** *comp_all*

Different options call different test scripts. For *paper** and *batch**, they call the *test_1.py* and *test_2.py* in *test_src/scripts*. For *end2end**, the *end2end.py* in *test_src/cfg4paper* is called. As for *comp**, it calls both *test_*.py* and *test_original_*.py* located in *test_src/scripts* as well as *multi_platform* does. After these python files are called, the template in *cfg4paper* directory will be used. As for the test of other arbitrary parameters, we also provide a customization way in the section A.7.

A.6 Evaluation and expected results

For options without the *all* tag, the expected results are the minimized outputs of the corresponding figure with time (only one set of parameters for each figure).

For options with the *all* tag, the expected results are the full tests of the corresponding figure with time. (Throughput metrics are also included.)

A.7 Experiment customization

For customizing the tests with specific parameters, the *test_1.py* or *test_2.py* which located in *test_src/scripts* is useful. The tests can be run with the parameters *-N [batch size]-H [input height] -W [input width] -CO [output channel] -CI [input channel] -KH [kernel height] -KW [kernel width] -strides [strides] -padding [padding]*. They provide an entrance to call the template implemented with the aid of theory located in *tvm/topi*. If you want to test the end to end results with customization, the *test_src/cfg4paper/end2end.py* provides the entrance to call the tuning process which the templates are implemented through the theory.