Characterizing and Demystifying the Implicit Convolution Algorithm on Commercial Matrix-Multiplication Accelerators

Yangjie Zhou[1,2], Mengtian Yang[1], Cong Guo[1,2], Jingwen Leng[1,2], Yun Liang[3,2], Quan Chen[1,2], Minyi Guo[1,2], Yuhao Zhu[4]
[1]*Shanghai Jiao Tong University*, [2]*Shanghai Qi Zhi Institute*, [3]*Peking University*, [4]*University of Rochester*

***Abstract*—Many of today's deep neural network accelerators, e.g., Google's TPU and NVIDIA's tensor core, are built around accelerating the general matrix multiplication (i.e., GEMM). However, supporting convolution on GEMM-based accelerators is not trivial. The naive method explicitly lowers the convolution to GEMM, commonly known as `im2col`, which introduces significant performance and memory overhead. Existing implicit `im2col` algorithms require unscalable hardware and are inefficient in supporting important convolution variants such as strided convolution. In this paper, we propose a memory-efficient and hardware-friendly implicit `im2col` algorithm used by Google's TPU, which dynamically converts a convolution into a GEMM with practically zero performance and memory overhead, fully unleashing the power of GEMM engines. Through comprehensive experimental results, we quantitatively argue that this algorithm has been adopted in commercial closed-source platforms, and we are the first to describe its high-level idea and implementation details. Finally, we show that our algorithm can also be generally applied to Nvidia's Tensor Cores (TC), matching and out-performing the measured performance on TCs.**

I. Introduction

The recent development of convolutional neural network (CNN) models [1] has lead to its wide adoption in many fields such as autonomous driving [2]–[4] and natural language processing [5]–[7]. Yet, many commercial neural network accelerators, such as Google's TPU [8], NVIDIA's Tensor Cores (TCs) since the Volta architecture [9], Habana Gaudi [10], and Intel's NNP-T [11], choose the general matrix-matrix multiplication (GEMM) as the basic computation primitive.

It is non-trivial to support CNNs on the GEMM-specialized accelerator. Many recent works [12]–[14] make the assumption of explicit `im2col` (image-to-column) algorithm, which lowers the convolution to a matrix multiplication via input transformation. The naive approach performs an *explicit* `im2col` transformation to prepare the lowered feature map in the form of the expanded matrix. As such, this matrix can be consumed directly by the GEMM engine without any hardware modifications. This explicit `im2col` transformation leads to significant performance and memory overheads.

Commercial GPUs adopt the *implicit* `im2col` algorithm [15] to avoid the performance and memory overheads in the explicit algorithm. However, the exact implicit algorithm is not published and it is unclear how to implement it on GEMM-based accelerators like TPUs. In this work, we study the only described implicit `im2col` method in the public domain [16]. We find that it requires an unscalable hardware design (heavily-banked memory with a large crossbar) for porting to the TPU, and is also inefficient in executing common `CONV` variants such as strided and dilated `CONV` [17].

In this paper, we demystify a hardware-friendly and memory-efficient implicit `im2col` algorithm used by the TPU, which dynamically converts a convolution into a GEMM with practically zero performance and memory overhead, fully unleashing GEMM engines' power. Such an implicit algorithm leverages the associativity and commutativity in convolution, and requires the memory layout and tiling strategy optimization. As such, the GEMM engine in the TPU is served with data from a simple single-bank memory while allowing off-chip memory access and computation to be fully overlapped.

We develop a configurable cycle-level TPU simulator for performance evaluation. Our simulator is validated against the cloud TPUv2 measurement and has an average error rate of less than 5%. We plan to make the simulator open-source to encourage more study.

In addition to the TPU, we also show that our implicit `im2col` can also be applied to the TCs on the GPU. The challenge is to maximally utilize the many TCs on the GPU. To that end, we devise a blocked version of our `im2col` algorithm. We exploit tile reordering to avoid stalling from off-chip memory accesses. We implement and evaluate our algorithm on a V100 GPU. The difference in performance between us and cuDNN is 1% at batch size of 8.

To our best knowledge, we are the first in the public domain to introduce a generic implicit `im2col` algorithm that is implemented on both a systolic array and the TCs. We do not know (nor claim) whether (part of) our design is implemented in TPU or the TCs in Nvidia's GPUs. That said, we present our educated guess as to what part of our design is likely implemented in the TPU and/or the TCs, and why our design achieves higher performance than the proprietary TPU and GPU designs in certain scenarios.

In summary, the paper makes the following contributions:

- We quantify the performance and memory overhead of explicit `im2col` method over implicit `im2col` method.
- To our best knowledge, we study the first open, public design of implicit `im2col`, which is generally applicable to GPUs and systolic array-like accelerators (e.g., TPUs) with zero memory and near-zero performance overhead.
- We implement two concrete instances of the above implicit `im2col` method on the commodity TPUs and GPUs, via simulation and software implementation respectively. We show that our methods are on-par with and sometimes even better than the vendor's proprietary implementations.
- All the artifacts are made available. We hope our design can shed some light upon, and identify potential room for improvement of the proprietary designs.

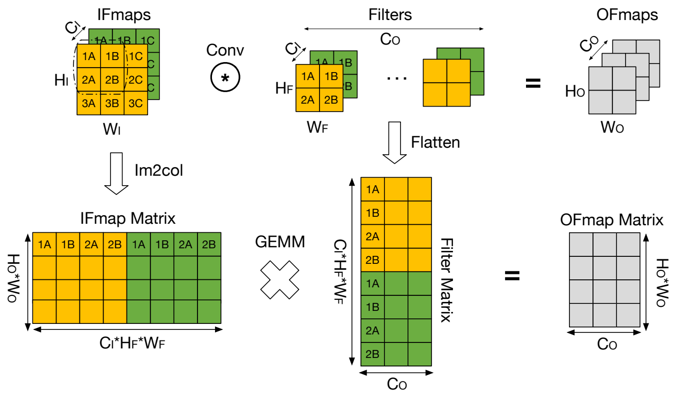


Fig. 1: Illustration of the im2col algorithm that converts a CONV layer to a GEMM operation. For simplicity, this examples assumes no padding in the IFMap.

We organize the paper as follows. Sec. II introduces the background on im2col and quantifies the inefficiencies of existing im2col methods. Sec. III describes our im2col methods. We then describe how to implement and optimize our im2col on the TPU (Sec. IV) and the TCs on the GPU (Sec. V). After the experimental methodology (Sec. VI), we evaluate our im2col designs on the TPU and the GPU (Sec. VII). Sec. VIII describes the related work and Sec. IX concludes the paper.

II. Motivation

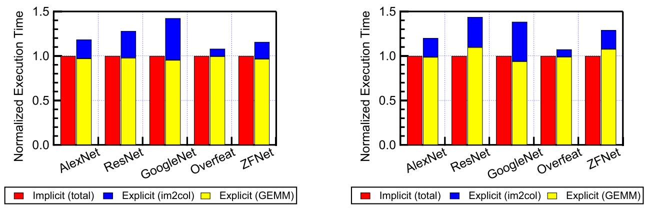
This section first introduces the background on im2col (Sec. II-A). We then quantitatively demonstrate that explicit im2col is both memory inefficient and slow (Sec. II-B). However, existing implicit im2col implementations have inherent limitations and/or are proprietary (Sec. II-C).

A. Background on Im2Col

Many commercial hardware accelerators for deep neural network (DNN), such as Google's TPU [8], NVIDIA's Tensor Cores since the Volta architecture [9], Habana's Gaudi [10], and Intel's NNP-T [11], choose the general matrix-matrix multiplication (GEMM) as the basic computation primitive. The reason is that most of the operations in DNN models, including the fully connected and convolutional layer, can be directly mapped or lowered to GEMMs.

Convolution is still an important and fundamental workload, e.g., Google has acknowledged that the portion of CNNs in its data centers increases from 5% to 24% [18]. The upper part of Fig. 1 shows the details of a CONV layer. It takes in a number of C_I input feature maps (or input channels), each sized of $H_I \times W_I$. Every input feature map is convolved by a sliding kernel (or weight) size of $H_F \times W_F$ to calculate one pixel in the output feature map. A total of C_O feature maps (or output channels) will be generated as output to the next layer. In many settings (e.g., training), a batch of N inputs can be executed in parallel to amortize the cost of weight accesses.

However, lowering DNN operations to GEMM is not automatic. im2col (image-to-column) is the de facto algorithm used to lower a convolution to a GEMM [19], [20]. The bottom



(a) NVIDIA V100 GPU. (b) Google TPU-v2.

Fig. 2: The execution time comparison of explicit and implicit im2col methods for convolutional layer on V100 GPU and TPU-v2. We use a batch size of 64 for all CNNs.

of Fig. 1 shows an example of this transformation. The $(H_I \times W_I \times C_I)$ IFMap is first expanded into a $(H_O W_O \times H_F W_F C_I)$ matrix, which we call the *lowered* feature matrix. Each row in the lowered matrix corresponds to the receptive field of an element in the OFMap, as Fig. 1 shows. The filters are then flattened to a matrix with the size of $(C_I H_F W_F \times C_O)$.

B. The Need for Implicit Im2Col

The naive approach performs an explicit im2col transformation to prepare the lowered feature map before the latter is consumed by the GEMM engine (e.g., the Tensor Cores in Nvidia's GPUs or the TPU). This explicit im2col transformation leads to significant performance and memory overheads over the GEMM computation itself.

Our measurements show that neither Nvidia's GPUs nor the TPU uses explicit im2col (although the option is available on Nvidia's GPUs), presumably because of the high overhead. Instead, their proprietary implementations, which we call the *implicit* im2col, show little overhead. We quantitatively demonstrate the inefficiencies of explicit im2col to motivate implicit im2col. To our best knowledge, no such analysis is available for TPU, nor other GEMM accelerators.

Memory Overhead The lowered feature matrix from the im2col transformation takes up to $H_F \times W_F$ times more memory than the original feature map, because the overlapped receptive fields generate duplicated data.

We use GPU as a case study to demonstrate the overhead. Tbl. I compares the memory required for storing the lowered input matrix in the explicit im2col method across a range of different models. As a reference, we also show the original size of the input feature (IFMap). The data is measured on a V100 GPU using the explicit im2col APIs from the cuDNN library [20]. The additional storage requirement is generally $1.5\times - 10\times$ of the input feature maps.

Performance Overhead Explicit im2col also introduces significant performance overhead. To demonstrate this, we measure the GPU execution time of both the implicit and

TABLE I: Memory usage (MB) breakdown for executing different CNNs with the explicit im2col on V100 GPU.

	AlexNet	ResNet	VGG16	YOLO	DesNet
IFmaps	1.39	34.55	34.65	530.56	1196.48
Lower IFmaps	14.57	81.11	311.80	869.50	5641.70

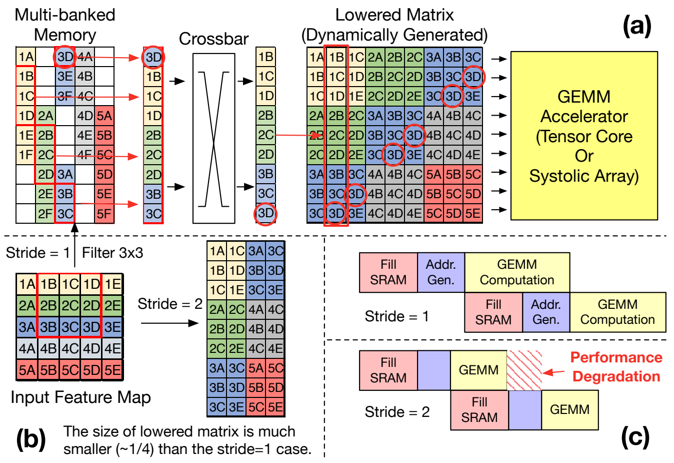


Fig. 3: (a) The existing implicit `im2col` design [16] dynamically forms the lowered IFMap, assuming a stride of 1. It has little performance overhead, but requires a multi-banked on-chip memory and a large crossbar between the memory and the GEMM engine. (b)&(c) The $stride = 2$ causes throughput degradation: the latency to fill in the on-chip memory could not be hidden by the GEMM latency, which is about 1/4 of the $stride = 1$ case.

explicit versions using their corresponding cuDNN APIs. The stacked bars in Fig. 2a break down the execution time on the V100 GPU into the GEMM time and the explicit `im2col` transformation time (batch size 64). The execution time is normalized to the execution time of the implicit method.

We find that explicit `im2col` is 28% slower than the implicit approach on average. Critically, the GEMM time in the explicit method is almost identical to the implicit method. This suggests that the implicit method has near-zero performance overhead: all its time is spent on GEMM. In other words, the explicit method introduces a performance overhead of about 26% compared to the GEMM on GPU.

The conclusion holds on the cloud TPU-v2 platform [21]. Fig. 2b compares the execution time of both methods on TPU-v2. Note that the TPU does not provide the explicit option; we thus mimic the behavior of explicit `im2col` as if it was to be used on the TPU by combining the GEMM time on TPU and the time of explicit `im2col`, which is estimated by using the GPU results. This strategy provides a performance *lower bound* for the explicit `im2col` if it was supported on TPU, because we omit the overhead of transmitting the lowered matrix to the TPU.

On average, the explicit `im2col` transformation introduces 26% overhead. The explicit method is 23% slower than the implicit method, whose execution time is roughly the same as the GEMM time in the explicit method, indicating little performance overhead of `im2col` in the implicit method.

C. Limitations of Existing Implicit Approach

Characterization results suggest that both GPU and TPU use some forms of implicit `im2col` method. However, their implementations are proprietary. We describe a prior academic effort based on Lym et al. [16], a conceptually clear design to support implicit `im2col` for GPUs' CUDA Core. We migrate this design to Tensor Core. The difference between the two computational patterns is that for the CUDA core, the warp-level GEMM is computed via outer product, while the TensorCore computes GEMM via inner product [22]. We do not claim (nor know) whether the design by Lym et al. is the same as that in GPUs, but we show that today's GPUs suffer from some of the similar inefficiencies to that of Lym et al. Thus, Lym et al. would provide a sensible design to understand the inefficiencies in existing implicit `im2col` implementations.

The basic idea of Lym et al. [16] is to use a flexible on-chip memory structure to dynamically form the lowered feature matrix before the latter is fed into the compute engine. Fig. 3 illustrates the details. The input feature map (IFMap) is stored in the on-chip SRAM (e.g., the shared memory in the GPU). Each element in the IFMap is dynamically routed to the correct PE in the GEMM engine, effectively forming the lowered IFMap (middle matrix) at run time.

The advantage of Lym et al. is two-fold. First, the lowered IFMap does not require additional storage because it is dynamically formed and immediately consumed. Second, it can sustain the full throughput of the GEMM engine. This is because filling the on-chip memory for the next block/tile (through accessing DRAM) and GEMM computation can be overlapped, as shown in the lower right panel in Fig. 3.

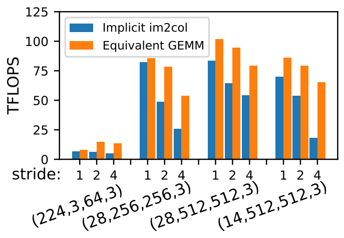
However, the hardware design by Lym et al. (and today's GPUs) does not generally scale to other forms of GEMM accelerator and/or incurs significant performance overhead for common convolution variants such as strided and deformable convolution [23]. Let us elaborate below.

Unscalable Hardware The main requirement of prior work is a multi-banked SRAM with a large crossbar, which routes elements in the IFMap (stored in the SRAM) to the correct PEs in the GEMM engine at each cycle. Consider the second column in the lowered matrix in Fig. 3. All 9 elements that enter the GEMM engine in one cycle have to come from different banks of the SRAM in order to not stall the GEMM engine. The bank conflict can be avoided by carefully laying out the IFMap elements in the SRAM offline [16].

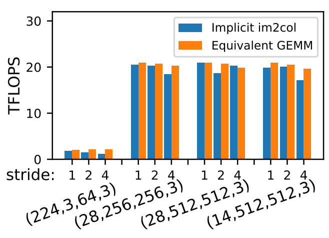
Critically, each element in the SRAM needs to be mapped to different PEs at different cycles, entailing a crossbar. E.g., the element 3D at the current cycle maps to the PE in the last row, but will map to the second last row next cycle.

Lym et al. made an astute observation that modern GPUs naturally provide such a multi-banked SRAM (i.e., the shared memory with 32 banks) and a crossbar (i.e., the 32×32 crossbar for shuffling data within a warp) in each SM. Therefore, this implicit `im2col` design introduces little additional hardware on top of existing GPU hardware.

However, this design is unscalable, because the size of the crossbar and the number of banks in the SRAM would have to scale proportionally to the PE array size in the GEMM engine. For instance, TPUv1 [8] uses a 256×256 PE array, requiring a 256×256 crossbar and a 256-bank SRAM. The crossbar area and power increase quadratically with respect



(a) NVIDIA V100 GPU.



(b) Google TPU-v2.

Fig. 4: TFLOPS of implicit im2col on representative ResNet layers (represented by W_I, C_I, C_O, W_F) under different strides. As a reference, we also show the TFLOPS of GEMM only. GPU performance (a) significantly degrades with larger strides, whereas the TPU (b) is insensitive to the stride.

to the number of ports [24], and a large number of memory banks also degrades the area and power efficiency [25].

Supporting CONV Variants The performance of the existing implicit im2col approach degrades significantly for key convolution variants such as the strided convolution, which are commonly used in modern CNNs [26].

Fig. 4a shows the performance measured in TFLOPS (Tera Floating Point Ops per Second) of several representative layers of ResNet [27] under different strides on the Tensor Cores of V100 GPU. Compared with the stride as 1, the GPU performance drops by 30% under a stride of 2 and 60% under a stride of 4. To understand the performance drop, Fig. 4a also shows the TFLOPS of a GEMM kernel operating on a matrix of the same size as the lowered IFMap. The GEMM's TFLOPS is much higher than that of the implicit im2col under larger striders, indicating the implicit method becomes severely memory-bound with a greater-than-one stride.

Implicit im2col becomes memory-bound under larger strides because the GEMM latency (of a tile) reduces; thus, the latency of filling the on-chip SRAM could not be hidden by the GEMM computation. The bottom half of Fig. 3 illustrates the reason. With $stride = 1$, the address generation overhead to access the multi-banked memory (Fig. 3 top) makes the overall performance roughly the same as the equivalent GEMM computation in Fig. 4. This indicates the GEMM tile latency in Fig. 3 bottom can just overlap with the SRAM filling and address generation process together. However, with $stride = 2$, the size of the lowered matrix is reduced significantly (about 1/4 for the stride of two). Thus, the GEMM latency is reduced significantly while the SRAM loading time does not change, leading to large performance degradation.

TPU does *not* show the same performance degradation. Fig. 4b shows the results of the same experiment but on TPU. The results demonstrate that TPU performance is insensitive to the stride value. The difference between the TPU and the GPU suggests that the TPU and the GPU potentially use different implicit im2col designs — there is a design space for implicit im2col that has not been explored in prior work.

Summary Existing implicit im2col design in the public domain requires a complicated, unscalable hardware design and is inefficient in handling common convolution variants. We observe that some of these issues exist in the proprietary designs from commercial accelerators as well.

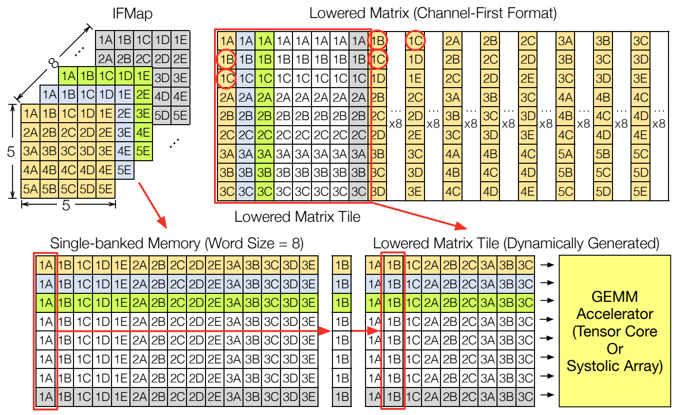

IFMap
Lowered Matrix (Channel-First Format)
8
5
5
x8
Lowered Matrix Tile
Single-banked Memory (Word Size = 8)
Lowered Matrix Tile (Dynamically Generated)
GEMM Accelerator (Tensor Core Or Systolic Array)

Fig. 5: We store the IFMap with the *HWC* format in the on-chip SRAM. This allows the GEMM to read data from the SRAM as a wide word rather than from different banks, simplifying the hardware.

We study a previously unpublished implicit im2col algorithm that addresses both issues. Our real hardware measurement results show that this algorithm is highly possible used in the TPU architecture. We also demonstrate its general feasibility and advantage on the tensor core GEMM accelerator.

III. Implicit Channel-First Im2col

In this section, we explain a powerful im2col algorithm called channel-first im2col method, which avoids the two sources of inefficiency associated with today's im2col algorithm. We first present the basic idea that allows us to use a simple crossbar-free, single-bank on-chip memory to serve IFMap to the GEMM engine (Sec. III-A). We show that the basic idea can be trivially extended to balance the speed of GEMM computation and filling the SRAM, eliminating the inefficiencies in computing CONV variants (Sec. III-B).

A. Channel-First Im2col Method

The basic idea of our approach is to layout the IFMap in the on-chip SRAM in such a way that each IFMap element is sent to a deterministic PE. In this way, we can avoid the costly multi-banked SRAM and the associated crossbar.

We will first use a concrete example to describe how such an IFMap layout looks and why such a layout ensures correctness. We will then explain the general principle behind such a layout, which is conceptually nothing more than simply reordering elements in the lowered IFMap matrix.

On-chip SRAM Data Layout Assuming an IFMap with 8 channels (C_I), each with the dimension 5×5 ($H_I \times W_I$), Fig. 5 shows our proposed IFMap layout in the SRAM, where each row is an unrolled vector of a channel in the IFMap. In other words, each column consists of elements of the same position across channels. For instance, the 8 elements in the first column are the elements at position 1A from all 8 channels. Each cycle, one column is read out as a whole word from the SRAM, and fed into the GEMM engine.

We call this is the *Channel-First*, or the HWC layout, since the channel dimension is unrolled first. Note that when tiling

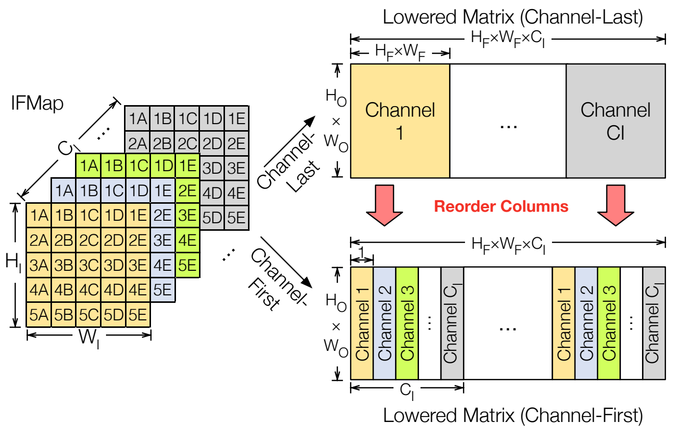


Fig. 6: The channel-last and -first layout for the lowered matrix.

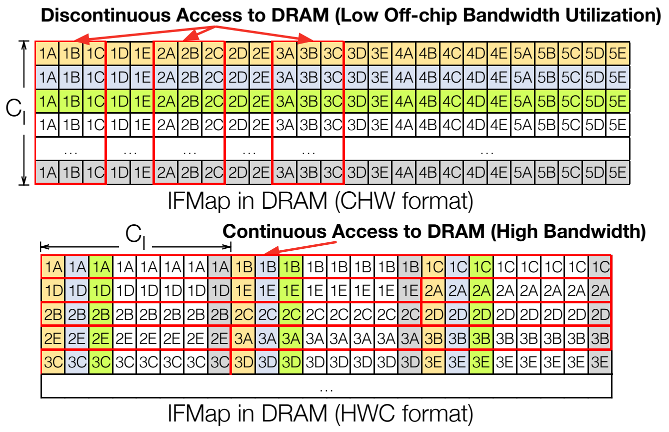

Discontinuous Access to DRAM (Low Off-chip Bandwidth Utilization)
C_I
IFMap in DRAM (CHW format)
Continuous Access to DRAM (High Bandwidth)
IFMap in DRAM (HWC format)

Fig. 7: Advantage of *HWC* format over commonly used *CHW*.

is applied to the IFMap, as is commonly done when the entire IFMap size is too large for the SRAM, only a tile of IFMap is stored in the SRAM at a time *using this layout*.

With such an SRAM layout, let us explain how to lower a 3×3 CONV layer to a GEMM. In the first 9 cycles, *columns* of `1A, 1B, 1C, 2A, 2B, 2C, 3A, 3B, 3C` are read out from the SRAM, one column (word) per cycle. These 9 columns correspond to the first sliding window in the IFMap that the filter operates on (assuming no padding). In the next 9 cycles, columns corresponding to the next sliding window (with a stride of 1), i.e., `1B, 1C, 1D, 2B, 2C, 2D, 3B, 3C, 3D`, are read out from the SRAM and fed to the GEMM engine, one column at a cycle. This process repeats until all the sliding windows in the current IFMap tile resident in the SRAM finish, at which point the next tile starts until all the IFMap tiles finish.

Critically, while each IFMap element is read multiple times, it is deterministically sent to one fixed PE throughout the execution. For instance, all the `1C` elements are read three times, but they are always read as a whole word and sent to the corresponding PEs each time. In this way, the entire on-chip SRAM can be organized as a single bank with a word size of 8 elements (e.g., 8 Bytes if INT8 is used). Note that this requires the channel size to be a multiple of word size, the implication of which is discussed in Sec. IV.

DRAM Layout To maximize the DRAM bandwidth utilization when loading the IFMap data to the SRAM with the HWC layout, we propose to store the IFMap in the DRAM using the HWC format instead of the commonly used CHW format. Fig. 7 compares the access patterns to produce the lowered matrix tile example in Fig. 5 between the two DRAM layouts.

The access pattern to the CHW-based IFMap contains discontinuous accesses (e.g., `1A-1C`, `2A-2C`, `3A-3C` in the first channel) to the DRAM, which under-utilizes the off-chip memory bandwidth and increases the SRAM filling latency. In contrast, the access pattern to the HWC-based IFMap contains mostly continuous accesses (e.g., eight channels of `1A` to `1C`), which better utilizes the off-chip bandwidth.

Note that when stride=1 the gap between the CHW and HWC format is small because the W-dimension (e.g., 128) is typically large enough for using the DRAM bandwidth. However, the HWC format is critical to resolve the performance issue of larger strides, as we explain later.

General Principle Inherently, the above approach simply changes the way lowered IFMap elements are arranged in existing `im2col`. Fig. 6 illustrates the difference between the channel-first method and existing channel-last method. Note that this reordering is conceptual, as the lowered IFMap never physically exists — it is dynamically generated and consumed.

Recall from Fig. 1 that in `im2col` the IFMap is converted to a $[H_O W_O \times H_F W_F C_I]$ lowered IFMap. In the existing channel-last `im2col` approach, the $H_F W_F C_I$ dimension of the lowered IFMap is expanded in the $C_I \rightarrow H_F \rightarrow W_F$ order, which stores the $H_F \times W_F$ elements in a sliding window across all channels sequentially. In contrast, our approach constructs the $H_F W_F C_I$ dimension in the $H_F \rightarrow W_F \rightarrow C_I$ order, which stores elements of the same position across the C_I channels sequentially. Intuitively, our lowered IFMap simply shuffles the columns of the lowed IFMap in the existing method.

In this sense, the correctness of the channel-first method can be understood as: changing the column order in a matrix does not change the result of GEMM (so long as the other matrix elements are reordered accordingly).

B. Supporting Conv Variants

We will first describe a particularly useful way of understanding the proposed channel-first `im2col`, using which we then explain how this algorithm can be trivially extended to support CONV variants, such as the strided convolution, which are inefficient using existing implicit `im2col` method.

Decomposed 1×1 CONVs The studied implicit `im2col` method essentially decomposes the $H_F \times W_F \times C_I$ filter into $H_F \times W_F \times C_I$ 1×1 filters (and properly accumulating the partial sums), because each column of data that enters the GEMM engine consists of elements from the same position across all channels, effectively performing 1×1 CONVs.

Critically, the 1×1 CONVs can be performed in an arbitrary order due to the commutativity of accumulation. We propose to iterate over the decomposed filters (as opposed to over sliding windows). We first compute the 1×1 CONVs associated with

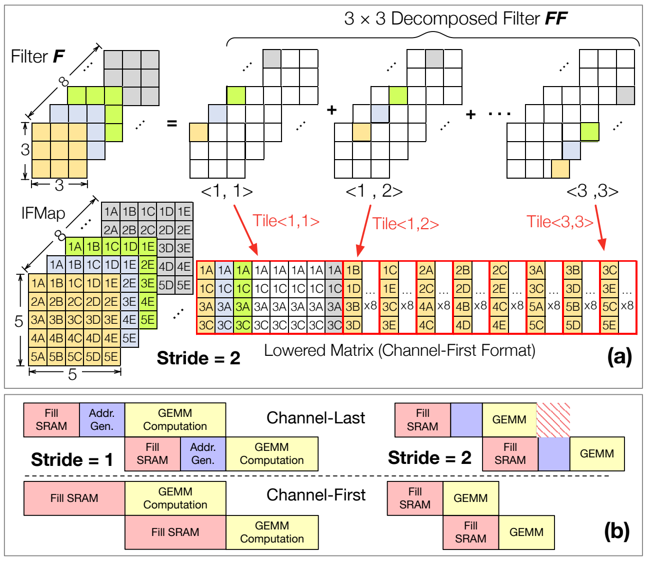


Fig. 8: Example for supporting CONVwith a stride of two. (a) We tile the lowered matrix according to the filter decomposition, where each tile corresponds to a 1×1 CONV layer. (b) Our channel-first `im2col` is insensitive to stride because it requires simple address generation and the on-chip SRAM filling time also reduces as the stride increases from 1 to 2.

the first decomposed filter and then move to the 1×1 CONVs associated with the next decomposed filter, and so on. Consider the example in Fig. 8a. The first decomposed filter $\langle 1,1 \rangle$ convolves with IFMap data at `1A, 1C, 3A`, and `3C`; the next decomposed filter, say, $\langle 1,2 \rangle$ convolves with IFMap data at `1B, 1D, 3B`, and `3D`. The lower-right corner in Fig. 8a shows the corresponding lowered IFMap.

The channel-first algorithm is naturally insensitive to the stride size, because the SRAM filling latency and the GEMM latency decrease simultaneously and proportionally for each tile. Fig. 8b illustrates the difference between the existing channel-last method and our channel-first method as the stride changes from 1 to 2. The performance of the existing method degrades because the GEMM latency of a tile decreases but the SRAM filling latency remains the same. In the channel-first method, however, when the stride decreases, the size of each lowered IFMap tile proportionally decreases, e.g., from 9×8 (Fig. 5) to 4×8 (Fig. 8a). Thus, the GEMM latency can still hide the SRAM filling latency, without hurting performance.

IV. Support for Systolic Array

In this section, we describe how to support the proposed implicit channel-first `im2col` method on systolic array using Google's TPU architecture as the base design. We first describe the basic idea of mapping the channel-first `im2col` algorithm to the TPU (Sec. IV-A), followed by optimizations that further improve the mapping efficiency (Sec. IV-B).

Our measurement results show that it's very likely that the TPU adopts the same algorithm and same set of optimizations. To our best knowledge, we are the first to show how to support implicit `im2col` in a systolic array in the public domain.

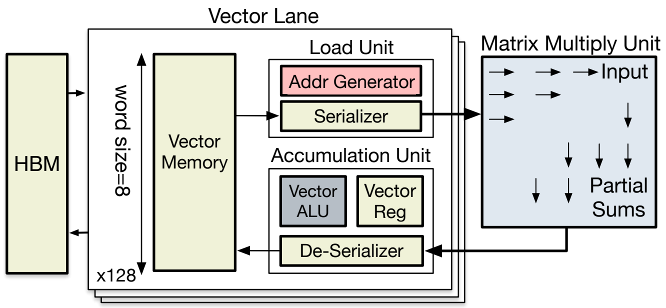


Fig. 9: The simplified TPU-v2 core architecture details [18], [28].

That said, we will briefly discuss why we suspect that the TPU design is similar to what we describe here, and how the baseline design needs specific optimizations for the higher performance in certain scenarios.

A. *Basic Algorithm Mapping*

Baseline TPU We model the baseline systolic array architecture after the published details of Google's TPU-v2/v3 architecture [18], [28]. In particular, we consider a baseline dual-core TPU-v2. TPU-v3 is similar in the core microarchitecture but uses different core numbers. Fig. 9 shows the details of a single TPU core, which contains a 128×128 weight stationary systolic array for GEMM. We describe the architectural details that are most relevant to implementing our `im2col` algorithm.

Different from the separate buffer design in TPU-v1 [8], TPU-v2 uses a unified on-chip SRAM for storing IFMap, weights, and OFMap. The unified SRAM is split into 128 different SRAM arrays, each dedicated to exchanging data with *a fixed PE row in the systolic array*. That is, this is *not* a 128-bank SRAM with a 128×128 cross-bar, but 128 separate SRAM arrays, each with a single read/write port.

The TPU-v2 chooses a word size of eight for each SRAM array, which is thus called a vector memory in Fig. 9. For instance, if FP16 or BFloat16 is to be used, each access to a vector memory would be reading/writing 16 bytes.

Implementing Implicit im2col The key challenge of implementing the aforementioned channel-first implicit `im2col` method on the TPU is that the TPU, as a systolic array, has a time-delayed data access pattern. One naive way to address it would be to skew the data layout in the SRAM and DRAM described in Sec. III. However, it would lead to frequent skewing and restoring for other non-GEMM layers such as pooling and batch normalization [29].

Instead, the channel-first implicit `im2col` can be trivially mapped to the TPU by leveraging the unique SRAM organization of the TPU, which uses 128 independent SRAM arrays, each for a PE row. We simply map each row of our proposed *HWC* SRAM layout (Fig. 8a) to an SRAM array. Naturally, each SRAM array stores a single channel in the IFMap. Since each SRAM array is independent, we would simply use an address generation logic to generate the appropriate address to each SRAM array such that the 128 addresses are skewed by one cycle to fit the systolic data flow. That is, instead

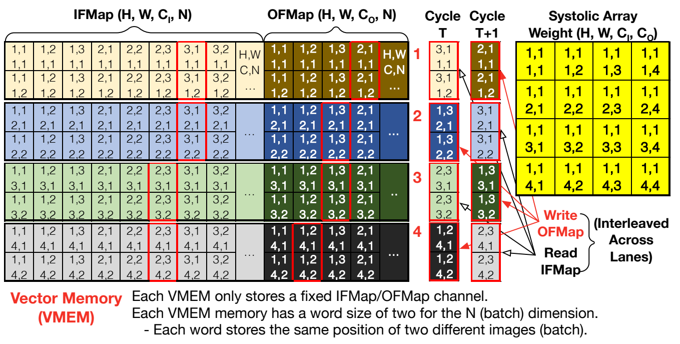

IFMap (H, W, C_I, N)
OFMap (H, W, C_O, N)
Cycle T
Cycle T+1
Systolic Array Weight (H, W, C_I, C_O)
H,W C,N
Write OFMap
Read IFMap
(Interleaved Across Lanes)
Vector Memory (VMEM)
Each VMEM only stores a fixed IFMap/OFMap channel.
Each VMEM memory has a word size of two for the N (batch) dimension.
- Each word stores the same position of two different images (batch).

Fig. 10: The example of executing the tile $\langle 1,1\rangle$ for IFMap $N = 2, C_I = 4, H_I = C_I = 5$, filter $H_F = W_F = 3$, OFMap ($H_O = C_O = 3$) on the 4×4 weight stationary systolic array with 4 separate SRAMs. Each SRAM has a word size of 2, and stores both IFMap and OFMap.

of skewing the data layout, we skew the address generation, which is enabled by the SRAM organization in the TPU.

Leveraging Large Word Size Each of 128 SRAM arrays in the TPU has a word size of 8. Therefore, each time we read from an SRAM array, it would return 8 data elements (whether it is FP16 or BFloat16). To utilize the large word size, our idea is to put different inputs from the same batch into the same SRAM array. That is, each SRAM array would store a channel of the IFMap from 8 different inputs. We call this new data layout *HWCN*, where N denotes the batch dimension.

A potential issue with the large word size is that, while an SRAM array can produce 8 data elements, each PE in the GEMM engine would accept only one data element per cycle. Therefore, we propose to use a buffer (the serializer in Fig. 9) in-between the SRAM array and the GEMM engine to hold all the 8 data elements read at once; the serializer would then issue one element to the GEMM engine each cycle. Accordingly, we would read data from each SRAM array and update the serializer only once every 8 cycles, reducing the SRAM switching activity.

Leveraging the Unified Memory The on-chip memory in the TPU is unified in that it stores both IFMap and OFMap. As a result, storing to the OFMap and reading from the IFMap will contend the same port of each SRAM array. We address this issue by leveraging the fact that each SRAM array is read only once per 8 cycles, which allows us to interleave storing the OFMap data to an SRAM array with the load, effectively posing zero contention/overhead. For implementation, we add a de-serializer for each vector memory in Fig. 10, which receives the result from the systolic array every cycle and writes to the vector memory every 8 cycles.

Example Without loss of generality, we use a small working example in Fig. 10 to illustrate the on-chip vector memory layout and its interaction with the systolic array when executing the tile $langle1,1\rangle$ for the filer size of 3×3. The indices in a vector memory element indicate the height, width, channel, and batch, correspondingly. Each column in a vector memory is a word, e.g., $(3,1,1,1)$ and $(3,1,1,2)$ in the first vector memory. Because we are currently executing the GEMM for tile $\langle 1,1\rangle$, we only need to store each $\langle 1,1\rangle$ position of the

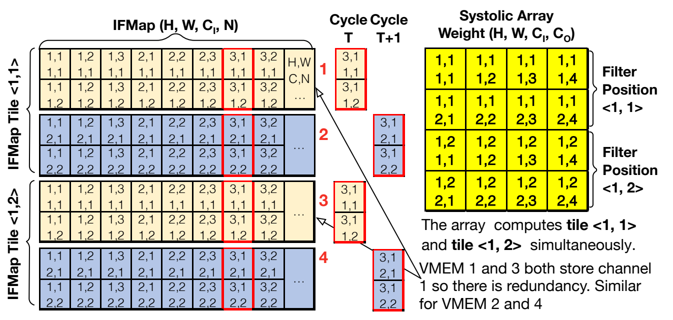

IFMap (H, W, C_I, N)
Cycle T
Cycle T+1
Systolic Array Weight (H, W, C_I, C_O)
IFMap Tile <1,1>
IFMap Tile <1,2>
H,W C,N
Filter Position <1, 1>
Filter Position <1, 2>
The array computes tile <1, 1> and tile <1, 2> simultaneously.
VMEM 1 and 3 both store channel 1 so there is redundancy. Similar for VMEM 2 and 4

Fig. 11: The example of multi-tile computation (tile $\langle 1,1\rangle$ and $\langle 1,2\rangle$) for CONV with $N = 2, C_I = 2, H_I = C_I = 5, H_F = W_F = 3, H_O = C_O = 3$, which doubles the array utilization compared to the single-tile computation. It leads to input duplication (e.g., $2\times$) in the SRAM.

$C_I \times C_O$ (i.e., 4×4) filters in the systolic array.

Because this example has a word size of two, each vector memory is read for IFMap or written for OFMap every two cycles in an interleaved fashion. For example, at the cycle T, the first and third row of the systolic array reads a word from the corresponding vector memory, while the second and fourth column (the partial sum results comes from the bottom PEs in Fig. 9) writes to the vector memory. At cycle $T+1$, each vector memory switches to read or write.

B. Optimizations

Limitations The design described above becomes inefficient when the input channel size is small, e.g., 3 in the first layer of today's CNNs. This is because each PE row in the systolic array reads from one SRAM array, which stores one input channel in the IFMap. An IFMap channel of 3 leaves 125 PE rows idle, i.e., leading to a severe array under-utilization.

As a result, the design described before requires the channel size to be padded to 128, respectively. We introduce one nifty optimization that avoids these wastes.

Multi-tile Computation To mitigate array under-utilization and hence performance loss when the input channel is small, we propose to fill the vector memories with data from other tiles (yet to be computed), essentially concurrently computing multiple tiles. For instance, the original method in Fig. 10 only computes a single tile $\langle 1,1\rangle$ out of the total $H_F \times W_F$ tiles. When the channel size is small, say half of the array size, we can compute the tile $\langle 1,1\rangle$ and $\langle 1,2\rangle$ simultaneously.

Fig. 11 shows an example when IFMap channel size C_I is 2 and the array size is 4. The systolic array stores all the weight elements from the $\langle 1,1\rangle$ and $\langle 1,2\rangle$ positions of 2×4 (i.e., $C_I \times C_O$) filters, and therefore computes the feature matrix tile $\langle 1,1\rangle$ and $\langle 1,2\rangle$ at the same time. This optimization is essentially merging the two tiles to form a larger tile so the correctness is guaranteed by the associativity of GEMM.

This optimization does not require any hardware modification as its only difference from the single-tile computation is the vector memory filling and address generation, which can be supported via instructions. Meanwhile, the tile data replication is performed before IFMaps flow into the systolic array, so no additional synchronization is needed.

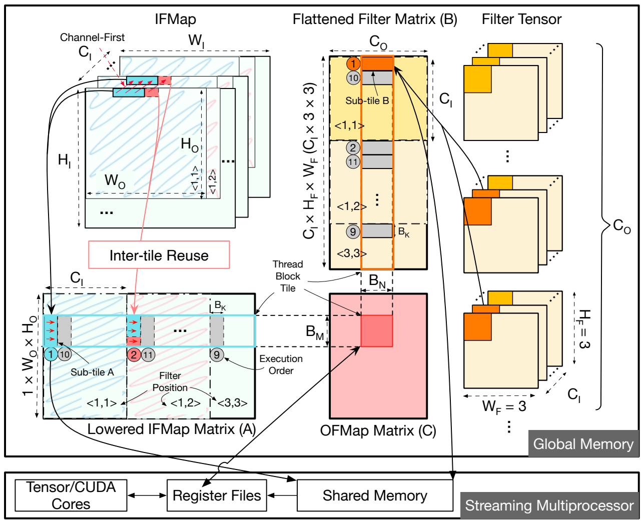


Fig. 12: Block-level implicit channel-first `im2col` on GPU TCs.

However, this optimization also leads to the IFMap duplication in the on-chip vector memory. For example, computing tile $\langle 1,1 \rangle$ and tile $\langle 1,2 \rangle$ at the same time requires storing the same channel twice inside the vector memory in Fig. 11. The maximum number of tiles allowed is a parameter for the trade-off between memory overhead and performance improvement, which we explore in the evaluation section.

C. Discussions

Our educated guess is that the design described in Sec. IV-A is similar to what is implemented in the TPU. This will be evident in our evaluation. We suspect that the reason the TPU uses separate SRAM arrays is exactly to allow each SRAM array to be individually addressed so as to feed data to the GEMM engine in a time-delayed fashion.

The reason for each SRAM array to have a large word size is to amortize the area/power overhead. It is established that a small SRAM with a narrow word is inefficient. As we will show in Sec. VII, for an SRAM array of size 256 KB, having a word size of 4 bytes increases the area overhead by 3.2 times compared to that when the word size is 32 bytes. TPU design is clever in leveraging the large word size through batching, which is common in training — a key focus of TPU-v2/v3.

V. Implementation on Tensor Cores

In this section, we describe how to implement the channel-first implicit `im2col` on GPU with Tensor Cores. The TCs [9], [22] adopt dot-product units for GEMM acceleration, which exhibit regular access pattern instead of the TPU's time-delayed access pattern. Therefore, the channel-first implicit `im2col` is naturally amenable to GPU implementation.

Block-level Channel-First im2col The challenge in applying our method to the TC is that there are usually many TCs on a GPU (eight TCs per streaming multi-processors/SM), so it is necessary to parallelize the GEMM to achieve high-performance on the GPU. To avoid the atomic update to the output matrix, GPU typically partitions the output matrix and assigns an output matrix tile to a thread block (TB). Thus, different TBs run on different SMs in parallel.

TABLE II: TPU-v2 simulator configurations.

Compute Unit	128×128 Systolic Array @ 700Mhz 256 Vector ALUs for partial sum accumulating
Regs	256 Vector Regs, 8×4 bytes for each reg
On-chip Memory	32 MB Unified On-chip Memory 128 SRAMs with 8×4 bytes Word Size
Off-chip Memory	700 GB/s High Bandwidth Memory

However, this parallelization approach is incompatible with the straightforward version of the channel-first implicit `im2col` algorithm. The reason is that the channel-first algorithm relies on filer decomposition, which needs to accumulate the OFMap $H_F \times W_F$ times and thus requires atomic updates.

We address the above challenge by simply applying our implicit `im2col` method at the blocked GEMM-level. Fig. 12 illustrates the basic idea, which is to perform the channel-first `im2col` after the equivalent GEMM is blocked. In this way, different thread blocks still update different parts of the OFMap matrix, which avoids the expensive atomic update.

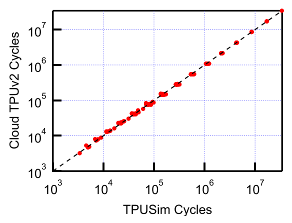
Inter-tile Reuse GPU's on-chip SRAM per SM is small compared to that of the TPU, and competition for resources between thread blocks assigned to the same SM could degrade the thread-level parallelism [12]. It is thus equally important to increase the utilization of the SRAM to avoid the blocked GEMM being bounded by the off-chip DRAM bandwidth.

We propose a reordering technique to increase data use and thus SRAM utilization. Our critical observation is that when the filter size is smaller than the stride size (which is true to virtually all CNNs today), the corresponding tiles of different decomposed filters have significant overlaps. For instance in Fig. 8c, half of the IFMap data required by decomposed filter $\langle 1,1 \rangle$ and $\langle 1,3 \rangle$ overlap (i.e., elements at 1C and 3C). When the IFMap size increases to 99×99, the working set overlap between these two decomposed filters becomes 96%.

This overlap allows us to reduce the SRAM filling latency by reordering the decomposed filters. A naive execution order of the above block-level `im2col` is to iterate over decomposed filters as they show up on the original filter, which is equivalent to iterating over the lowered IFMap tile of the same filter position and then move to the other position. For the example in Fig. 12, this would fetch the subtile of matrix A with the order of ①,⑩, ...②, ⑪, which has no data reuse. Instead, a simple reordering to ①, ②...⑩,⑪... would exploit the data reuse between subtiles. We leave it to future work to design an optimal reordering strategy.

VI. Methodology

TPU Experiment Setup For the systolic array experiment, we design and implement a configurable cycle-accurate simulator TPUSim. We configured this simulator with the same parameters as TPUv2 [28] shown in Tbl. II. For the off-


(a) GEMM. Average error: 4.42%.

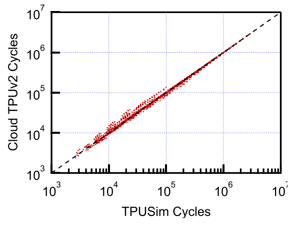
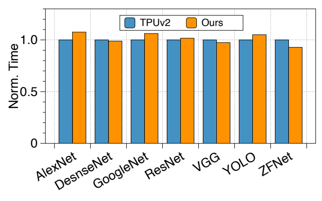

(b) Conv. Average error: 4.87%.

Fig. 13: TPUSim and TPUv2 comparison on microbenchmarks.


(a) Model Result.

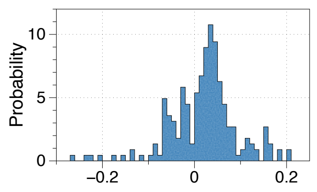
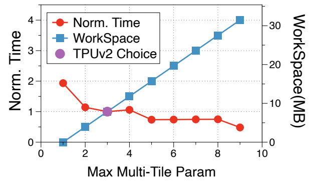

(b) Layer-wise error distribution.

Fig. 15: Performance comparison between TPUSim and TPUv2 on DNN models at batch size of 8. The MAE for all layers is 5.8%


(a) Multi-tile parameter effect.

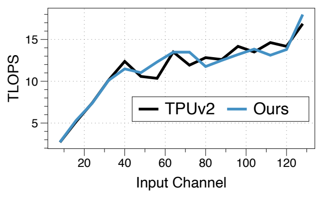
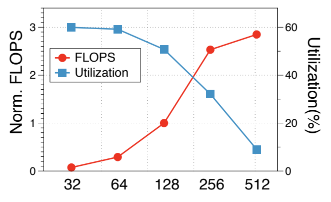

(b) Multi-tile parameter validation.

Fig. 14: The effect and validation of multi-tile parameter. (a) We use a CONV layer with $N = 8, C_I = 8, W_I = C_O = 128, W_F = 3$ and vary the number of tiles. (b) Validation of TPU's strategy (number of tiles equals $MIN(128/C_I, W_f)$) with an average error of 5.3%.


(a) Systolic array size.

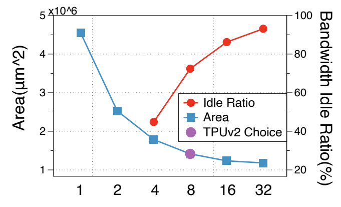

(b) Word size.

Fig. 16: Hardware design space exploration using TPUSim.

chip HBM and on-chip SRAM, we use DRAMSim3 [30] and CACTI [31] to obtain the access latencies, respectively.

GPU Experiment Setup For the GPU experiment, our evaluations are done on the NVIDIA Volta 100 GPUs using the FP16 data type. The software stack is CUDA 10.2, and the comparison baseline is cuDNN 7. Our blocked GEMM baseline is implemented based on the `cudaTensorCoreGemm` kernel from NVIDIA CUDA SDK 11.3 [32].

Workload We evaluate 7 popular neural networks, AlexNet [33], DenseNet [34], GoogleNet [35], ResNet [27], VGG [36], YOLO [37], and ZFNet [38], which cover tasks in different domains and models of different sizes. We report inference process with the widely used ImageNet dataset.

Baselines To verify the performance of our algorithm on the systolic array, we compared the experimental results of our simulator with TPUv2 with equivalent hardware parameters.

In the GPU experiments, we called the CUDNN_CONVOLUTION_FWD_ALGO_IMPLICIT_PRECOMP_GEMM interface to test the performance of different cases running on Tensor Cores as our baseline for comparison.

VII. Evaluation

A. Evaluation on TPU

TPUv2 Validation We validate the simulation accuracy of our simulator TPUSim against the TPUv2 using synthetic microbenchmarks. We first perform validation for the GEMM primitive that TPU-v2 targets. We vary the three parameters (e.g., M, N, K) of GEMM from 256 to 8192, and compare the execution cycles in the cloud TPU-v2 and TPUSim. Fig. 13a shows the validation results, where the averaged error for our simulation cycles for the GEMM primitive is 4.42%.

We compare TPUSim with our implicit `im2col` against TPUv2 using CONV layers that do not trigger our optimizations in Sec. IV-B. Fig. 13b shows the simulation cycle comparison with an averaged error of 4.87%. The close performance confirms our previous hypothesis that the TPU implements a similar implicit method.

Multi-Tile Parameter We first use a set of experiments to study the effectiveness and specific strategies of the multi-tile optimization (Sec. IV-B) for input channel size less than 128.

We use a CONV layer with $N = 8, C_I = 8, W_I = C_O = 128, W_F = 3$ and vary the number of tiles in the multi-tile computation optimization. Fig. 14a shows that the required on-chip vector memory workspace increases linearly as the maximum multi-tile param increases, but the performance improvement shows a diminishing return. When the number of tiles is 3, our simulation results match with TPUv2.

We then repeat the above experiments with different channel and filter sizes. We find that the TPU sets the number of tiles to $MIN(128/C_I, W_f)$: the multi-tile size is first bounded by the filter size and is just enough to occupy the 128×128 systolic array. Based on that strategy, Fig. 14b compares the TPUSim and TPUv2 performance in TFLOPS for varying input channel size, which has an average error of 5.3%. As such, we use this strategy in subsequent experiments.

End-to-end Model Results Besides synthetic CONV layers, we also compare the simulated and measured performance on real world CNN models that are described in Sec. VI. Fig. 15(a) shows that our design achieves performance results matched to TPUv2. Fig. 15(b) shows the error distribution of simulated and measurement latency for all layers. The MAE of all layers is 5.8%, which validates our algorithm and simulator.

Hardware Design Space With our validated simulator, we leverage its configurability to performing a design space exploration to understand the design decisions of TPU.

We first understand the impact of the systolic array size.

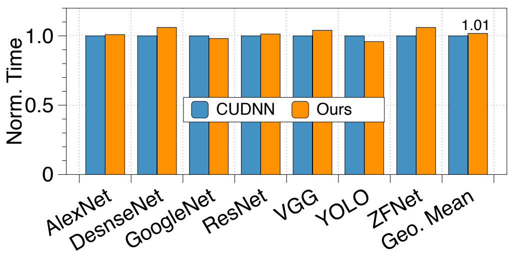
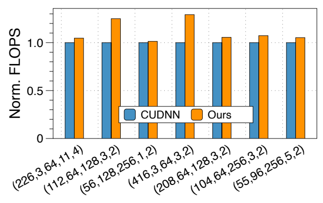
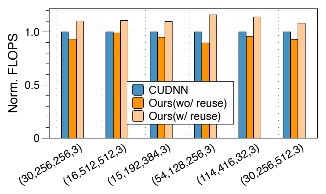


Fig. 17: Normalized execution time of cuDNN and our implicit method on V100 GPU with tensor cores at batch size of 8.



(a) Stride sensitivity.



(b) Inter-tile reuse impact.

Fig. 18: Evaluation of our GPU optimizations. The digits of x-axis label indicate W_I, C_I, C_O, W_F, with an additional digit for stride in (a).

Fig. 16a shows how the performance (FLOPS) and the array utilization changes when the array size increases from 32×32 to 512×512 when running the VGG model. As the array size increases, the performance increases while the utilization decreases. The utilization decreases by half when the array size increases from 128 to 256. This highlights the diminishing return of increasing the systolic array size, and corroborates the design decision of choosing a size of 128 for balancing peak FLOPS and utilization in TPUv2.

We also evaluate the choice of word size in the vector memory in TPU-v2. We use the OpenRAM SRAM compiler [39] with the 45 nm (freepdk45) process to estimate the area overhead when changing the word size while fixing the SRAM capacity at 256 KB. Fig. 16b shows how the SRAM bandwidth idle ratio (in VGG16 inference) and the SRAM area change as the word size increases from 1 to 32.

The word size 8 achieves the area efficiency that is close to the minimum value, while the word size 1 leads to a 5× overhead. As such, TPUv2 uses the word size 8 for the area efficiency. However, with a word size of 8, the vector memory bandwidth utilization is below 50%. This insight explains why the TPUv3 chooses to add another systolic array to leverage this extra vector memory bandwidth [18].

B. Evaluation on Tensor Cores

End-to-end Model Results Fig. 17 shows the execution time of our GPU implementation normalized to that of the baseline cuDNN implementation. Our implementation is almost identical to the baseline, average 1% slower when $N = 8$ (Fig. 17). We note that cuDNN uses low-level microarchitecture-specific optimizations [40] that are unavailable to us. The results indicate that our im2col algorithm is competitive to Nvidia's proprietary implementation.

Strided Convolution The advantage of our method over the cuDNN is when the stride is greater than 1. Fig. 18a shows the performance (FLOPS) of our design normalized to that of cuDNN for CONV layers from the benchmarked models that have a stride value greater than one (indicated by the last digit of the *x*-axis label). Our method is on average 20%, up to 40%, faster than cuDNN.

Inter-tile Reuse Fig. 18b evaluates the effectiveness of our inter-tile reuse optimization on GPUs for a set of layers in different models. These layers are used here because the overhead of global memory accesses in these layers is not completely overlapped by the computation in the pipeline, which makes it important to increase on-chip data reuse through inter-tile reuse. Overall, our optimization leads to an average 16.7% performance improvement.

VIII. Related Work

im2col Jia et al. [41] applies im2col algorithm in a deep learning system to accelerate convolution operation in DNN inference and training. Some work [42]–[44] reduce memory overhead and improve the performance by modifying the lowered IFMap into a compact format, but still explicit GEMM. Some studies [45], [46] achieve efficient direct convolution through a special data layout to optimize computation. Prior work Delta [16] provides an analytical model for existing implicit channel-last im2col on GPU, for which Duplo [12] provides a hardware-based acceleration solution. In contrast, our work identifies the more general and flexible implicit channel-first im2col that are likely used by TPUs. We describe its implementation details on the systolic array architecture and software-level implementation on GPUs.

DNN Accelerators Most of today's DNN accelerators, especially ones used in industry, target GEMM [8]–[10], [47], [48], requiring some form of im2col. Other accelerators target convolution [49]–[52] by designing a dedicated data flow for direct convolution. SCALE-Sim [53] proposes a systolic array simulator accelerating GEMM and assumes an explicit im2col execution method. Sparsity is an important property of DNN models, which prior works have exploited for acceleration [54] and robustness enhancement [55], [56]. As such, researchers have proposed various sparse accelerator designs, which are based on direct convolution [57], [58] or assume the usage of explicit im2col [13], [14], [59] and the implicit channel-last im2col algorithm [60].

Our work identifies a generic im2col algorithm that translates convolution to GEMM with practical zero-cost performance and memory overhead. We demonstrate that it can be applied to both TCs-like GEMM engines and systolic arrays. We believe that our work can encourage future study for designing sparse CNN accelerators based on the described channel-first implicit im2col algorithm.

IX. Conclusion

In this work, we propose an implicit im2col algorithm called channel-first im2col that is very likely used by TPUs. It dynamically converts a convolution into a GEMM with zero performance and memory overhead. We describe its implementation details on the TPU architecture. We also demonstrate its general applicability and performance advantages on other GEMM-engines such as the tensor cores on Nvidia's GPUs. We hope this can encourage future work on accelerating CNNs on specialized GEMM engines.

Acknowledgements We thank the anonymous reviews for their thoughtful comments and suggestions. Jingwen Leng is the corresponding author of this paper. This work was supported by the National Key R&D Program of China under Grant 2020YFB1711801, and National Natural Science Foundation of China (NSFC) grant 62072297.

References

[1] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[2] J. Levinson, J. Askeland, J. Becker, J. Dolson, D. Held, S. Kammel, J. Z. Kolter, D. Langer, O. Pink, V. Pratt *et al.*, "Towards fully autonomous driving: Systems and algorithms," in *2011 IEEE Intelligent Vehicles Symposium (IV)*. IEEE, 2011, pp. 163–168.

[3] J. Janai, F. Guney, A. Behl, A. Geiger *et al.*, "Computer vision for autonomous vehicles: Problems, datasets and state of the art," *Foundations and Trends in Computer Graphics and Vision*, 2020.

[4] E. Yurtsever, J. Lambert, A. Carballo, and K. Takeda, "A survey of autonomous driving: Common practices and emerging technologies," *IEEE Access*, vol. 8, pp. 58 443–58 469, 2020.

[5] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[6] J. Devlin *et al.*, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv*, 2018.

[7] G. G. Chowdhury, "Natural language processing," *Annual review of information science and technology*, vol. 37, no. 1, pp. 51–89, 2003.

[8] N. P. Jouppi *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *ISCA*, 2017.

[9] T. NVIDIA, "V100 gpu architecture. the world's most advanced data center gpu. version wp-08608-001_v1. 1," *NVIDIA. Aug*, p. 108, 2017.

[10] E. Medina and E. Dagan, "Habana labs purpose-built ai inference and training processor architectures: Scaling ai training systems using standard ethernet with gaudi processor," *IEEE Micro*, 2020.

[11] A. Yang, "Deep learning training at scale spring crest deep learning accelerator (intel® nervana™ nnp-t)," in *2019 IEEE Hot Chips 31 Symposium (HCS)*. IEEE, 2019, pp. 1–20.

[12] H. Kim, S. Ahn, Y. Oh, B. Kim, W. W. Ro, and W. J. Song, "Duplo: Lifting redundant memory accesses of deep neural networks for gpu tensor cores," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 725–737.

[13] A. Gondimalla, N. Chesnut, M. Thottethodi, and T. Vijaykumar, "Sparten: A sparse tensor accelerator for convolutional neural networks," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 151–165.

[14] M. Zhu, T. Zhang, Z. Gu, and Y. Xie, "Sparse tensor core: Algorithm and hardware co-design for vector-wise sparse neural networks on modern gpus," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 359–371.

[15] X. Li *et al.*, "Performance analysis of gpu-based convolutional neural networks," in *ICPP*, 2016.

[16] S. Lym, D. Lee, M. O'Connor, N. Chatterjee, and M. Erez, "Delta: Gpu performance model for deep learning applications with in-depth memory system traffic analysis," in *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2019, pp. 293–303.

[17] L.-C. Chen, G. Papandreou, I. Kokkinos, K. Murphy, and A. L. Yuille, "Deeplab: Semantic image segmentation with deep convolutional nets, atrous convolution, and fully connected crfs," *IEEE transactions on pattern analysis and machine intelligence*, 2017.

[18] N. P. Jouppi, D. H. Yoon, G. Kurian, S. Li, N. Patil, J. Laudon, C. Young, and D. Patterson, "A domain-specific supercomputer for training deep neural networks," *Communication of ACM*, 2020.

[19] K. Chellapilla, S. Puri, and P. Simard, "High performance convolutional neural networks for document processing," in *Tenth International Workshop on Frontiers in Handwriting Recognition*. Suvisoft, 2006.

[20] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cudnn: Efficient primitives for deep learning," *arXiv preprint arXiv:1410.0759*, 2014.

[21] Google, "Cloud TPU," https://cloud.google.com/tpu.

[22] M. A. Raihan, N. Goli, and T. M. Aamodt, "Modeling deep learning accelerator enabled gpus," in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2019, pp. 79–92.

[23] J. Dai, H. Qi, Y. Xiong, Y. Li, G. Zhang, H. Hu, and Y. Wei, "Deformable convolutional networks," in *2017 IEEE International Conference on Computer Vision (ICCV)*, 2017, pp. 764–773.

[24] B. Grot, J. Hestness, S. W. Keckler, and O. Mutlu, "Kilo-NOC: A heterogeneous network-on-chip architecture for scalability and service guarantees," in *2011 38th Annual International Symposium on Computer Architecture (ISCA)*, 2011, pp. 401–412.

[25] J. A. Rivers, G. S. Tyson, E. S. Davidson, and T. M. Austin, "On high-bandwidth data cache design for multi-issue processors," in *Proceedings of 30th Annual International Symposium on Microarchitecture*, 1997.

[26] C. Kong and S. Lucey, "Take it in your stride: Do we need striding in CNNs?" *arXiv preprint arXiv:1712.02502*, 2017.

[27] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[28] T. Norrie, N. Patil, D. Yoon, G. Kurian, S. Li, J. Laudon, C. Young, N. Jouppi, and D. Patterson, "The design process for google's training chips: Tpuv2 and tpuv3," *IEEE Micro*, 2021.

[29] S. Ioffe and C. Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift," in *Proceedings of the 32nd International Conference on International Conference on Machine Learning*, 2015.

[30] D. Wang, B. Ganesh, N. Tuaycharoen, K. Baynes, A. Jaleel, and B. Jacob, "Dramsim: a memory system simulator," *ACM SIGARCH Computer Architecture News*, vol. 33, no. 4, pp. 100–107, 2005.

[31] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. P. Jouppi, "Cacti 5.1," Technical Report HPL-2008-20, HP Labs, Tech. Rep., 2008.

[32] D. Guide, "Cuda c programming guide," *NVIDIA, July*, 2013.

[33] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[34] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 4700–4708.

[35] C. Szegedy *et al.*, "Going deeper with convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015.

[36] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[37] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016.

[38] M. D. Zeiler and R. Fergus, "Visualizing and understanding convolutional networks," in *European conference on computer vision*, 2014.

[39] M. R. Guthaus, J. E. Stine, S. Ataei, B. Chen, B. Wu, and M. Sarwar, "Openram: An open-source memory compiler," in *International Conference on Computer-Aided Design (ICCAD)*, 2016.

[40] L. Ma, Z. Xie, Z. Yang, J. Xue, Y. Miao, W. Cui, W. Hu, F. Yang, L. Zhang, and L. Zhou, "Rammer: Enabling holistic deep learning compiler optimizations with rtasks," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI*.

[41] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *Proceedings of the 22nd ACM international conference on Multimedia*, 2014, pp. 675–678.

[42] M. Cho and D. Brand, "Mec: memory-efficient convolution for deep neural network," in *International Conference on Machine Learning*. PMLR, 2017, pp. 815–824.

[43] A. Anderson, A. Vasudevan, C. Keane, and D. Gregg, "Low-memory gemm-based convolution algorithms for deep neural networks," 2017.

[44] ——, "Low-memory gemm-based convolution algorithms for deep neural networks," *arXiv preprint arXiv:1709.03395*, 2017.

[45] E. Georganas, S. Avancha, K. Banerjee, D. Kalamkar, G. Henry, H. Pabst, and A. Heinecke, "Anatomy of high-performance deep learning convolutions on simd architectures," in *SC18: International Conference*

for High Performance Computing, Networking, Storage and Analysis. IEEE, 2018, pp. 830–841.

[46] E. Georganas, K. Banerjee, D. Kalamkar, S. Avancha, A. Venkat, M. Anderson, G. Henry, H. Pabst, and A. Heinecke, "Harnessing deep learning via a single building block," in *International Parallel and Distributed Processing Symposium (IPDPS)*, 2020.

[47] E. Qin, A. Samajdar, H. Kwon, V. Nadella, S. Srinivasan, D. Das, B. Kaul, and T. Krishna, "Sigma: A sparse and irregular gemm accelerator with flexible interconnects for dnn training," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020, pp. 58–70.

[48] C. Guo, Y. Zhou, J. Leng, Y. Zhu, Z. Du, Q. Chen, C. Li, B. Yao, and M. Guo, "Balancing Efficiency and Flexibility for DNN Acceleration via Temporal GPU-Systolic Array Integration," in *Proceedings of the Design Automation Conference*, 2020.

[49] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 1, pp. 269–284, 2014.

[50] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam, "Shidiannao: Shifting vision processing closer to the sensor," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, 2015, pp. 92–104.

[51] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *Proceedings of the 2015 ACM/SIGDA international symposium on field-programmable gate arrays*, 2015, pp. 161–170.

[52] Y. Chen *et al.*, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *ISCA*, 2016.

[53] A. Samajdar, Y. Zhu, P. Whatmough, M. Mattina, and T. Krishna, "Scale-sim: Systolic cnn accelerator simulator," *arXiv preprint arXiv:1811.02883*, 2018.

[54] C. Guo, B. Y. Hsueh, J. Leng, Y. Qiu, Y. Guan, Z. Wang, X. Jia, X. Li, M. Guo, and Y. Zhu, "Accelerating sparse dnn models without hardware-support via tile-wise sparsity," in *International Conference for High Performance Computing, Networking, Storage and Analysis*, 2020.

[55] Y. Qiu, J. Leng, C. Guo, Q. Chen, C. Li, M. Guo, and Y. Zhu, "Adversarial Defense Through Network Profiling Based Path Extraction," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019.

[56] Y. Gan, Y. Qiu, J. Leng, M. Guo, and Y. Zhu, "Ptolemy: Architecture Support for Robust Deep Learning," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020.

[57] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, "Cnvlutin: Ineffectual-neuron-free deep neural network computing," *ACM SIGARCH Computer Architecture News*, 2016.

[58] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, "Scnn: An accelerator for compressed-sparse convolutional neural networks," *ACM SIGARCH Computer Architecture News*, vol. 45, no. 2, pp. 27–40, 2017.

[59] A. Delmas Lascorz, P. Judd, D. M. Stuart, Z. Poulos, M. Mahmoud, S. Sharify, M. Nikolic, K. Siu, and A. Moshovos, "Bit-tactical: A software/hardware approach to exploiting value and bit sparsity in neural networks," in *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019.

[60] Y. Wang, C. Zhang, Z. Xie, C. Guo, Y. Liu, and J. Leng, "Dual-side sparse tensor core," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021, pp. 1083–1095.

APPENDIX

A. Abstract

The artifact contains the source code and python scripts to set up the environment and perform the evaluation. We describe how to obtain the source code and build the project for both TPU and GPU experiments.

B. Artifact check-list (meta-information)

- **Compilation:**
  - nvcc 10.2
  - gcc 7.5.0
  - cmake 3.21.0
- **Model:** AlexNet, DenseNet, GoogleNet, ResNet, VGG, YOLO, and ZFNet
- **Data set:** ImageNet
- **Run-time environment:** Ubuntu 18.04 with Linux kernel 5.4.0
- **Hardware:**
  - CPU: Intel(R) Xeon(R) Silver 4210 CPU @ 2.20GHz
  - GPU: NVIDIA V100
  - TPU: cloud TPUv2-8
- **Output:** Console, log file and csv file.
- **Experiments:** The TPU part contains test experiments with the simulator and cloud TPU. the GPU part contains comparative experiments with our implemented code and cudnn.
- **How much disk space required (approximately)?:** 1TB for dataset. 500GB for the full evaluation.
- **How much time is needed to prepare workflow (approximately)?:** One day for datasets downloading and environment setup.
- **How much time is needed to complete experiments (approximately)?:** One day
- **Publicly available?:** Yes.
- **Code licenses (if publicly available)?:** Apache Licenses
- **Data licenses (if publicly available)?:** No
- **Workflow framework used?:** No
- **Archived (provide DOI)?:** Yes. Available at https://doi.org/10.5281/zenodo.5535284

C. Description

1) How to access: A repository that contains the code and evaluation scripts can be found in: https://doi.org/10.5281/zenodo.5535284.

2) Hardware dependencies:

- CPU: Intel(R) Xeon(R) Silver 4210 CPU @ 2.20GHz
- GPU: NVIDIA V100
- TPU: cloud TPUv2-8

3) Software dependencies:

- Ubuntu 18.04
- GPU Driver: 470.57.02
- CUDA: 10.2
- CUDNN: 7.6.5
- TPU Software Version: 1.15.0

4) Data sets: All datasets were downloaded from ImageNet(https://www.image-net.org).

D. Installation

After cloning/downloading the source code and dataset, follow these steps.

- For cloud TPU experiements, visit the cloud TPU website (https://cloud.google.com/tpu) and build the environment according to the official documentation.
- For TPU Simulator and GPU experiments, follow the instructions in README file.

E. Experiment workflow

Ensure that implementations and associated dependences are installed properly. Download the datasets using the "download.py" script. To run the benchmarks follow the instructions in README file in each subfolder. For example, running "python TPU/Sim/Sim-Code/run_sim.py" will run TPUSim tests.

F. Evaluation and expected results

Example output files for the experiments are in the result folder of the subfolder. The csv file of the expected output results can be imported into a plotting tool such as datagraph and used to draw Fig. 13 to 18.