Duplo: Lifting Redundant Memory Accesses of Deep Neural Networks for GPU Tensor Cores

Hyeonjin Kim*, Sungwoo Ahn*, Yunho Oh†, Bogil Kim*, Won Woo Ro*, William J. Song*
* *School of Electrical and Electronic Engineering*, *Yonsei University*, Seoul, Korea
† *EcoCloud*, *École Polytechnique Fédérale de Lausanne (EPFL)*, Lausanne, Vaud, Switzerland
{hyeonjin_kim, sungwoo.ahn, bogilkim, wro, wjhsong}@yonsei.ac.kr, yunho.oh@epfl.ch

***Abstract*—This paper introduces a GPU architecture named *Duplo* that minimizes redundant memory accesses of convolutions in deep neural networks (DNNs). Convolution is one of the fundamental operations used in various classes of DNNs, and it takes the majority of execution time. Various approaches have been proposed to accelerate convolutions via general matrix multiplication (GEMM), Winograd convolution, fast Fourier transform (FFT), etc. Recent introduction of tensor cores in NVIDIA GPUs particularly targets on accelerating neural network computations. A tensor core in a streaming multiprocessor (SM) is a specialized unit dedicated to handling matrix-multiply-and-accumulate (MMA) operations. The underlying operations of tensor cores represent GEMM calculations, and lowering a convolution can effectively exploit the tensor cores by transforming deeply nested convolution loops into matrix multiplication. However, lowering the convolution has a critical drawback since it requires a larger memory space (or workspace) to compute the matrix multiplication, where the expanded workspace inevitably creates multiple duplicates of the same data stored at different memory addresses. The proposed Duplo architecture tackles this challenge by leveraging compile-time information and microarchitectural supports to detect and eliminate redundant memory accesses that repeatedly load the duplicates of data in the workspace matrix. Duplo identifies data duplication based on memory addresses and convolution information generated by a compiler. It uses a load history buffer (LHB) to trace the recent load history of workspace data and their presence in register file. Every load instruction of workspace data refers to the LHB to find if potentially the same copies of data exist in the register file. If data duplicates are found, Duplo simply renames registers and makes them point to the ones containing the same values instead of issuing memory requests to load the same data. Our experiment results show that Duplo improves the performance of DNNs by 29.4% on average and saves 34.1% of energy using tensor cores.**

***Index Terms*—Deep Neural Network, GPU, Tensor Core**

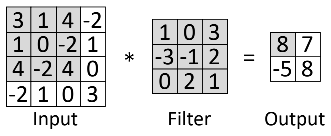
I. Introduction

Deep neural networks (DNNs) offer unparalleled accuracy in a wide range of application domains such as image classification [6], [17], [39], objection detection [33], [34], [37], image generation [16], [31], etc. A DNN is constructed by concatenating multiple layers that exploit different kinds of operations. Convolution is one of the fundamental operations used in various classes of DNNs. It extracts visual features from input data via dot-product operations between filters and receptive fields (i.e., sub-regions of input data). Notably, convolutions account for the majority of execution time of DNNs [2], [13]. In particular, recent DNNs such as generative adversarial network (GAN) [31] and YOLO [33] tend to compose neural networks using only convolutional layers. These networks therefore spend more than 99% of execution time on processing the convolution operations. A convolution is conducted by moving a filter (a.k.a. feature detector or kernel) across an input image and calculating the sum of element-wise dot products between the filter and input. The calculation is repeated over a number of filters, channels, and input images. Consequently, sliding filters on multi-dimensional input data results in deeply nested loops in every convolutional layer [40].

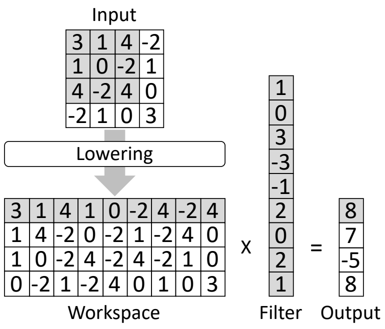
Graphics processing units (GPUs) have been serving as a primary hardware choice to accelerate neural network computing [7], [35], [36]. Instead of moving a filter across input data of a convolutional layer, unrolling the convolution loop by expanding the input data into a larger workspace matrix reduces the execution time since a GPU can exploit a greater degree of thread-level parallelism. In particular, the sum of element-wise dot products of matrices (between the filter and receptive field of input) in each iteration of convolution loop is transformed into an inner product of vectors. As opposed to repeatedly working on small vectors through the loop, unrolling the loop makes the convolution become large matrix multiplication where each loop iteration is represented as the row and column vectors of matrices. This technique is often referred to as *lowering the convolution* [1], [4]. The matrix multiplication of expanded data then can be quickly calculated via a well-known numerical library such as the general matrix multiplication (GEMM) of basic linear algebra subprograms (BLAS) [19], [30], thereby accomplishing rapid convolution.

However, the conventional GPUs still fall short of desirable throughput that neural networks demand. Tensor cores introduced in recent NVIDIA Volta and Turing GPUs [25], [26] attempt to augment the computational intensity of neural networks. A tensor core in a streaming multiprocessor (SM) is a specialized unit dedicated to processing matrix-multiply-and-accumulate (MMA) operations [23], [32]. A pair of tensor cores in the SM offer superior throughput to conventional CUDA cores by operating on matrices instead of scalar values in single-instruction, multiple-data (SIMD) fashion. The underlying operations of tensor cores in fact express GEMM calculations [23], and thus lowering convolutions can effectively exploit the tensor cores to accelerate DNNs.

Despite the effectiveness of lowering a convolution especially with tensor cores, it has a critical drawback in that the convolution becomes highly memory-inefficient. Expanding



(a) Direct convolution



(b) GEMM-based convolution

Fig. 1: (a) Direct convolution with a sliding filter. An output element is a dot product between the filter and overlapping region of input. The filter slides over the input horizontally from the left to right and produces output values in the same row, and it repeats the sliding in the subsequent rows. (b) GEMM-based convolution unrolls the convolution loop by expanding the input data into a large workspace matrix so that the convolution becomes multiplication. The matrix multiplication can be rapidly calculated using the well-known GEMM algorithm for fast convolution [19], [30].

input data during the lowering process inevitably generates multiple duplicates of the same data. As a result, lowering the convolution increases the size of matrix to compute and requires correspondingly larger memory space to accommodate the expanded volume of data. It also exacerbates the number of memory transactions to load the duplicates of data that are stored at different memory addresses in the expanded workspace matrix. For instance, hardware measurements on an NVIDIA RTX 2080 Ti GPU [26] show us that lowering convolutions provides about 14.5x performance speedup for ResNet [6] over the simplest direct convolution method (i.e., sliding filters in deeply nested loops). However, the performance speedup is traded with 8.4x increases in the number of memory transactions to load the replicates of the same data created during the lowering processes.

This observation motivates us to devise a GPU architecture named *Duplo* that minimizes redundant memory accesses of convolutions in DNNs attempting to load the duplicates of the same data. Duplo utilizes the compile-time information related to a convolution including data and filter dimensions (i.e., width, height, channels), striding distance, and batch size. When the GPU launches a convolution kernel, the set of compile-time information is loaded to the detection unit of Duplo. Data replication created during a lowering process manifests a regular pattern in an expanded workspace matrix, and it is possible to identify which matrix elements contain identical values by associating their memory addresses with the convolution information. Duplo proposes and uses

TABLE I: Configuration of Convolutional Layers in DNNs

Network	Layer	Input(NHWC)	Filter(NHWC)	Pad	Stride
ResNet [6]	C1	8x224x224x3	64x7x7x3	3	2
	C2	8x56x56x64	64x3x3x64	1	1
	C3	8x56x56x64	128x3x3x64	0	2
	C4	8x28x28x128	128x3x3x128	1	1
	C5	8x28x28x128	256x3x3x128	0	2
	C6	8x14x14x256	256x3x3x256	1	1
	C7	8x14x14x256	512x3x3x256	0	2
	C8	8x7x7x512	512x3x3x512	1	1
GAN [31]	TC1	8x4x4x512	256x5x5x512	2	2
	TC2	8x8x8x256	128x5x5x256	2	2
	TC3	8x16x16x128	64x5x5x128	2	2
	TC4	8x32x32x64	3x5x5x64	2	2
	C1	8x64x64x3	64x5x5x3	2	2
	C2	8x32x32x64	128x5x5x64	2	2
	C3	8x16x16x128	256x5x5x128	2	2
	C4	8x8x8x256	512x5x5x256	2	2
YOLO [33]	C1	8x224x224x3	32x3x3x3	1	1
	C2	8x112x112x32	64x3x3x32	1	1
	C3	8x56x56x64	128x3x3x64	1	1
	C4	8x28x28x128	256x3x3x128	1	1
	C5	8x14x14x256	512x3x3x256	1	1
	C6	8x7x7x512	1024x3x3x512	1	1

a novel detection mechanism that identifies the uniqueness of workspace data. A load history buffer (LHB) is coupled with the detection mechanism to record the load history of workspace data and their mapping to registers (i.e., which registers contain the loaded values). Every load instruction for workspace refers to the LHB to find if preceding instructions have already loaded the datum and if it is still present in the register file. If a duplicate is found, a warp register is simply renamed as the one containing the identical value instead of issuing a memory request to load a copy of the same datum again. As described, Duplo detects and eliminates redundant memory accesses of DNNs for faster and efficient convolutions.

The remainder of this paper is organized as follows. We compare various convolution methods and explain how tensor cores provide computational advantages. Then, we describe the sources of data duplication caused by lowering convolutions and introduce a method to identify data duplication. Lastly, we present the detailed implementation of Duplo and discuss experimental results.

II. Background and Related Work

This section overviews various approaches to accelerate convolution operations in DNNs and describes the detailed process of lowering a convolution that transforms deeply nested convolution loops into matrix multiplication. Tensor cores introduced in recent NVIDIA Volta and Turing GPUs [25], [26] help accelerate matrix multiplication based on the principle of GEMM [19], [30] to achieve fast convolution.

A. Comparison of Convolution Methods

There exist various methods to accelerate convolutions in DNNs [4], [9], [42]. A convolution is performed by moving a filter horizontally and vertically over an input as illustrated in

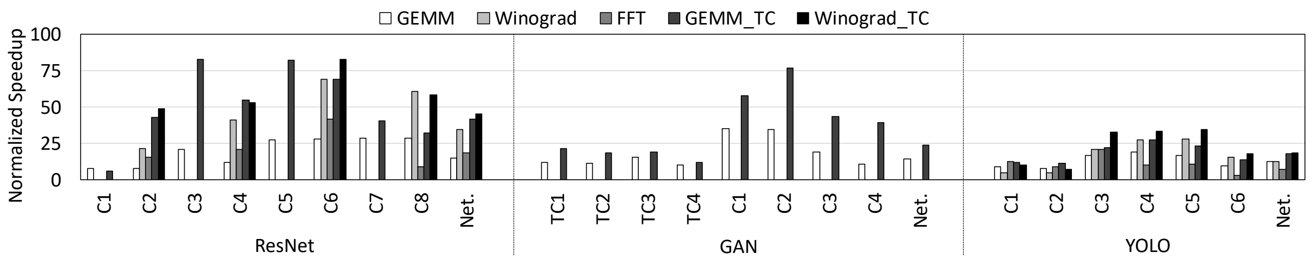


Fig. 2: Performance speedup of various convolution methods over the simplest direct convolution, measured on an NVIDIA RTX 2080 Ti GPU [26] for three representative DNNs; ResNet [6], GAN [31], and YOLO [33]. On average, the GEMM-based convolution with conventional CUDA cores provides 13.5x performance speedup, and activating the tensor cores accelerates the computations by 25.7x. The other convolution methods using the Winograd and FFT algorithms deliver 20.7x and 11.5x speedups, respectively.

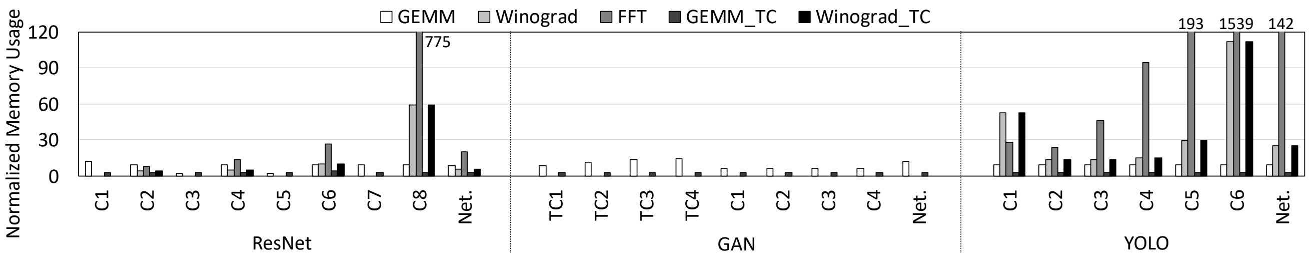


Fig. 3: Relative memory usage of various convolution methods over the direct convolution. Measurement results on an NVIDIA RTX 2080 Ti GPU [26] show that accelerating the convolutions is traded with increased memory usages. On average, the GEMM-based convolution requires 9.7x larger memory space using conventional CUDA cores, and tensor cores need 1.1x more memory space. The Winograd and FFT algorithms incur 12.2x and 53.5x greater memory usages.

Figure 1(a). In this example, a 3×3 filter is initially anchored at the upper-left corner of a 4×4 input. The highlighted output cell is calculated as the sum of element-wise dot products between the filter and overlapping region (a.k.a. receptive field) of input. The next output value in the same row is calculated as the filter slides horizontally to the right. If the filter reaches the end of row, it moves downwards and repeats the sliding from the left to right to produce the next output row. This process naively presents the mathematical expression of convolution, and we refer to it as *direct convolution*. The example in Figure 1(a) is shown with the single filter and single-channel input, but in fact the calculation is repeated numerous times over multiple channels, filters, and inputs. Consequently, the direct convolution forms deeply nested loops in a program code.

Instead of sliding filters across input data in deeply nested loops, lowering convolutions helps speeding up the computations by unrolling the loops. In the example shown in Figure 1(b), the 4×4 input is expanded to a larger 4×9 workspace matrix, where the receptive field of input in each iteration of convolution loop is flattened into a row vector of the workspace. Lowering the convolution effectively transforms the convolution loop into matrix multiplication that can be rapidly calculated via the well-known numerical library, i.e., GEMM [19], [30]. We refer to this approach as *GEMM-based convolution*. Hardware measurements on an NVIDIA RTX 2080 Ti GPU [26] show in Figure 2 that the GEMM-based convolution with conventional CUDA cores achieves 13.5x speedup on average for three representative DNNs; detailed specification of various convolutional layers forming the DNNs. Different shapes of convolutional layers are labeled as C1, C2, $\cdots$, C_n in the table. Transposed convolution of GAN is abbreviated to TC, which upsamples input data by inserting zeros before performing a convolution [31], [38], [47]. Although the table lists only three representative DNNs, many other neural networks can be easily derived by using different combinations of convolutional layers shown in the table, such as VGG [39], DiscoGAN [16], and fully convolutional network (FCN) [38].

Despite the effectiveness of GEMM algorithm for faster convolution, it has a critical drawback since the computation becomes highly memory-inefficient. As explained with the example of Figure 1(b), expanding the input inevitably generates multiple duplicates of the same data in the workspace. The amount of data replication depends on data dimensions (e.g., width, height, channels, batches) and convolution details (e.g., striding distance, filter sizes). Figure 3 shows that the GEMM-based convolution requires 9.7x larger memory space on average than the simplest direct convolution.

Other well-known convolution methods include the Winograd [18] and fast Fourier transform (FFT) [24] algorithms. In both approaches, a filter and input first need to be transformed into the Winograd or Fourier domain. Then, a convolution simply becomes an element-wise dot product between the transformed filter and input. The resulting output has to be inversely transformed back to the original domain to get the final outcome. On average, the Winograd and FFT achieve respectively. However, the performance increases are traded with 12.2x and 53.5x larger memory usages as plotted in Figure 3. The primary limitations of Winograd and FFT-based convolutions lie in their applicability. In particular, the Winograd algorithm works only with specific filter sizes (e.g., 3×3 or 5×5) [41], and both the Winograd and FFT cannot handle convolutions with non-unit-stride filters. Missing bars for the Winograd and FFT cases in Figure 2 and 3 reflect such inapplicable situations (i.e., the entire GAN and C1 layer of ResNet). Thus, accelerating the GEMM-based convolution is a more practical and useful solution with no limitations in its uses. In addition, a recent work by Cho et al. [4] demonstrates that the GEMM-based convolution can be even faster than the Winograd algorithm that is generally perceived as the fastest convolution method.

B. Acceleration with Tensor Cores

Convolution acceleration using tensor cores follows the methodology of GEMM-based convolution. A tensor core is a specialized unit dedicated to handling MMA operations in a processing block of SM [25], [26]. A couple of tensor cores in the processing block share scheduling resources (i.e., warp scheduler, register file) with other conventional combinational logic such as floating-point units and integer ALUs. Each tensor core is comprised of 16 four-element dot product (FEDP) units that collectively work on 4×4 MMA operations per cycle. A set of four consecutive threads in a warp called *threadgroup* generate a 4×8 rectangular block of data, and it takes two steps for the threadgroup to produce the data block using the tensor core. A pair of threadgroups called *octet* work together to compute an 8×8 tile of data. Collectively, four octets in a warp execute 16×16 MMA operations [32].

A pair of tensor cores are used in tandem to handle 16×16 matrix operations [23], [25], which require partitioning and fitting convolution data into 16×16 matrices. The Volta architecture has a strict restriction on the matrix size [25], but the succeeding Turing architecture has more flexibility with the supported matrix sizes [26]. The GEMM performs the matrix calculation of $D = A \times B + C$, where the A, B, C, and D matrices represent image inputs, filters, accumulators, and outputs, respectively [23]. Borrowing the example from Figure 1, Figure 4 illustrates how the A, B, and resulting D matrices are laid out for the tensor cores to compute the lowered convolution. For multi-dimensional inputs, data points belonging to different channels and batches are concatenated in the directions illustrated in the figure.

Tensor cores provide the GEMM with greater operational intensity than the conventional execution units by densely packing floating-point multipliers and adders into FEDPs. For instance, an NVIDIA Tesla V100 GPU [25] contains 16 32-bit floating-point units in a processing block. In contrast, the tensor cores in the processing block are capable of handling 256 half-precision multiplications and additions per cycle and thus offer 16x greater operational intensity (or 8x considering the data precision). Thus, the GEMM can be rapidly computed by leveraging the architectural advantages of tensor cores.

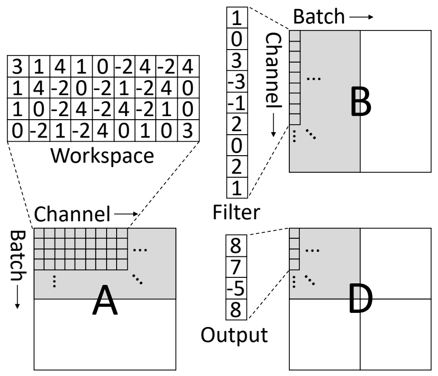


Fig. 4: Accelerating a convolution using tensor cores follows the methodology of GEMM-based convolution, where multi-dimensional image data (i.e., width, height, channels, batches) are expanded to a large workspace matrix. A pair of tensor cores work on a 16×16 tile of GEMM (i.e., $D = A \times B + C$), and each of four 8-thread octets in a warp produces a quarter of output matrix. A pair of threadgroups in an octet work in tandem using FEDPs to generate an 8×8 tile [32].

The 32 threads of a warp are separated into 8-thread octets, and each octet takes a half of A and B matrices as inputs (i.e., 8×16 or 16×8 matrices). A couple of threadgroups in an octet work on a series of 4×4 data using FEDPs to produce an 8×8 tile, where each threadgroup generates a 4×8 data block. Collectively, every octet produces a quarter of 16×16 matrix as shown in Figure 4. To parallelize the executions of octets in a warp, each half of input matrices A and B are loaded twice by different octets [32]. For instance, the shaded upper-half of matrix A in Figure 4 is used twice, once to generate the upper-left tile of matrix D and again to generate the upper-right tile of matrix D. Tensor cores have dedicated memory instructions for efficient transfers of matrix data between register file and device memory. In particular, a tensor-core-load instruction fetches 16 half-precision data (e.g., a row of matrix A or a column of B) to eight 32-bit registers for a tensor core to compute 4×4 MMA operations. Since the executions of octets are independent from each other, tensor cores generate two separate tensor-core-load requests to fetch the same data that are then placed at disjoint registers. Such an approach doubles up the number of load requests and register file occupancy to hold the dual copies of the same data, which even exacerbates the data duplication problem of GEMM-based convolution.

C. GEMM Optimization in GPGPU

GEMM-based convolution requires generating a large workspace matrix to lower the convolution as described with Figure 1(b). Lowering the convolution can be achieved in two different ways, i) explicitly creating the workspace in the global memory or ii) lazily lowering the convolution via a shared memory. The cuDNN library using tensor cores leverages the latter approach referred to as implicit GEMM [3]. It creates a portion of workspace by repeatedly loading input data and expanding them into the shared memory. Since the unexpanded input data are loaded from the same global memory addresses, the implicit GEMM can exploit data locality in caches and saves the global memory space [22]. Once the workspace is formed in the shared memory, tensor cores can quickly load MMA operands from the shared memory. As a result, the implicit GEMM uses 8.8x less global memory space than the explicit method as shown in Figure 3.

Despite the benefits of implicit GEMM, its performance is bounded by the size of shared memory since it limits how much portion of workspace can be formed at a time and thus the number of cooperative thread arrays (CTAs) that can be concurrently scheduled. In particular, a shared memory in NVIDIA Volta architecture [25] can be configured up to 96KB. With a half-precision mode, a CTA of implicit GEMM consumes total 64KB of shared memory space (i.e., 16KB each for half-precision matrices A and B, and 32KB for a single-precision matrix C) to perform $A \times B + C$. It implies that the implicit GEMM can schedule only one CTA within the shared memory, and it does not provide enough thread-level parallelism (TLP). As revealed in the work of Yan et al. [45], GEMM operations using tensor cores are memory-bounded, and thus provisioning a sufficient degree of TLP is essential to the performance.

We used `cudaTensorCoreGemm` from NVIDIA CUDA SDK 9.1 [27] to implement GEMM kernels. We compared three different cases of storing GEMM operands in the shared memory, i) loading all A, B, and C (64KB per CTA), ii) storing only A and C (48KB per CTA), and iii) placing only C in the shared memory (32KB per CTA). Within the 96KB shared memory, the first case (i.e., loading all matrices to the shared memory) can allocate only one CTA and shows the worst performance because the limited TLP cannot effectively hide memory latencies. The last case (i.e., having only C in the shared memory) shows the highest performance by allocating up to three CTAs, achieving 29.7% better performance than the first scenario. Thus, we use the last case as our baseline GEMM implementation throughout the paper.

III. Motivation and Methodology

Convolutions account for the majority of execution time of DNNs, and thus accelerating the convolutions is a key to speed up the DNNs [2], [13]. Lowering the convolutions helps accelerate the DNNs by unrolling deeply nested convolution loops into large matrix multiplication as shown in Figure 1. However, a lowering process inevitably generates multiple duplicates of the same data while expanding input data into a large workspace. Thus, it becomes wasteful for the memory and register file of a GPU to store the enlarged workspace where a good portion of data are duplicate copies. Such an observation motivates us to devise an architectural solution to detect and eliminate redundant memory accesses loading the same data multiple times. This section describes our methodology to identify data duplication in the process of GEMM-based convolution.

A. Sources of Data Duplication

Horizontal and vertical striding of a filter over input data causes data duplication in a workspace matrix, as depicted in

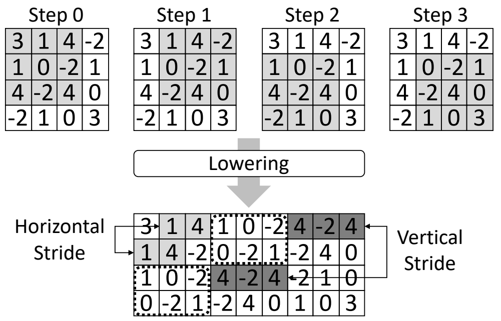


Fig. 5: Lowering a convolution generates multiple duplicates of input data in a workspace. Horizontal and vertical striding of a filter causes overlapping regions among movements, and the input data in the overlapping regions are the ones being replicated in the workspace.

Figure 5. The example shows that a 3×3 filter slides over a 4×4 input with a unit stride, similar to the example illustrated in Figure 1. The horizontal stride of filter from step 0 to step 1 makes two middle elements of the first row (i.e., [1, 4]) appear twice in the workspace matrix in the diagonal direction from the upper right to lower left as highlighted in the figure. The same phenomenon occurs in the subsequent rows (i.e., [0, -2], [-2, 4]) and other horizontal movements (i.e., from step 2 to 3). Similarly, the vertical movement of filter from step 0 to step 2 causes two middle rows appear twice in the workspace (i.e., [1, 0, -2] and [4, -2, 4]). The horizontal and vertical striding of filter creates patterns of data duplication. For example, two dotted boxes in Figure 5 are exactly identical (i.e., [1, 0, -2], [0, -2, 1]), and such a duplicate tile is referred to as a *patch*. We define data duplication induced by horizontal striding of a filter as *intra-patch duplication* since data replicates appear within a patch (i.e., [1, 4] in the first and second rows), and we refer to data duplication caused by vertical striding as *inter-patch duplication* (i.e., [4, -2, 4] in the first and third rows). The amount of data duplication and patterns are determined by convolution parameters including data dimensions (i.e., width, height, channels), filter size, striding distance, and batch size (i.e., number of images).

B. Identification of Duplication Patterns

With given convolution parameters, it is possible to identify which entries of a workspace matrix have the identical copies of data. Figure 6 illustrates the proposed method to identify data duplication occurring in the workspace. Workspace entries are shown with sequential array indices starting from 0, assuming that the data are organized in an one-dimensional array. Since the vertical striding of a filter causes inter-patch duplication, we label individual patches to identify them using patch IDs. A *patch ID* is given as `patch_id = patch_row_idx + patch_col_idx`, where the `patch_row_idx` and `patch_col_idx` are the row and column positions of a patch. The row index of patch is given as `patch_row_idx = worksp_row_idx / output_height`, and the column index is `patch_col_idx = worksp_col_idx / filter_width`. The `worksp_row_idx` and `worksp_col_idx`

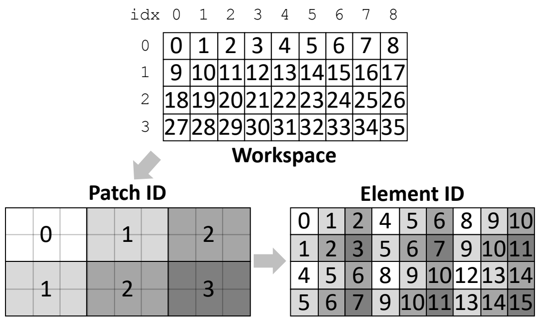


Fig. 6: The proposed method uses *patch ID* and *element ID* to identify data duplication created by the vertical and horizontal striding of a filter. Labeling patches and workspace elements in such a way enables Duplo to distinguish them since duplicates patches or elements are assigned the identical IDs.

are the row and column indices of a workspace entry derived from `worksp_row_idx = array_idx / (filter_width * filter_height)` and `worksp_col_idx = array_idx % (filter_width * filter_height)`, where `array_idx` is an array index of workspace sequentially assigned from 0 to 35 as shown in Figure 6. Labeling the patches in such a way makes them uniquely identifiable, and importantly duplicate patches are assigned the identical IDs. Replicates of a patch appear diagonally from the upper right to lower left as shown in the figure, and therefore patches in the same diagonal contain the identical set of values.

Horizontal striding of a filter results in intra-patch duplication. Detecting intra-patch duplication needs further identification of entries within a patch. For this purpose, we use the patch ID as an *offset* such that `offset = patch_id * input_width`. A workspace entry is assigned an *element ID* using the offset as `element_id = worksp_row_idx % output_width + worksp_col_idx % filter_width + offset`. In the example shown in Figure 6, there are total 16 unique element IDs from 0 to 15, and the count matches the number of elements in the original of 4×4 input. The examples so far have been shown with a single-channel image with a unit-stride filter, but in fact convolutions in DNNs work on a batch of multi-dimensional inputs possibly with non-unit-stride filters.

C. Multi-Batch/Channel, Non-Unit Stride

Generalizing the data identification method with multi-image batches, multi-channel inputs, and non-unit strides of filters requires introducing a few more rules. A multi-dimensional convolution input is comprised of four dimensions, N, H, W, and C as listed in Table I. N denotes the number of input images in a batch, and H, W, and C are the height, width, and number of channels of an image, respectively. Organizing convolution data in the order of *NCHW* is known to be better exploited by conventional CUDA cores [20], but cuDNN library mandates organizing them in the *NHWC* order for tensor cores to work [28].

A multi-dimensional input is expanded to a workspace as shown in Figure 4. For instance, the matrix A in the figure shows that channel data are appended horizontally to the right, and the batch images are concatenated downwards. The fundamentals of data duplication (i.e., intra and inter-patch duplication) still hold with the multi-dimensional input, but extra rules are necessary to differentiate data across the boundaries of different batch images and channels. In particular, there exists no data duplication between disjoint images in a batch. When batch images are simply put together, workspace elements located in the same diagonal do not necessarily have identical values if the elements belong to different images. To differentiate workspace elements across the boundaries of batch images, it is necessary to pair an element ID with a *batch ID* that can be calculated as `batch_id = worksp_row_idx / (output_width * output_height)`.

The use of non-unit-stride filters affects both intra and inter-patch duplication. A non-unit stride (i.e., striding distance greater than 1) creates discontinuous patterns of patch replication and thus needs to modify the calculation of patch ID as `patch_ID = patch_row_idx * stride_dist + patch_col_idx`, where `stride_dist` is a striding distance.

Similar to the case of multi-image batch, different channels of an input do not cause data duplication. It is also necessary to differentiate data across the boundaries of different channels, and it amends the rules to calculate a patch offset and element ID as follows. With a multi-channel input, the patch offset becomes `offset = patch_id * input_width * num_channels`, where `num_channels` is the number of channels in the input data. Using the offset, an element ID becomes `element_id = worksp_row_idx % output_width * num_channels * stride_dist + worksp_col_idx % (filter_width * num_channels) + offset`.

As described, the proposed method generates and uses a pair of `batch_id` and `element_id` to identify the uniqueness of workspace data. The IDs are calculated based on convolution parameters such as input data dimensions, striding distance, filter dimensions, and batch size. Importantly, the method assigns the identical IDs to the workspace elements containing the duplicates of the same data, and the total number of unique IDs is equivalent to the size of original input data. The identification mechanism is calculable as long as the workspace and convolution information is provided. It may be plausible to consider compiler-only solutions without hardware supports to achieve the goal. However, the compiler solutions have several critical limitations, and we will discuss them later in Section IV-D.

IV. Duplo Architecture

We propose a GPU architecture named *Duplo* that resolves the data duplication problem of GEMM-based convolution using tensor cores. Duplo leverages the identification mechanism described in Section III to detect and remove tensor-core-load instructions that attempt to fetch duplicate data that were already loaded and placed in register file. It utilizes a compiler support to generate the convolution information including input data dimensions (i.e., width, height, channels), striding distance, filter dimensions, batch size, and starting address of workspace.

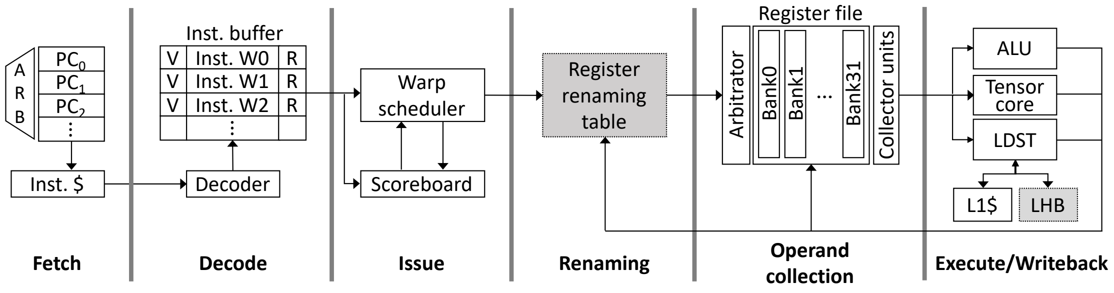


Fig. 7: Duplo architecture adopts and uses the warp register renaming scheme from the work of Kim et al. [15]. If a tensor-core-load instruction finds the presence of duplicate data by referring to the LHB, Duplo replaces its destination register with the one indicated by the LHB entry and updates the register renaming table accordingly. The use of register renaming scheme enables Duplo to bypass the redundant memory request upon the detection of data duplication.

The set of convolution information is initially stored in global memory and loaded to the detection unit of Duplo when a convolution kernel launches. The parameters are used for programming the detection unit that checks if the GPU pipeline issues redundant memory requests of duplicate data. Figure 7 overviews the pipeline stages of Duplo architecture.

The detection unit is composed of an ID generator and load history buffer (LHB) as drawn in Figure 8. The ID generator calculates a pair of element and batch IDs based on the memory address of a tensor-core-load instruction. The element ID is used for indexing the LHB whose entries include the recent history of tensor-core-load instructions and locations of loaded values in the register file. The LHB informs if any preceding instructions have already fetched duplicate data and if the values are still present in the register file. If a matching entry is found in the LHB, Duplo skips processing the load request and renames a warp register to point to the one holding the identical datum. If no matching entries exist in the LHB, the load request is sent out to the L1 cache and processed in the conventional manner. Consequently, eliminating redundant memory accesses that repeatedly load duplicates of data avoids traversing the memory hierarchy and thus leads to performance improvement and energy saving. The remainder of this section presents the details of Duplo architecture and its operations.

A. Compiler Support and ID Generation

Duplo leverages a compiler support to generate a set of convolution information encompassing dimensions of input data and filters, striding distance, batch size, and starting address of workspace. The information is initially stored in the global memory of GPU and recalled at the launch of a convolution kernel. The size of convolution information generated by the compiler totals only 32 bytes per kernel.

The detection unit is initially power-gated to conserve unnecessary power dissipation, and it is woken up upon a kernel launch. At the initiation of kernel execution, the convolution information generated by the compiler is loaded to the detection unit to program the ID generator. The ID generator leverages this information to calculate a pair of batch and element IDs, using the identification method described in Section III. Since data duplication appears only in a workspace, the ID generator checks if the memory address of a tensor-core-load instruction received from the load-store (LDST) unit falls into the workspace region. If so, the memory address is then translated to an array index as shown in Figure 6 to calculate the batch and element IDs.

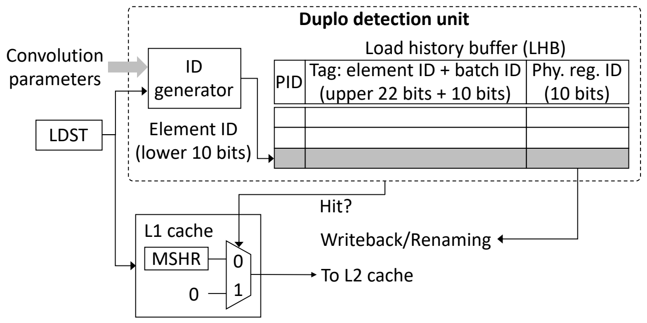


Fig. 8: A detection unit is comprised of an ID generator and load history buffer (LHB). The ID generator calculates a pair of batch and element IDs based on the memory address of a tensor-core-load instruction. The generated IDs are used as tags and also for indexing the LHB whose entries record the history of preceding tensor-core-load instructions and locations of loaded values in register file.

To simplify the logic design of ID generator, each convolution parameter is mandated to be a power of two, which is generally the case. Duplo requires data dimensions to conform to the size requirement, similar to tensor cores that support only certain predefined data dimensions. Such a restriction significantly simplifies the ID generator since all the multiply, divide, and modulo operations involved in the ID calculations simply become bit-shifting and masking operations. However, filters do not have to comply with the power-of-two size restriction. In nearly all cases, they have small odd-numbered sizes (e.g., 3×3). There are known solutions to simplify divide and modulo operations using simple logical operators for small divisors [10] such as 3, 5, etc. It is assumed that accessing the ID generator and LHB takes two clock cycles. With a more conservative assumption of three-cycle latency, our experiments with the DNNs in Table I experienced only about 0.9% performance degradation.

TABLE II: An Example of Duplo Workflow Using Load History Buffer (LHB) to Detect Data Duplication

Inst. #	Inst (op, dst, src, stride)	`array_idx`	`element_ID`	LHB entry #	LHB status	Reg. renaming	LHB operations
1	`wmma.load.a, %r4, [%r23], %r27`	2	2	2	Miss	`%r4` $\rightarrow$ **`%p2`**	Entry allocation
2	`wmma.load.b, %r2, [%r21], %r30`	-	-	-	-	`%r2` $\rightarrow$ `%p1`	N/A
3	`wmma.load.a, %r3, [%r14], %r27`	10	2	2	Hit	`%r3` $\rightarrow$ **`%p2`**	Register reuse
4	`wmma.load.a, %r8, [%r16], %r27`	28	6	2	Miss	`%r8` $\rightarrow$ `%p6`	Entry replacement

B. LHB and Register Renaming

A load history buffer is appended to the LDST unit of GPU pipeline to record the history of tensor-core-load instructions and locations of loaded values in the register file as shown in Figure 8. After generating a pair of batch and element IDs, the least-significant 10 bits of element ID are hashed for indexing the 1024-entry, direct-mapped LHB. The upper 22 bits of element ID are combined with the 10-bit batch ID and a process ID (PID), and they are used as the tag of an LHB entry. The total 32 bits of element ID can cover up to 4GB workspace in a GPU, and the 10-bit batch ID can include up to 1,024 images.

A tensor-core-load instruction consults with the LHB to find if any preceding instructions have already loaded duplicate data. Since the tag field of an LHB entry uniquely identifies data using batch and element IDs, a matching tag indicates that the wanted data were already loaded. The LHB entry contains a 10-bit register ID to record which registers hold the duplicates of data. When the tag matches (i.e., LHB hit), Duplo skips the tensor-core-load instruction and renames its destination register as the one pointed by the register ID field of LHB entry. Duplo adopts and uses the warp register renaming scheme from the work of Kim et al. [15]. When an instruction is issued, it reads the renaming table to find the proper translation of source operands and records the new mapping of destination register. If a tensor-core-load instruction finds the presence of duplicate data by referring to the LHB, it replaces the destination register with the one indicated by the LHB entry and updates the register renaming table accordingly. Duplo is distinguished from previous instruction elimination techniques [5], [14], [15], [43], [44], [46], [48], [49] in that the prior methods can eliminate instructions only when successive load instructions have the same memory addresses, whereas Duplo identifies and removes tensor-core-load instructions that fetch duplicate data at different memory addresses using a much simpler microarchitectural scheme.

Tensor cores perform warp-granular operations divided into four octets as described in Section II and Figure 4. Since threads of a warp collectively work on a single 16×16 tile, presence of a tile element implies that the entire tile data are in the register file. Thus, Duplo renames registers at the warp granularity instead of performing element-wise register renaming. The use of register renaming scheme enables Duplo to bypass a redundant memory request upon the detection of data duplication, and the bypassed load instruction continues its progress along the pipeline as if the memory request is immediately served. If the LHB misses, the tensor-core-load request is sent out to the L1 cache in the conventional way. To hide the latency of LHB lookup, Duplo accesses the LHB and L1 cache in parallel. In case of LHB hit, the L1 cache is signaled to cancel the load request so that no further progresses are made down the memory hierarchy.

A hit in the LHB must guarantee that a copy of duplicate data exists in the register file. The LHB releases an entry when the corresponding tensor-core-load instruction retires, since its destination register can be overwritten by subsequent instructions after the retirement and thus no longer guarantees the right value. However, continuous hits at the LHB entry can relay the warp register to the next tensor-core-load instructions until the very last one commits, thereby extending the effective lifetime and reusability of warp register. Upon an encounter of a store instruction with matched tags, a LHB entry is released to avoid consistency problems. However, such a case was never observed in our experiments.

C. Duplo Workflow

Table II demonstrates an example of Duplo workflow with tensor-core-load instructions, based on the array indices and element IDs used in the example of Figure 6. It is assumed that the batch size is 1, so this example does not rely on `batch_id` for the ID calculation. Each `wmma.load` instruction fetches a block of data to eight 32-bit registers per thread. Tensor cores in the Volta architecture use half-precision floating-point data for *A* and *B* matrices, and thus each load instruction brings total 16 half-precision data at a time to fill eight registers [25]. For simplicity, `wmma.load` in Table II is shown with a single destination register.

When a tensor-core-load instruction is issued, the LHB checks the validity of memory address whether it belongs to the workspace region. The first instruction in the table attempts to load workspace data, and a generated element ID is 2. The instruction is mapped to the entry #2 of LHB and results in a compulsory miss. The load request is sent out to the L1 cache for the LHB miss, and the LHB allocates a new entry for this instruction. The destination register `%r4` is renamed as `%p2`, and the register renaming table is correspondingly updated for subsequent instructions to refer to the renamed register. The next tensor-core-load instruction bypasses the LHB and directly heads to the L1 cache since its memory address does not fall into the workspace region. The third instruction has a different memory address from the first one, but it is assigned the same element ID implying that its load value should be identical to the one already loaded by the first instruction. Duplo simply renames the `%r3` register as `%p2` and skips the memory transaction. The memory address of last instruction has the element ID of 6, and it is again mapped to the LHB

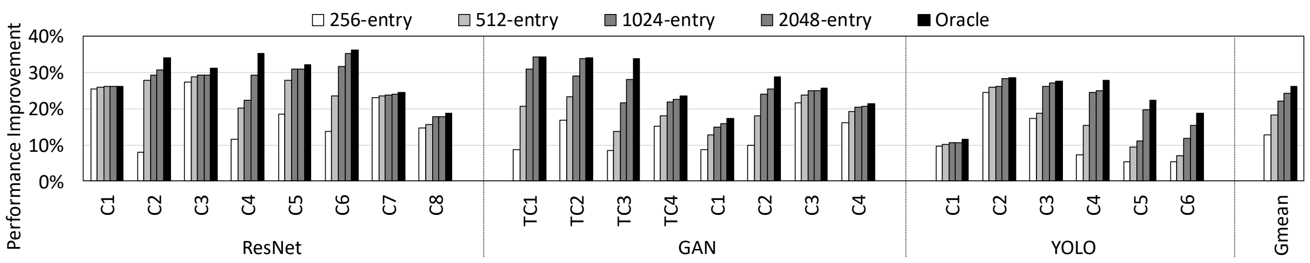


Fig. 9: Performance improvements of Duplo with variable-sized LHBs for ResNet [6], GAN [31], and YOLO [33]. The performance improvements are dependent on the LHB size, and an infinite-sized LHB (i.e., oracle) delivers 25.9% performance speed up on average.

entry #2. However, the occupied entry has a different tag and thus results in a conflict miss. The LHB replaces the entry and stores the most recent tensor-core-load information.

D. Limitations of Compiler-Only Solutions

The proposed identification mechanism to detect data duplication utilizes the convolution information and memory addresses of workspace data that are available during compile time. It is plausible to consider compiler-only solutions without microarchitectural supports, but they have several critical limitations as follows.

A possible compiler solution is to identify data duplication offline and move duplicate data between registers, i.e., `mov` instead of `load`. For instance, assume two warps, *A* and *B*. The warp *A* precedes *B*, and the warp *B* wants to reuse the register value of warp *A* to avoid an unnecessary memory transaction. However, a compiler cannot handle warp-to-warp register moves without hardware supports because the warp information is not available during the compile time [8], [29].

Another plausible implementation is to append ID tags to every tensor-core-load instruction, using the identification mechanism of Duplo but calculated offline by a compiler. A primary drawback of this approach is the gross size of tags. For example, the C2 layer of YOLO involves approximately 6.8 million tensor-core-load instructions for its workspace. It implies that this approach will need 27.2GB of storage just for the tags. In addition, it is not possible for the compiler to know which warps hold wanted data. It requires a hardware support such as a load history buffer to trace the liveliness of warp registers and rename them for duplicate data.

V. Experiment Results

A. Experiment Environment

We used the GPGPU-sim with a tensor core model [32] for evaluations, and it is configured as NVIDIA Titan V-like GPU [12] as shown in Table III. We use this setup as a baseline GPU model in our experiments. Evaluations are made with three representative DNNs from different application domains; ResNet [6], GAN [31], and YOLO [33]. They are implemented based on a `cudaTensorCoreGemm` kernel from NVIDIA CUDA SDK 9.1 [27]. Table I lists the detailed specification of various convolutional layers constructing the DNNs. Area and energy implications of Duplo are assessed using the McPAT [21].

TABLE III: Configuration of Baseline GPU Model [25]

Configuration	Parameter
# of SMs	80
Clock frequency	1,200MHz
Max # of CTAs/SM	32
Max # of warps/SM	64
Warp schedulers/SM	4
Warp scheduling policy	Greedy-then-oldest (GTO)
Tensor cores/SM	8
Register file/SM	256KB
Unified L1 cache/SM	128KB
L2 cache	4.5MB, 32 sets, 24 ways, 120 cycles [11]
DRAM bandwidth	652.8GB/s

B. LHB Size and Performance

Duplo employs the identification mechanism explained in Section III and a load history buffer to detect redundant tensor-core-load instructions that attempt to fetch duplicate copies of data. By eliminating the redundant memory accesses via warp register renaming, Duplo achieves performance gains as shown in Figure 9. The graph shows the performance improvements of Duplo with variable-sized LHBs over the baseline, i.e., GEMM-based convolutions using tensor cores. An infinite-sized LHB (shown as "oracle" in the graph) delivers about 25.9% performance increase on average for the convolutional layers of three representative DNNs.

Although GPUs are known to hide memory latencies well, the result implies that the memory system has non-trivial contributions to the overall GPU performance. The work of Kim et al. [15] reported that removing load instructions via register renaming had little impact on the performance in their warp instruction reuse (WIR) architecture, but Duplo can turn it into performance improvement via aggressive elimination of redundant memory accesses. In particular, the oracle case in Figure 9 shows that it eliminates about 76% of tensor-core-load instructions on average and decreases the number of LDST stall cycles by 24.0%, which the WIR scheme of Kim et al. [15] cannot achieve since their implementation can remove successive load instructions only when they happen to have the same memory addresses.

A 2048-entry, direct-mapped LHB shows near-oracle performance with only 1.8% difference to it. Apparently, the LHB size makes a tradeoff between performance and overhead, and we claim that it is a design option. Considering the tradeoff, we

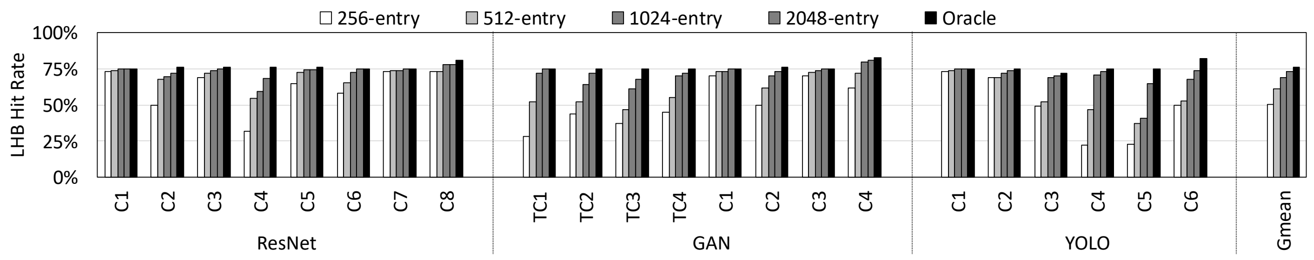


Fig. 10: Hit rate of direct-mapped LHB with its size varied between 256 and 2,048 entries. The LHB hit rate is governed by buffer size and convolution parameters. The LHB favors large buffer size and shorter striding distance of convolution filters. Increasing the number of channels or batch size has a negative impact on the LHB hit rate.

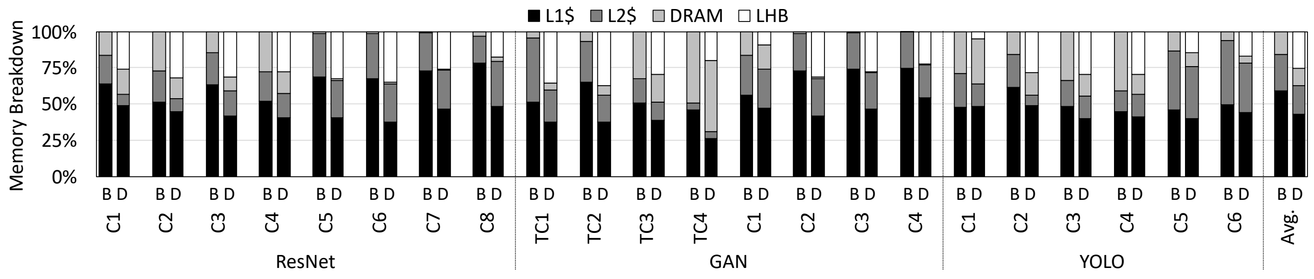


Fig. 11: Breakdown of data services along the memory hierarchy of baseline (**B**) and Duplo (**D**) with an 1024-entry, direct-mapped LHB. The graph shows which component in the memory hierarchy supplies data. The results show that the LHB saves a good portion of the DRAM traffic and turns the savings into performance increase and energy reduction.

chose an 1024-entry, direct-mapped LHB as a default option for case studies in the remainder of this section. The 1024-entry LHB shows on average 22.1% performance speedup that is about 4/5 of the oracle scenario.

C. LHB Hit Rate

Performance enhancements of Duplo originate from bypassing redundant memory accesses. The hit rate of LHB is crucial for the performance since a higher hit rate indicates that the LHB filters more redundant memory accesses. Figure 10 plots the hit rate of LHB with its size varied between 256 and 2,048 entries. The LHB hit rate is governed by two primary factors, i) buffer size and ii) convolution parameters. A large buffer helps magnify the viewing scope of LHB, and it leads to a higher hit rate. Convolution parameters determine the amount of duplicate data and size of workspace that the LHB should handle. In particular, the LHB favors a short striding distance since it increases the number of duplicate data and thus filtering chances in the LHB. On the other hand, increasing the number of channels or batch size negatively affects the LHB hit rate because it enlarges the size of workspace without creating data duplication between batch images or channels.

Empirical data for the DNNs in Table I tell us that the theoretical upper limit of LHB hit rate is 88.9%. However, Figure 10 shows that the LHB hit rates saturate around 76% even with the oracle scenarios. The cause is due to evicting LHB entries at the retirement of tensor-core-load instructions. After a tensor-core-load instruction retires, there is no guarantee that its loaded values are still valid in register file since the warp register can be overwritten by subsequent instructions. To avoid such mishaps, the LHB proactively evicts the entry when the tensor-core-load instruction retires unless the warp register is relayed by successive tensor-core-load instructions for repeated LHB hits. It may be possible to preserve the warp register even after the retirement of tensor-core-load instruction, but such an operation requires a tight coupling between the register renaming table and LHB and thus is difficult to manage.

D. Impacts on Cache and Memory Behaviors

An LHB hit indicates that the load value of a tensor-core-load instruction has already been fetched and present in register file. Duplo then skips the redundant memory request that attempts to fetch duplicate datum by renaming its destination register as the one holding the identical value. Such a scheme saves a significant amount of memory transactions sent out to the memory hierarchy of GPU. Figure 11 reveals which component in the memory hierarchy serves data upon memory requests. The results show that Duplo with a 1024-entry, direct-mapped LHB reduces the DRAM traffic by 26.6% on average, which highlights the strength of Duplo that turns off-chip DRAM transactions into in-SM register renaming. In case of implicit GEMM (refer to Section II-C), Duplo can still achieve performance improvements by transforming shared memory accesses into simpler register renaming.

In the meantime, the amount of data services by the L1 and L2 caches decrease by 28.1% and 19.2%, respectively. If a bypassed memory request in Duplo were to hit in the caches of baseline GPU, eliminating the redundant memory access still contributes to performance improvements by replacing 28-cycle L1 or 120-cycle L2 latencies [11] (see Table III)

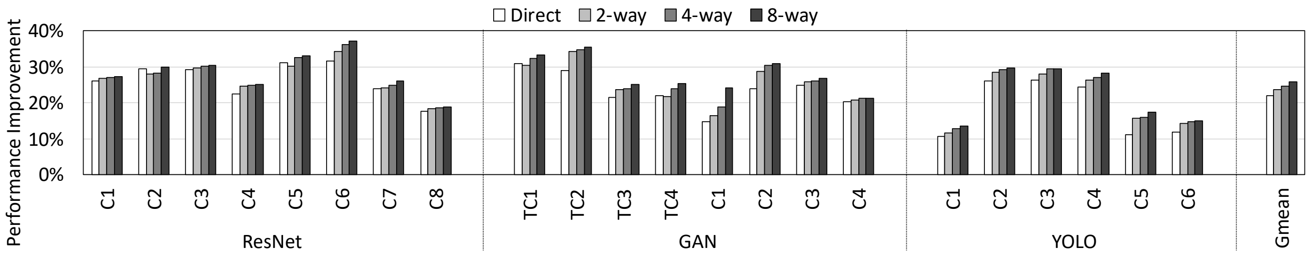


Fig. 12: Performance impacts of set-associative LHBs. The results show that the 8-way LHB provides only 3.6% better performance than the direct-mapped LHB. Since a series of tensor-core-load instructions tend to access sequentially aligned data, neighboring tensor-core-load instructions are well dispersed across disjoint LHB entries.

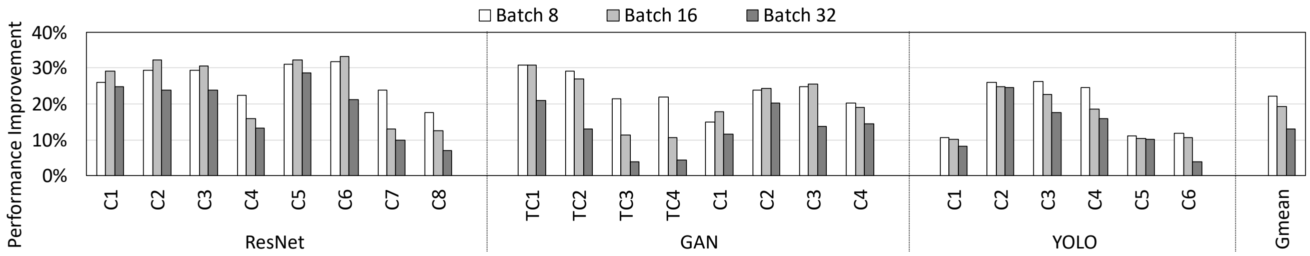


Fig. 13: Performance implications of variable-sized batches. Using a large batch proportionally increases the size of workspace. The results show that increasing the batch size from 8 to 32 images decreases the overall performance by 8.2% since there is no data duplication created across different batch images. This can be mitigated by employing a larger LHB.

with two-cycle delay of detection unit and LHB in Duplo (see Figure 7 and 8). In the baseline GPU, 32KB L1 and 4.5MB L2 caches experience 60.2% and 42.9% miss rates on average for the DNNs listed in Table I. Even if the caches are increased to 512KB L1 (i.e., 16x larger than the baseline) and 18MB L2 (i.e., 4x greater), they produce only 1.8% performance speedup. It implies that simply increasing the cache sizes is not a proper solution to accelerate the DNNs.

E. Set-Associative LHB

Figure 12 shows performance changes when the 1024-entry LHB is configured as set-associative buffers while the total buffer size is kept unchanged. The experiment did not consider additional timing delays incurred by set-associative designs, and thus it overestimates the performance of set-associative LHBs. The graph shows that the 8-way LHB produces a marginal performance improvement of 3.6% over the direct-mapped buffer. Since a series of tensor-core-load instructions tend to access sequentially aligned data, neighboring tensor-core-load instructions are well distributed to disjoint LHB entries without causing many conflict misses. Therefore, a simple direct-mapped buffer is sufficient to serve the purpose, and the experiment results in Figure 12 prove that set-associative buffers are not necessary.

F. Large Batch Inputs

There exists no data duplication across the boundaries of different batch images, as discussed in Section III. Using a large batch in neural networks proportionally increases the size of workspace. It requires a correspondingly larger LHB to cover the large-batch workspace. Figure 13 shows performance changes when engaging different size of batches with an 1024-entry, direct-mapped LHB. Executing a large batch of input data degrades the overall performance of Duplo; all of our earlier experiments (i.e., Figure 9 to 12) were done with the batch size of 8 (see Table I). The resulting behaviors observed in Figure 13 can be categorized into three cases. In the most typical case, increasing the batch size diminishes performance improvements (e.g., TC4 layer of GAN) since the fixed-size LHB ends up handling relatively smaller fraction of workspace. The second case is when increasing the batch size causes little changes to the performance (e.g., C5 layer of YOLO). In this case, the batch size of 8 is already too large for the 1024-entry LHB to cover effectively. The last case shows increasing performance improvements for larger batches (e.g., C1 layer of ResNet). This situation arises when the effective coverage of LHB is greater than the size of workspace, and thus increasing the batch size results in performance enhancements.

G. Network-Level Performance

In the prior evaluations, the results were all shown per convolutional layer. In fact, a DNN is constructed as a combination of various forms of convolutional layers as listed in Table I. Figure 14 shows the total execution time of three representative DNNs. Pooling and softmax layers are not shown in the figure because they account for infinitesimally small fraction of execution time and thus are invisible when plotted in the figure. Since the DNNs spend most of execution time on processing convolution operations, accelerating the convolutions in Duplo by eliminating redundant memory accesses is directly translated into performance improvements for both inference and training of DNNs. On average, Duplo

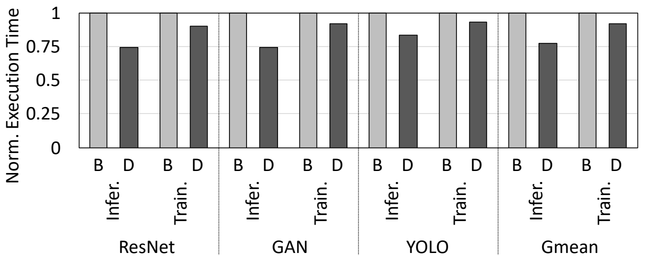


Fig. 14: Comparison of network-level execution time between the baseline (**B**) and Duplo (**D**). On average, Duplo reduces the execution time for inference and training by 22.7% and 8.3%, respectively.

decreases the execution time of DNNs by 22.7% and 8.3% for inference and training executions, respectively.

H. Energy and Area Overhead

Duplo introduces a load history buffer and ID generator to detect data duplication. The addition of new components in the GPU pipeline incurs energy and area overhead, but in the meantime bypassing redundant memory accesses helps Duplo conserve energy in the memory hierarchy. Considering only on-chip components (i.e., registers, caches, and detection unit of Duplo), Duplo results in 34.1% energy reduction for 0.77% area overhead compared to the size of register file. Most of energy saving comes from skipping unnecessary traversals in the memory hierarchy except for the L1 cache since Duplo simultaneously looks up both L1 cache and LHB to hide the access latencies.

VI. Conclusion

The proposed Duplo architecture tackles the data duplication problem of GEMM-based convolution using tensor cores. By employing a novel detection mechanism that identifies the uniqueness of convolution data, Duplo effectively replaces redundant memory accesses with simple register renaming. The following summarizes key insights provided by Duplo.

- Duplo saves a good portion of off-chip DRAM transactions and L2 cache accesses and turns them into in-SM register renaming, which help improve performance and reduce energy in GPUs.
- A simple direct-mapped buffer is sufficient to filter redundant tensor-core-load instructions because of their sequential access patterns. The buffer size makes a tradeoff between performance and overhead, and we leave it as a design option.
- Increasing the number of channels or batch images does not necessarily hurt the performance of Duplo. Experiment results prove that Duplo can effectively handle large workspace data.

Acknowledgment

This research was supported by Samsung Research Funding and Incubation Center of Samsung Electronics under the project #SRFC-IT1801-04. William J. Song is the corresponding author of this paper.

References

[1] K. Chellapilla, S. Puri, and P. Simard, "High Performance Convolutional Neural Networks for Document Processing," *International Workshop on Frontiers in Handwriting Recognition*, Oct. 2006, pp. 1–7.

[2] Y. Chen, J. Emer, and V. Sze, "Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks," *ACM/IEEE International Symposium on Computer Architecture*, June 2016, pp. 367–379.

[3] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cuDNN: Efficient Primitives for Deep Learning," pp. 1–9, Oct. 2014, [Online], Available: https://arxiv.org/abs/1410.0759.

[4] M. Cho and D. Brand, "MEC: Memory-Efficient Convolution for Deep Neural Network," *International Conference on Machine Learning*, Aug. 2017, pp. 815–824.

[5] S. Collange, D. Defour, and Y. Zhang, "Dynamic Detection of Uniform and Affine Vectors in GPGPU Computations," *International Conference on Parallel Processing*, Aug. 2009, pp. 46–55.

[6] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," *IEEE Conference on Computer Vision and Pattern Recognition*, June 2016, pp. 770–778.

[7] P. Hill, A. Jain, M. Hill, B. Zamirai, C. Hsu, M. A. Laurenzano, S. Mahlke, L. Tang, and J. Mars, "DeftNN: Addressing Bottlenecks for DNN Execution on GPUs via Synapse Vector Elimination and Near-compute Data Fission," *IEEE/ACM International Symposium on Microarchitecture*, Oct. 2017, pp. 786–799.

[8] H. Jeon, G. S. Ravi, N. S. Kim, and M. Annavaram, "GPU Register File Virtualization," *IEEE/ACM International Symposium on Microarchitecture*, Dec. 2015, pp. 420–432.

[9] Y. Jeon and J. Kim, "Constructing Fast Network through Deconstruction of Convolution," *Advances in Neural Information Processing Systems*, Dec. 2018, pp. 5951–5961.

[10] D. Jones, "The Arithmetic Tutorial Collection," pp. 1–4, Nov. 2011, [Online], Available: http://homepage.divms.uiowa.edu/~jones/bcd/.

[11] M. Khairy, A. Jain, T. Aamodt, and T. Rogers, "Exploring Modern GPU Memory System Design Challenges through Accurate Modeling," pp. 1–10, Oct. 2018, [Online], Available: https://arxiv.org/abs/1810.07269.

[12] M. Khairy, A. Jain, T. Aamodt, and T. Rogers, "A Detailed Model for Contemporary GPU Memory Systems," *IEEE International Symposium on Performance Analysis of Systems and Software*, Mar. 2019, pp. 141–142.

[13] H. Kim, H. Nam, W. Jung, and J. Lee, "Performance Analysis of CNN Frameworks for GPUs," *IEEE International Symposium on Performance Analysis of Systems and Software*, Apr. 2017, pp. 55–64.

[14] J. Kim, C. Torng, S. Srinath, D. Lockhart, and C. Batten, "Microarchitectural Mechanisms to Exploit Value Structure in SIMT Architectures," *ACM/IEEE International Symposium on Computer Architecture*, June 2013, pp. 130–141.

[15] K. Kim and W. W. Ro, "WIR: Warp Instruction Reuse to Minimize Repeated Computations in GPUs," *IEEE International Symposium on High Performance Computer Architecture*, Feb. 2018, pp. 389–402.

[16] T. Kim, M. Cha, H. Kim, J. Lee, and J. Kim, "Learning to Discover Cross-Domain Relations with Generative Adversarial Networks," *International Conference on Machine Learning*, Aug. 2017, pp. 1857–1865.

[17] A. Krizhevsky, I. Sutskever, and G. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," *Advances in Neural Information Processing Systems*, Dec. 2012, pp. 1097–1105.

[18] A. Lavin and S. Gray, "Fast Algorithms for Convolutional Neural Networks," *IEEE Conference on Computer Vision and Pattern Recognition*, June 2016, pp. 4013–4021.

[19] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh, "Basic Linear Algebra Subprograms for Fortran Usage," *ACM Transactions on Mathematical Software*, vol. 5, no. 3, pp. 308–323, Sept. 1979.

[20] C. Li, Y. Yang, M. Feng, S. Chakradhar, and H. Zhou, "Optimizing Memory Efficiency for Deep Convolutional Neural Networks on GPUs," *International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov. 2016, pp. 633–644.

[21] S. Li, J. Ahn, R. Strong, J. Brockman, D. Tullsen, and N. Jouppi, "McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures," *IEEE/ACM International Symposium on Microarchitecture*, Dec. 2009, pp. 469–480.

[22] 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," *IEEE International Symposium on Performance Analysis of Systems and Software*, Mar. 2019, pp. 293–303.

[23] S. Markidis, S. Chien, E. Laure, I. Peng, and J. Vetter, "NVIDIA Tensor Core Programmability, Performance, and Precision," *IEEE International Parallel and Distributed Processing Symposium Workshops*, May 2018, pp. 522–531.

[24] M. Mathieu, M. Henaff, and Y. LeCun, "Fast Training of Convolutional Networks through FFTs," *International Conference on Learning Representation*, Apr. 2014, pp. 1–9.

[25] NVIDIA, "NVIDIA Tesla V100 GPU Architecture: The World's Most Advanced Data Center GPU," NVIDIA White Paper, WP-80608_v1.1, pp. 1–52, Aug. 2017.

[26] NVIDIA, "NVIDIA Turing GPU Architecture," NVIDIA White Paper, WP-09183-001_v01, pp. 1–79, Sept. 2018.

[27] NVIDIA, "NVIDIA Cuda C++ Programing Guide," pp. 1–346, Nov. 2019, [Online], Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide.

[28] NVIDIA, "NVIDIA Deep Learning SDK Documentation," pp. 1–52, Aug. 2019, [Online], Available: https://docs.nvidia.com/deeplearning/sdk.

[29] Y. Oh, M. K. Yoon, W. J. Song, and W. W. Ro, "FineReg: Fine-Grained Register File Management for Augmenting GPU Throughput," *IEEE/ACM International Symposium on Microarchitecture*, Oct. 2018, pp. 364–376.

[30] W. Qian, Z. Xianyi, Z. Yunquan, and Q. Yi, "AUGEM: Automatically Generate High Performance Dense Linear Algebra Kernels on x86 CPUs," *International Conference on High Performance Computing, Networking, Storage and Analysis*, Nov. 2013, pp. 1–12.

[31] A. Radford, L. Metz, and S. Chintala, "Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks," *International Conference on Learning Representations*, Nov. 2016, pp. 1–16.

[32] M. Raihan, N. Goli, and T. Aamodt, "Modeling Deep Learning Accelerator Enabled GPUs," *IEEE International Symposium on Performance Analysis of Systems and Software*, Mar. 2019, pp. 79–92.

[33] J. Redmon and A. Farhadi, "YOLO9000: Better, Faster, Stronger," *IEEE Conference on Computer Vision and Pattern Recognition*, June 2016, pp. 6517–6525.

[34] S. Ren, K. He, R. Girshick, and J. Sun, "Faster R-CNN: Toward Real-Time Object Detection with Region Proposal Networks," *Advances in Neural Information Processing Systems*, Dec. 2015, pp. 91–99.

[35] M. Rhu, N. Gimelshein, J. Clemons, A. Zulfiqar, and S. W. Keckler, "vDNN: Virtualized Deep Neural Networks for Scalable, Memory-efficient Neural Network Design," *IEEE/ACM International Symposium on Microarchitecture*, Oct. 2016, pp. 1–13.

[44] P. Xiang, Y. Yang, M. Mantor, N. Rubin, L. Hsu, and H. Zhou, "Exploiting Uniform Vector Instructions for GPGPU Performance, Energy Efficiency, and Opportunistic Reliability Enhancement," *ACM International Conference on Supercomputing*, June 2013, pp. 433–442.

[36] M. Rhu, M. O'Connor, N. Chatterjee, J. Pool, Y. Kwon, and S. W. Keckler, "Compressing DMA Engine: Leveraging Activation Sparsity for Training Deep Neural Networks," *IEEE International Symposium on High Performance Computer Architecture*, Feb. 2018, pp. 78–91.

[37] P. Sermanet, D. Eigen, X. Zhang, M. Mathieu, R.Fergus, and Y.LeCun, "OverFeat: Integrated Recognition, Localization and Detection using Convolutional Networks," *International Conference on Learning Representation*, May 2013, pp. 1–16.

[38] E. Shelhamer, J. Long, and T. Darrell, "Fully Convolutional Networks for Semantic Segmentation," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 39, no. 4, pp. 640–651, Apr. 2017.

[39] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Layer-Scale Image Recognition," *International Conference on Learning Representations*, May 2015, pp. 1–14.

[40] V. Sze, Y. Chen, T. Yang, and J. S. Emer, "Efficient Processing of Deep Neural Networks: A Tutorial and Survey," *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, Dec. 2017.

[41] K. Vincent, K. Stephano, M. Frumkin, B. Ginsburg, and J. Demouth, "On Improving The Numerical Stability of Winograd Convolutions," *International Conference on Learning Representation*, Apr. 2017, pp. 1–4.

[42] C. Wang, J. Yang, L. Xie, and J. Yuan, "Kervolutional Neural Networks," *IEEE Conference on Computer Vision and Pattern Recognition*, June 2019, pp. 31–40.

[43] K. Wang and C. Lin, "Decoupled affine computation for SIMT GPUs," *ACM/IEEE International Symposium on Computer Architecture*, June 2017, pp. 295–306.

[45] D. Yan, W. Wang, and X. Chu, "Demystifying Tensor Cores to Optimize Half-Precision Matrix Multiply," *IEEE International Parallel and Distributed Processing Symposium*, May 2020, pp. 634–643.

[46] Y. Yang, P. Xiang, M. Mantor, N. Rubin, L. Hsu, Q. Dong, and H. Zhou, "A Case for a Flexible Scalar Unit in SIMT Architecture," *IEEE International Parallel and Distributed Processing Symposium*, May 2014, pp. 93–102.

[47] A. Yazdanbakhsh, K. Samadi, N. S. Kim, and H. Esmaeilzadeh, "GANAX: A Unified MIMD-SIMD Acceleration for Generative Adversarial Networks," *ACM/IEEE International Symposium on Computer Architecture*, June 2018, pp. 650–661.

[48] T. Yeh, R. Green, and T. Rogers, "Dimensionality-Aware Redundant SIMT Instruction Elimination," *International Conference on Architectural Support for Programming Languages and Operating Systems*, Mar. 2020, pp. 1327–1340.

[49] A. Yilmazer, Z. Chen, and D. Kaeli, "Scalar Waving: Improving the Efficiency of SIMD Execution on GPUs," *IEEE International Parallel and Distributed Processing Symposium*, May 2014, pp. 103–112.