International Journal of Networking and Computing – www.ijnc.org, ISSN 2185-2847
Volume 12, Number 2, pages 295–316, July 2022

Data Rearrange Unit for Efficient Data Computation

Akiyuki Mamiya
Keio University
3-14-1 Hiyoshi, Kohoku-ku, Yokohama, Kanagawa 223-8522, Japan

Nobuyuki Yamasaki
Keio University
3-14-1 Hiyoshi, Kohoku-ku, Yokohama, Kanagawa 223-8522, Japan

Received: February 15, 2022
Revised: May 5, 2022
Accepted: May 16, 2022
Communicated by Michihiro Koibuchi

Abstract

Recently, the demand for computation-intensive applications such as multimedia and AI applications has increased. Data-parallel execution units are typically used for calculations in these applications to increase computational throughput. However, the data required for computation may need to be accessed at discontinuous memory addresses, which can reduce computation efficiency.

Generally, normal memory access instructions access data blocks of continuously allocated memory addresses, containing both valid and invalid data for computation. These memory access patterns result in low computation density in the data parallel execution units, wasting computation resources. Therefore, simply increasing the number of data-parallel execution units leads to an increase in wasted computational resources, which will become a significant issue in embedded systems where multiple resource limitations exist. It is essential to improve computational efficiency to perform practical computation in such systems.

This paper introduces a Data Rearrange Unit (DRU), which gathers and rearranges valid computation data between main memory and execution units. The DRU improves the performance of multimedia and AI application by significantly reducing the access rate from/to main memory and increasing computation efficiency. It is applicable to most hardware architectures, and its effectiveness can be further enhanced by the execution unit interface that directly connects the DRU to the execution unit. We demonstrate the effectiveness of our DRU by implementation on the RMTP SoC, improving convolution throughput on a data-parallel execution unit by a maximum of 94 times while only increasing the total cell area by about 12.7%.

Keywords: convolutional neural network, data-parallel, data rearrange

1 Introduction

Recently, AI and graphics applications have become more popular due to advancements in many techniques such as image classification, segmentation, object detection, and video processing techniques. These applications are computation-intensive as they require computation of multiple parameters of neural networks[18][8] to complete their processing. Therefore, data-parallel execution

units which can extract data parallelism are applied to increase their computational efficiency. The Convolutional Neural Network (CNN), a type of neural network with multiple hidden layers to extract features, has become a representative neural network in the field of deep learning[11]. Its parameters have increased in size and density throughout the past years, making the recognition process more complex by increasing the number of neurons. Each neuron is required to execute a computation-intensive task, increasing the demand for computations that require data-parallel execution units, such as vector units and Single Instruction Multiple Data (SIMD) units. Due to such increased computational load, the efficiency of each computation has become an important topic.

A CNN is composed of multiple layers of fully connected layers and convolutional layers to exploit spatial correlations of data. Convolutional layers are the most dominant in terms of computational load, as they require many Multiply-accumulate (MAC) instructions to compute the convolutions[23]. The MAC instructions can be executed on data-parallel execution units to extract data parallelism.

However, when computing these MAC instructions, the computational density of the data-parallel execution units such as SIMD units and vector units can be low, depending on the kernel size. Convolution requires multiple access to the input data array equal to the size of each kernel. Since the adequate data for calculation is allocated in discontinuous memory areas, each computation that can be calculated by the data-parallel execution unit is mainly composed of data that is irrelevant to the result of the computation. Irrelevant data will still be computed for each convolution as data-parallel execution units will mask the irrelevant result after computing it. The computation process results in a waste of computational resources, increasing in proportion to the amount of computation.

In addition, due to discontinuous memory addressing, each convolution calculation requires multiple memory accesses proportional to the kernel size, increasing computation time. Especially in embedded systems, excessive power consumption, silicon area, and memory usage lead to lower overall system utilization because most embedded systems operate on batteries.

Approaches to improve convolution efficiency consist of software and hardware. Software approaches such as im2col[3][20][1] are costly in power, as main memory access for allocating temporary computational data requires multiple memory access. This approach also pollutes a large portion of the main memory with rearranged input data, requiring the system to host a large main memory. A general processing unit cannot easily apply most hardware approaches to execute CNNs, primarily designed for CNN accelerators[5][10][6]. In embedded systems with limited resources in terms of main memory and execution units, the overhead for relieving the complexity in computation can conversely become an additional overhead.

In this paper, we propose a data rearrange unit(DRU) to eliminate the waste of computational resources in the computation of discontinuous addresses. The DRU rearranges the effective data of each computation into continuous memory-accessible data blocks, allowing the applications to perform the computation without wasting computation resources. Memory access for each convolution is performed by the DMAC for very efficient access to the input data array, rather than by the load instructions of the processor, to reduce total computation time. Additionally, the DRU provides three schemes of accessing rearranged data for application users to choose from as needed. One is the execution unit interface that allows the corresponding execution unit to access the rearranged data for its computation directly. The DRU requires only minor hardware modifications to the execution unit to achieve highly efficient computation without the use of slow load/store instructions. The data rearrangement and arithmetic data execution are performed by all hardware units in parallel, by pipelining each unit. Users of the DRU only need to set the convolution parameters by software and enable DRU usage to perform their optimal convolution with high efficiency.

We evaluate our DRU design on multiple CNN image sets to consider its impact on convolutions. We also evaluate the impact of each of the three DRU interfaces and buffer size, which holds rearranged data. The DRU is implemented on the SRMTP[15], an 8-way prioritized Simultaneous Multi-Threading (SMT) processor for embedded real-time systems. We use the vector unit implemented on the SRMTP as the target data-parallel execution unit. The proposed DRU improves the convolution operation throughput of the ImageNet data set by up to 94 times with only an approximate 12.7% increase in silicon area of the SRMTP. The contribution of this paper is the first proposal of a DRU, a compact hardware unit that reduces the overhead caused by processing

load/store instructions, and to confirm its effectiveness by actually designing and implementing it on an SoC.

The remainder of this paper is organized as follows. Section 2 provides background on CNNs and computations of discontinuously addressed data on parallel execution units. Section 3 introduces related works of software and hardware methods for proficient computation of CNNs, relieving the amount of memory access and computation. Section 4 proposes the Data Rearrange Unit (DRU) for efficient computation and introduces the design and implementations. Section 5 shows the implementation execution results of the DRU with its customizations against multiple CNN image sets. Section 6 concludes the paper and discusses future research on the design and implementations of the DRU.

2 Background

AI and graphics applications, including object recognition, image processing, and video processing, have risen in popularity, increasing the demand for quick and effective ways to compute them. Deep learning techniques which use Neural Networks (NN) are applied to these applications and have shown exemplary performance. Especially, a kind of NN, the Convolutional Neural Network (CNN), has been highly influential throughout the past years. We will introduce CNNs, problems when computing CNNs, and the target processor for our proposed unit.

2.1 CNN

CNN is a feed-forward neural network composed of convolutional layers and fully connected layers to extract abstraction and features of the input data array. As shown in Fig. 1, the CNN is composed of input, hidden, and output layers. The matrix at each layer has three dimensions: width, height, and channel. Parameters for convolution are shown in Table 1. A kernel is a filter to extract the features from the input data, being a matrix that slides over the input data and performs dot product computations, generating an abstract feature map as the output. The dot-product computations are translated into Multiply-Accumulate (MAC) operations within data-parallel execution units.

Table 1: CNN parameters

Parameter	Description
H_{in}/W_{in}	Height/Width of Input Data
H_{out}/W_{out}	Height/Width of Output Data
C_{in}/C_{out}	Channel Number of input/output map
k_{size}	Kernel Size

2.2 Parallel Execution of Discontinuous Addressed Data

In the convolutional layer of the CNN, every convolution computation requires access to the input data array. A matrix equal to the size of the kernel data array is accessed from memory to compute each convolution. However, since an array is allocated from width to height order inside the memory, the valid data within the matrix are allocated in discontinuous memory addresses. General memory access operations can only access data in bulk from a contiguous memory area to a computation register, making it difficult to access only the valid data. Other common memory access patterns in data-parallel execution units include stride memory access operations, which can access discontinuous addresses stored in a set stride amount interval. However, the valid data in CNNs are challenging to access with stride memory access as the valid data are not allocated in addresses at a fixed offset from one another. Stride memory access also suffers from power consumption and access latency due to activating multiple rows in the DRAM.

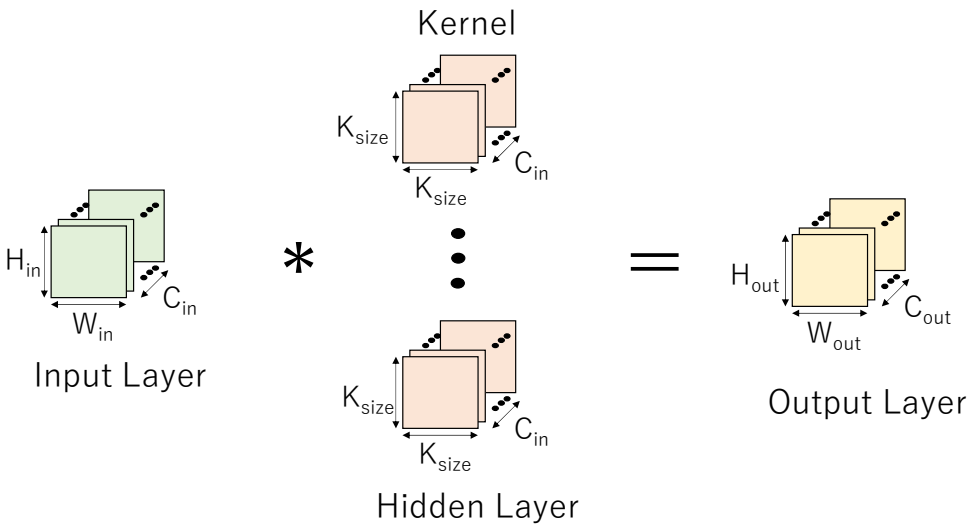

Figure 1: Convolutional Layer Computation

For example, in Fig. 2, when k_{size} is 3, the valid data of the input data array are shown in blue, requiring a total of three 256-bit memory line accesses to compute each convolution, resulting in multiple wasteful memory access per convolution. For these computations, the number of memory access M_c to execute the convolution for channel C_{in} is referred in Equation 1. The number of memory access increases proportionally to the k_{size}.

$$M_c = k_{size} \times (W_{in} \times H_{in}) \times C_{in} \tag{1}$$

Additionally, with general memory access in convolution, computation resource is wasted for every invalid data in each memory access. Computation on data-parallel execution units requires the simultaneous execution of multiple data equal to its wide execution register width. The amount of valid data in the execution line width is vital for high-density data-parallel computation performance. In the case of convolutions, the execution registers contain a considerable amount of invalid data per computation.

Fig. 3 shows an example of a SIMD execution with a 256-bit execution line size, which is one of the typical data-parallel executions. Only a total of three words (96 bits) of valid blue data is essential for this calculation, and the remaining five words (160 bits) of white data are wasted. At each iteration, the data-parallel execution unit computes all the data inside the execution registers, and the invalid execution results are masked. Therefore, each computation wastes execution time and performs an inefficient low-density execution. Especially for small kernel size executions, the invalid data amount of the execution register is high, resulting in more wasted computation resources.

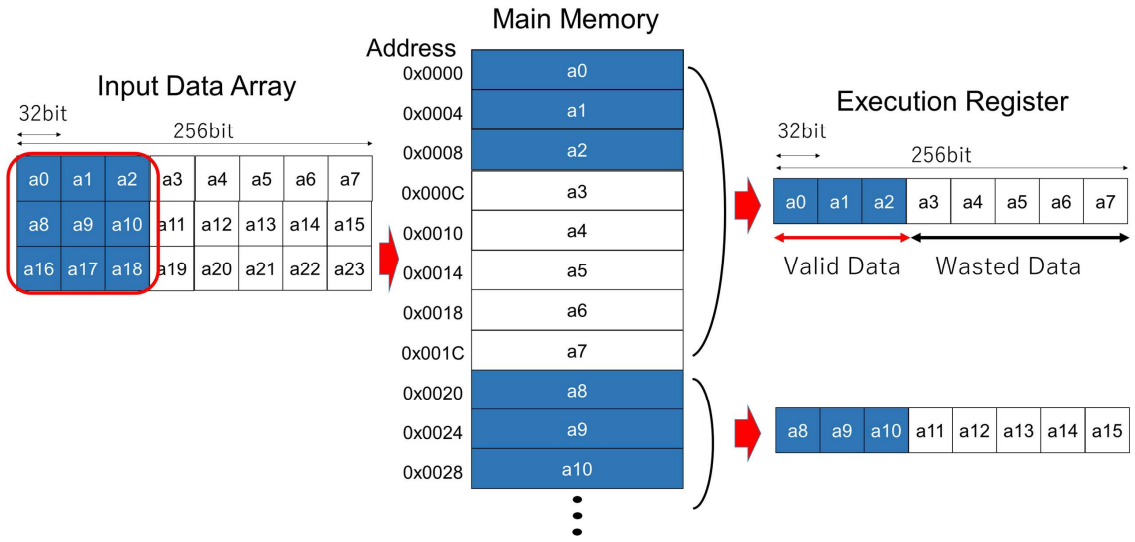

Figure 2: Convolution of Valid Data

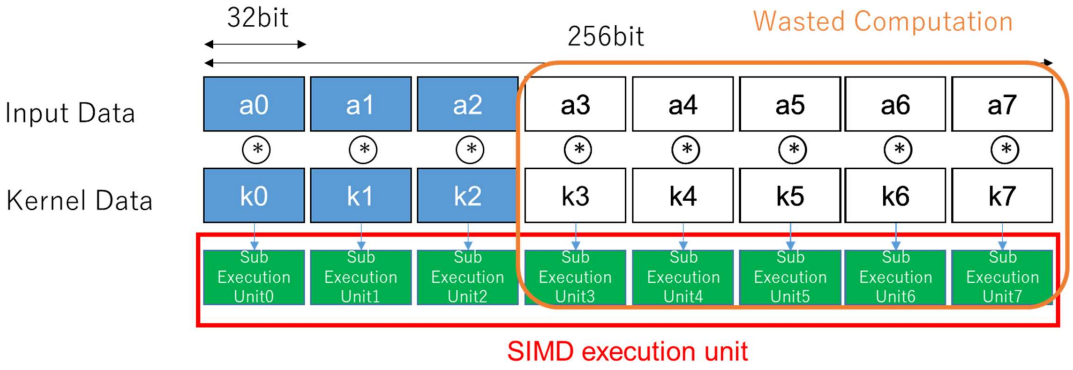

Figure 3: Wasted Execution in Data Parallel Execution Unit

3 Relative Work

Complex computations requiring access to discontinuous memory addressed data results in low usage of data-parallel execution units per computation. In order to improve memory access time and computation density, we consider related works on the optimization of convolution executions. Each considered approach can be categorized into software and hardware approaches, with the former focused on data reallocation and the latter focused on computation-specific accelerators.

3.1 Software Approach

One software approach relieving the access rate for the computations is im2col[3][20][1]. Im2col extracts data blocks of size k_{size}^2 out of the input data array for each position where the convolution takes place and rearranges the data inside memory. The rearrangement allows for convolution to be computed as a single MAC operation with a length equal to k_{size}. Valid data of the matrix multiplication are allocated in the continuous memory address, resulting in only single memory access for each computation.

However, the major drawback of the method is the amount of working memory allocated inside main memory during preprocessing. Number of bytes B required to be allocated for this approach, when a single input data is byte sized, is shown in Equation 2.

$$B = (k_{size}^2)(\frac{W_{in} - ksize}{S} + 1)(\frac{W_{in} - ksize}{S} + 1)C_{in} \tag{2}$$

Increase in W_{in}, H_{in}, and k_{size} drastically increases working memory allocation size. The allocation becomes a significant overhead for embedded systems with limited memory sizes. In this method, memory reallocation is performed by load and store instructions by the processor, polluting the data cache in the process.

Other software approaches to accelerate CNN operation include optimization techniques such as the reordering of the activation layers and pooling layers for fewer operations in the network by Daultani et al. [7]. Wang et al.[21] reuse smaller image size computations for computation in constrained embedded systems. Also, Liu et al.[13] achieve computation performance optimization by removing redundant kernels on computation. As most of the computational load is concentrated in the convolutional layers, relieving this convolutional burden will significantly improve CNN performance. However, the baseline convolution operations must be executed nonetheless, causing these approaches to have a minimal impact on performance compared to the rearrangement of data.

As described above, the im2col approach has a disadvantage that a large amount of memory must be temporarily allocated. We solve this problem and design the DRU so that it does not pollute main memory. Other software approaches require the processor to use load/store instructions when allocating memory and preparing processing data, which incurs a large overhead before the actual computation. On the other hand, the DRU is designed so that load/store instructions are not required, thus reducing the overhead.

3.2 Hardware Approach

For hardware approaches, many CNN accelerators designed for AI applications have been proposed[4]. The accelerator units mainly comprise data-parallel computation units and storage units for effective data allocation for each computation. Implementation of usage-specific architecture or Field Programmable Gate Arrays (FPGAs) is the current trend making these implementations costly and specific to architecture. An example of a specific architecture is the SPE[22] which prunes redundant kernel neurons to reduce computation amount. Another example is the permutation blocks using architecture[12] which reuses two adjacent convolutions per data window to improve power efficiency. Other approaches implement a usage-specific accelerator which includes the use of a systolic array[10], a spatial architecture[5], or multiple chips[6]. Simple embedded processors with cost and flexibility limitations cannot apply these high-cost hardware approaches, which require a change in the processing unit.

As many embedded systems are battery powered, hardware accelerators may not be power efficient for systems with power constraints[2]. Misko et al.[14] achieve extensible embedded processors for CNN computations with minimal addition to existing microprocessor hardware. Custom SIMD instructions of small convolution size 3×3 are added to increase the data parallelism of the SIMD units. The addition of new instructions to the general processor requires hardware decoders and compiler changes. For software-codesigned processors with hardware and software co-design, these changes may be sufficient; however, for most processors, the addition of instruction requires too much burden for software co-designed processors.

The DRU proposed in this paper unlike the proposed CNN accelerators proposed above, the DRU proposed in this paper is a simple architecture that can be applied to many types of processing units and applications. It is also applicable to many types of hardware, because it does not require a new instruction set to be added to the CPU.

4 Proposed Data Rearrange Unit

Low-density executions in data-parallel execution units degrade computation efficiency. Each execution wastes resources and power consumption, which is problematic in systems with constraints such as embedded systems. The requirement for an efficient CNN system is applicability to most architectures without significant hardware change. Therefore, we propose a Data Rearrange Unit (DRU) for high-density data-parallel executions and low memory access per computation.

The DRU is a new functional unit that gathers and rearranges the valid computation data between main memory and the execution unit. It is connected to main memory via memory bus to access the input data array of the convolution, as shown in the block diagram in Fig. 4. The DRU improves the spatial locality of valid computation data in neural network applications with the rearrangement unit inside. It consists of three main sub-blocks, including Direct Memory Access Controller (DMAC), Data Composer (DC), and Data Decomposer (DD). Each sub-block is connected to Dual Port Register Files (DPRFs), working memory to store temporary data used for rearrangement. We will introduce each sub-block and its functions throughout this section.

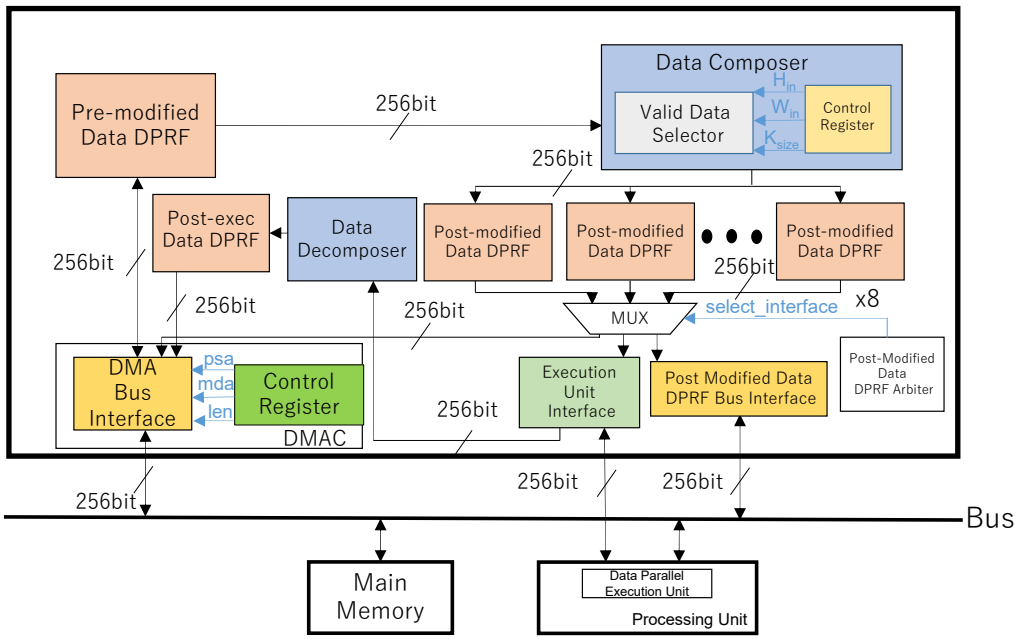

Figure 4: Block Diagram of Data Rearrange Unit

4.1 Direct Memory Access Controller (DMAC)

The DMAC sub-block is designed and implemented on the DRU as an interface between main memory and the DRU. Input data array allocated inside main memory is accessed in bulks of data by burst access. Burst access increases the data access rate by clock cycle proportionally to the burst size. Also, compared to main memory access with load instructions, DMAC function utilization does not corrupt data cache in the process. Therefore, the usage of DMAC has benefits for main memory access over the load instructions of the processor.

Port/Source address (psa), memory/destination address (mda), and data length (len) are set from software to control its access to main memory. Input data arrays accessed by the DMAC are buffered inside the Pre-Modified Data Dual Port Register File (DPRF) for temporary allocation to be used for rearrangement. Rearrangement efficiency is increased by allocating the input data array inside the DRU, as multiple access to main memory will result in stalls of the rearrangement and computation processes due to memory access. The DMAC can burst access main memory up to the maximum size of the Pre-Modified Data DPRF, where valid bits indicating rearrangement of valid data are applied to each line of the DPRF to control the temporary data usage. Once all valid data bits are disabled, another burst access is initiated if access to the whole data array is not complete. The whole input data array in the neuron is accessible up to the set data length.

4.2 Data Composer (DC)

The DC sub-block rearranges the input data array stored in the Pre-Modified Data DPRF. The DC implements a set of control registers used as rearrangement parameters, including the array size, W_{in} and H_{in}, as well as the kernel size k_{size}.

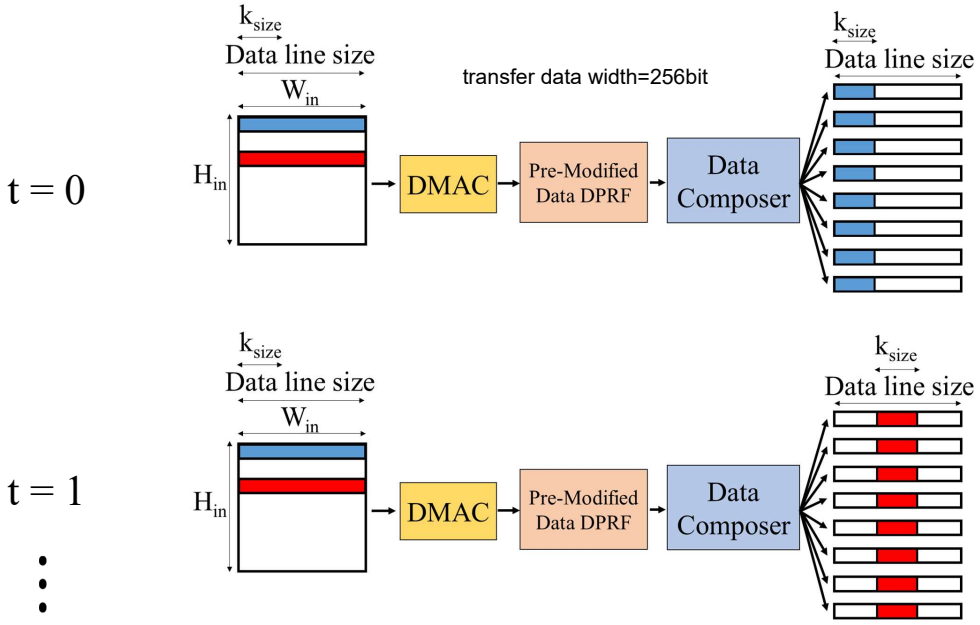

Figure 5: Data Rearrrange Pipeline

The Valid Data Selector (VDS) refers to the parameters in the control registers to extract valid data in parallel from a single-wide data line that consists of multiple words in each clock cycle. Fig.

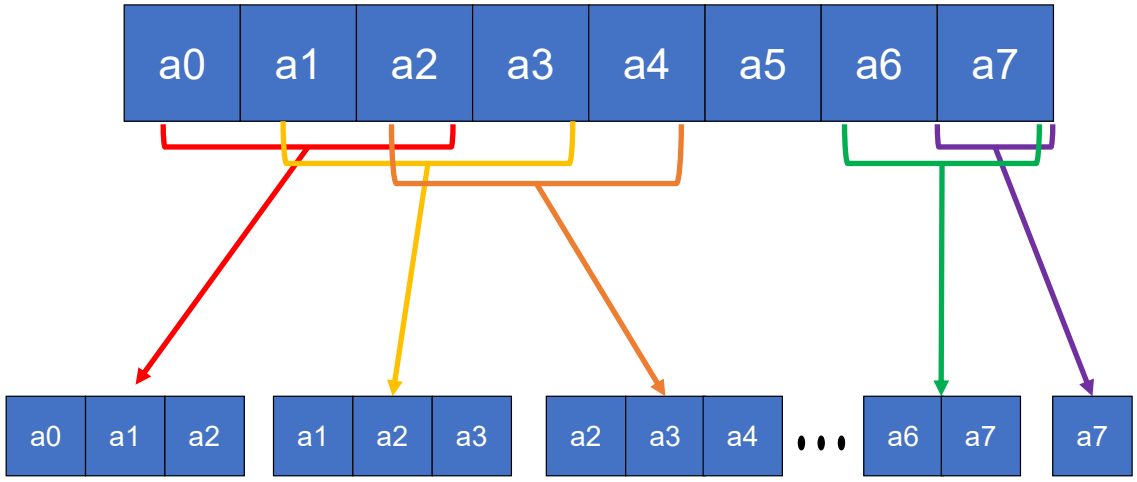

Figure 6: Concept of Valid Data Extraction

5 shows the data rearrange flow and Algorithm. 1 shows the pseudo code for the data rearrange process. The DRU divides and implements this algorithm in each piece of hardware and processes it in parallel in a pipelined manner, dramatically increasing the throughput of operations. A data line in the Pre-Modified Data DPRF is read to rearrange every clock cycle. While the clock cycle does not reach H_{in}, the kernel feature map slides along the input data array, causing the valid data in each convolution to shift accordingly. This causes the same data line to be accessed multiple times for each convolution. The overhead of multiple access to the same data line is relieved by extraction of valid data in parallel per clock cycle as shown in Fig. 6. When stride equals 1, a maximum of eight valid data blocks can be extracted from the data line. The data line does not need to be accessed for each convolution matrix, reducing the hardware clock cycle required for rearrangement up to an eighth in this case. For valid data that passes over two data lines, a portion is read from the data line read first and combined to the data line accessed on the following clock line. The whole process reduces the clock cycle of the rearrangement process compared to software approaches.

Algorithm 1 Data Rearrange Process

Input: $W_{in}, H_{in}, DataArray[H_{in}][W_{in}], k_{size}$

$i = 0$
$clock = 0$
$ComposedArray[W_{in} - k_{size} + 1][k_{size} \times H_{in}]$
while $clock < H_{in}$ **do**
  for $i \leq W_{in} - k_{size} + 1$ **do in parallel**
    $ComposedArray[i][clock : clock + k_{size} - 1] \leftarrow DataArray[clock][i : i + k_{size} - 1]$
  end for
  $clock = clock + 1$
end while

According to k_{size}, the valid data in each data line is extracted by the VDS and stored in the

Post-Modified Data DPRF. On the next clock cycle, the data line of the input data array shifts in the height direction. Extracted data is stored in the continuous address of the Post-Modified Data DPRF. The rearrangement process continues until a valid data block is generated equal to the kernel feature map's size. The rearrangement process and access to main memory by the DMAC become pipelined by continuously accessing the input data array from the Pre-Modified Data DPRF. The DRU reduces the memory access time to M_d as referred in Equation 3 decreasing memory access by $1/(1 + k_{size})$.

$$M_d = \frac{1 + k_{size}}{k_{size}}(W_{in} \times H_{in}) \times C_{in} \tag{3}$$

Rearranged data inside the Post-Modified Data DPRF can be transferred back to main memory by the DMAC. For each data line stored in the Post-Modified Data DPRF, a transfer is acknowledged by the DMAC. The DMAC prioritizes a burst memory access on transfers with lengths larger than the burst size. Once all rearranged data are transferred, a hardware interruption is activated to stop the DRU process.

For rearranges of k_{size} greater than 1, computation with rearranged data by the DRU improves computation efficiency by vastly reducing memory access time. Rearranged data is selected and applied for computation in the processing unit by an arbiter in three possible ways. The first one is to send the data back to main memory via the DMAC. Burst access allows a bulk write back of rearranged data in a few hardware clock cycles. However, transferring rearranged data back to main memory will pollute a large portion of it, wasting memory resources.

Therefore, to bypass these issues and further accelerate the rearrange and computation process, we design and implement a Post-Modified Data DPRF bus interface as a second way to access. The Post-Modified Data DPRFs are address mapped and accessible by the processing unit via the bus, reducing the number of memory accesses to main memory before vector execution. The rearrangement process and DPRF access are done in parallel, reducing execution time.

The prior two access modes of rearranged data still result in large execution times as load instructions are a huge overhead. As a third option, we design and implement an execution unit interface for the execution unit to directly access rearranged data. Fig. 7 shows the changes that had to be made to the execution unit. In the convolution process, the execution unit interface selects the rearranged data in the Post-Modified Data DPRF from the upper left most array data. Selected data will pass through an exclusive path to the execution unit, to be written into the reserved regions of the execution register. Execution units can compute with the rearranged data immediately, reducing the overhead of memory access on load instructions. Computation results are transferred back to the DRU by the writeback unit, which monitors valid computation results of the execution unit. The execution interface has control registers set from software to enable or disable its usage. Minor changes to the execution unit decreases memory access rate M_e, as shown in Equation 4. The memory access required is the input data array with DMAC to compute the whole convolution.

$$M_e = \frac{W_{in} \times H_{in}}{k_{size}} \tag{4}$$

4.3 Data Decomposer (DD)

Results of computation transferred to the execution unit interface are passed on to the DD sub-block. Computation results are stored in the Post-Exec Data DPRF to be burst transferred back to main memory. In order to utilize the burst memory access, result data are rearranged to match a single wide data line. An offset is applied to each data to compress the data into the designated data line. When the DRU becomes the bus master, results are written back to the main memory from the Post-Exec Data DPRF.

The execution unit interface reduces memory access rate by cutting the use of load and store instructions per computation. Rearrangement in the DC and the computations in the execution unit are pipelined to perform multiple steps in the same clock cycle. The DRU computes without accessing main memory after the rearrange process to avoid cache pollution and memory resource

usage. DRU can write back rearranged data via the DMAC in any simple architecture and improve computation efficiency with the execution unit interface.

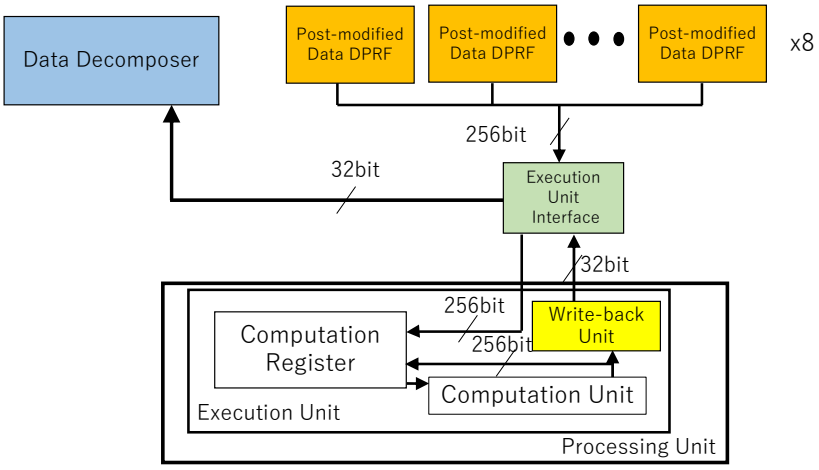

Figure 7: Execution Unit Interface

5 Evaluation

In this section, we evaluate the computation efficiency and cell area of the DRU to discuss its capabilities and limitations. The DRU performance for each use of interfaces and DPRF size is considered.

5.1 Targeted Processor

We apply our proposed hardware unit, the Data Rearrange Unit (DRU) covered in Chapter 4 to the Space Responsive Multithreaded Processor (SRMTP) SoC[15]. The overview of the SRMTP is shown in Fig. 8. The SRMTP is an embedded SoC for distributed real-time systems mainly focused on spacecraft control. It implements a Responsive Multithreaded Processing Unit (RMTPU)[15][19], which is an 8-way prioritized SMT processor. Spacecraft control requires real-time execution in which tasks have time constraints and need to be executed in priority order. The RMTPU achieves real-time execution with eight prioritized threads executing simultaneously in priority order, without context switching overhead. Previous generations of the RMTP series have been implemented and put into practical use on distributed real-time systems such as humanoid robots[19].

The RMTPU configurations are shown in Table 2. Vector units that consist of multiple SIMD units with stride access are designed and implemented in the RMTPU to extract data-level parallelism. In this paper, the target data-parallel execution unit is set to the vector unit in the evaluations.

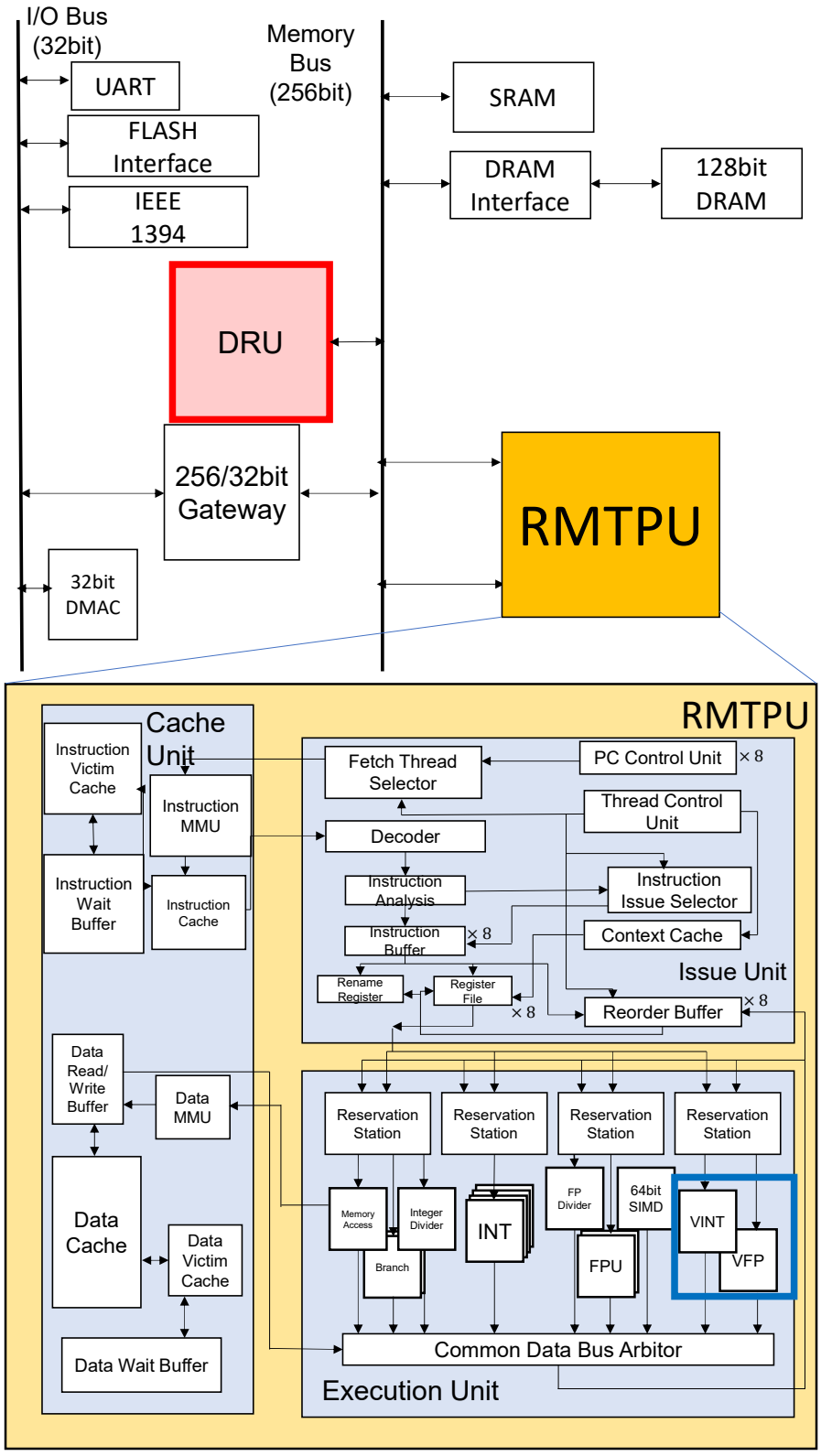

Figure 8: SRMTP Block Diagram

Table 2: RMTPU Configuration

Active Threads	8
Cached Threads	32
Fetch Width	8
Issue Width	4
Commit Width	4
General Purpose Register (GPR)	32 bits × 32 entries × 8 sets
Rename Register (GPR)	32 bit × 64 entries
Floating Point Register (FPR)	64 bits × 8 entries × 8 sets
Rename Register (FPR)	64 bit × 64 entries
ALU	4
Integer Divider	1
FPU	2
FP Divider	1
64bit SIMD Unit	1
Integer Vector Unit	1 (8 SIMD × 2 lanes)
Floating Point Vector Unit	1 (4 FP-SIMD × 2 lanes)
Branch Unit	2
Memory Access Unit	1

The vector unit shown in Fig. 9 is composed of a vector control unit, a vector register unit, and a vector execution unit. The vector control unit controls and dispatches instructions to the execution units. Mask bits for the computation and the vector length of computation are set by software prior to execution. The control unit implements a compound instruction buffer, a RMTP unique feature that can store vector instructions in advance of execution. The conceptual diagram of the compound instruction buffer is shown in Fig. 10. The instructions inside the buffer are treated as one long vector instruction that consists of multiple vector instructions and can be executed using a special RMTP vector instruction: the execute compound instruction (ECI). Once an ECI is issued, the stored vector instructions are executed. Multiple MAC instructions are stored inside the compound instruction buffer in this evaluation.

Vector registers can hold a maximum of 64bit × 512 entries and are reserved by each thread in advance according to the data width of the executing data. A unique reserve instruction is dispatched to reserve vector registers of 128 entries, 256 entries, and 512 entries. Scalar registers have a size of 64bit × 32 entries and are reserved proportionally to vector register reservation size as 8 entries, 16 entries, and 32 entries. Vector and scalar registers are released after use with the release instructions for other threads to use.

Vector execution unit has two pipes of 64bit SIMD with four lanes in parallel. Each pipe has an accumulator utilized when performing MAC operations.

In order to utilize the compound instructions of the vector unit, rearranged data can be transferred from the DRU's execution unit interface sequentially to perform MAC operations. Design and implementation of a transfer detection scheme inside the vector unit allow execution of the instructions stored in the compound instruction buffer without fetching the ECI instruction. This control scheme enables a pipelined execution of the rearrangement process inside the DRU and the execution of MAC instructions, further increasing computation efficiency. In the evaluation, the compound instructions are utilized, reducing the overhead of instruction fetch of computation instructions.

DRU's execution unit interface performance is evaluated by designing and implementing a write back unit for the vector unit. The Computation results of MAC operation are scalar values and stored in the scalar register. Write back unit monitors the data flow of write into the scalar register to access the computation results. Each computation result is transferred back to the DRU. Write back unit usage is controlled by control registers set from software.

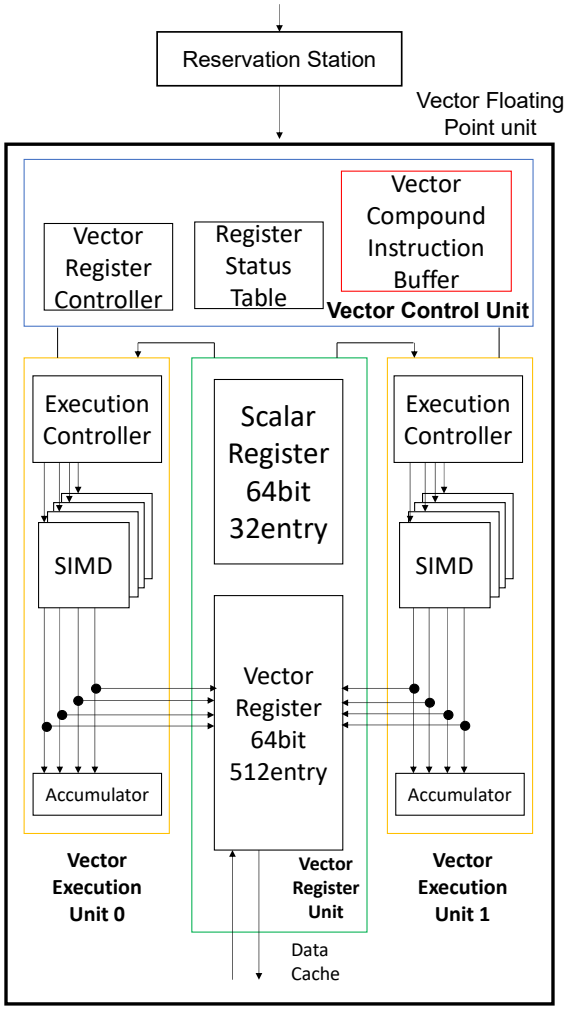

Figure 9: Vector Unit

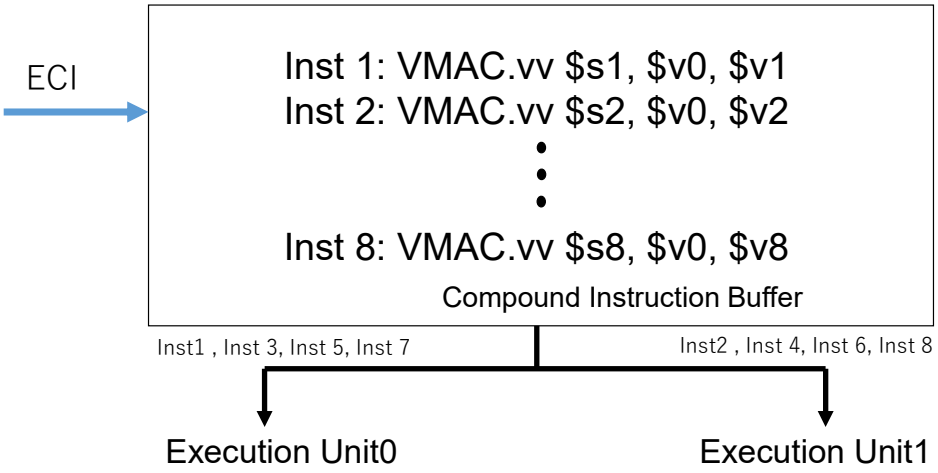

Figure 10: Compound Instruction Buffer

5.2 Experimental Setup

Performance evaluations are conducted on the Cadence XCELIUM Parallel Simulator. Image set benchmarks, CIFAR-10[9], ImageNet[17], and SVHN[16] are selected among those commonly used for evaluation of CNNs in object detection and recognition. ImageNet images are resized to 256×256, and SVHN images are resized to 32×32 to fit the DRAM size of the SRMTP SoC. CIFAR-10 images are set to the default size of 32×32.

We evaluate with various k_{size} of 3 to 7 as used in standard CNN sets and perform convolutions against ten images from each benchmark with 3 RGB channels to align the image count of each benchmark. The size of the Pre-Modified Data DPRF and the Post-Execution Data DPRF are both set to 4KB, large enough to host each image set. The eight Post-Modified Data DPRFs vary in size from a range of 512B to 2KB for each evaluation to evaluate how DPRFs affect performance.

5.3 Experimental Results

We first compare execution performance and memory usage of the rearranged data in the Post-Modified Data DPRF. The arbiter inside the DRU can select the access method of the rearranged data from the execution unit interface, bus interface, and the DMAC transferring data back to main memory. Latter two methods apply to most architectures without any change to the processing unit. The execution performance results of convolution are shown in Fig. 11, Fig. 12, and Fig. 13 for CIFAR-10, ImageNet, and SVHN, respectively. Fig. 11 and Fig. 13 have similar results because both have the image size of 32×32. However, the figures have approximately 1.01 times difference of value in average. A Post-Modified Data DPRF of total size 512B is employed in the evaluation. The bus interface allows any bus master to access the rearranged data for effective computation inside execution units. The use of DMAC inside the DRU to rewrite rearranged data back to main memory has the benefit of reducing the number of pipeline stalls in the DRU by quickly removing the Post-Modified Data DPRF contents. However, it also introduces a source of execution overhead, as it requires the mediation of another memory unit before computation.

Usage of bus interface is dominant for all kernel sizes, with an average of 1.26 times throughput increase compared to the DMAC write back. ImageNet also has a 1.23 times throughput increase showing an extensive benefit of direct access to the rearranged data. k_{size}s with a low value of 3 to 4 only have about an increase of about 1.04 times on average compared to large k_{size} values of 5 to 7 of about 1.31 times. The k_{size} difference is mainly affected by the width of the rearranged data stored in the Post-Modified Data DPRF. The number of values in bulk is equal to k_{size}^2, causing the frequency of DPRF to hit the maximum limit to increase. Once the DPRF is filled, the DMAC must transfer the rearranged data to main memory. The increase in the frequency of the DPRF filling up also increases the frequency of DMA transfer, becoming an overhead for large k_{size}s. Overall, bus interface transfer shows the merit of quick and concise transfer to the execution unit.

Secondly, we compare the effect of Post-Modified Data DPRF size on execution. Fig. 14, Fig. 15, and Fig. 16 respectively show CIFAR-10, ImageNet, and SVHN performance. The DPRF size increase is not as impactful as the execution unit interface, with only an average of 0.001% increase in execution time for ImageNets.

Next, we compare the DRU execution using the bus interface, DMAC, and execution unit interface with computation only utilizing the SRMTP's vector unit. Fig. 17, Fig. 18, and Fig. 19 respectively show CIFAR-10, ImageNet, and SVHN performance. ImageNet at k_{size} 4 with buffer size 2KB improved the execution throughput by a maximum of 94 times. Execution unit interfaces with vector unit, and the compound instructions on CIFAR and SVHN image sets have an average of 18.4 times throughput increase compared to not utilizing it. For ImageNets, this becomes an average of 31.6 times throughput increase. Results show the number of memory access affecting the throughput extensively. The overhead for not using the execution interface adds up on the write of rearranged data to main memory scaling with k_{size}. However, the results show smaller kernel sizes to have a more significant impact in throughput of about 40 times. The decrease in throughput can be considered to be occurring due to the stalls in the pipeline with larger k_{size} as the Post-Modified Data DPRF are more quickly maxed out. The larger image size of ImageNets impacts the throughput more than smaller image sizes, as memory access times drastically increase with image

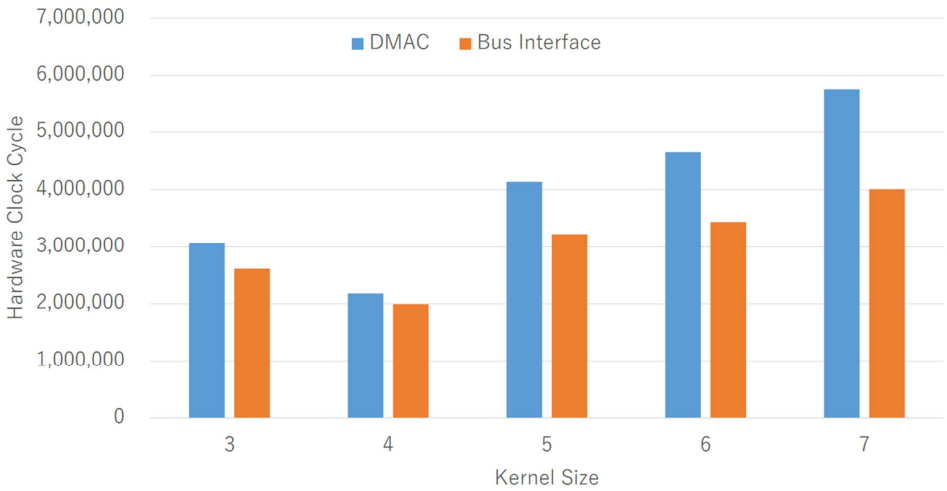

Figure 11: Bus Interface vs DMAC (CIFAR-10)

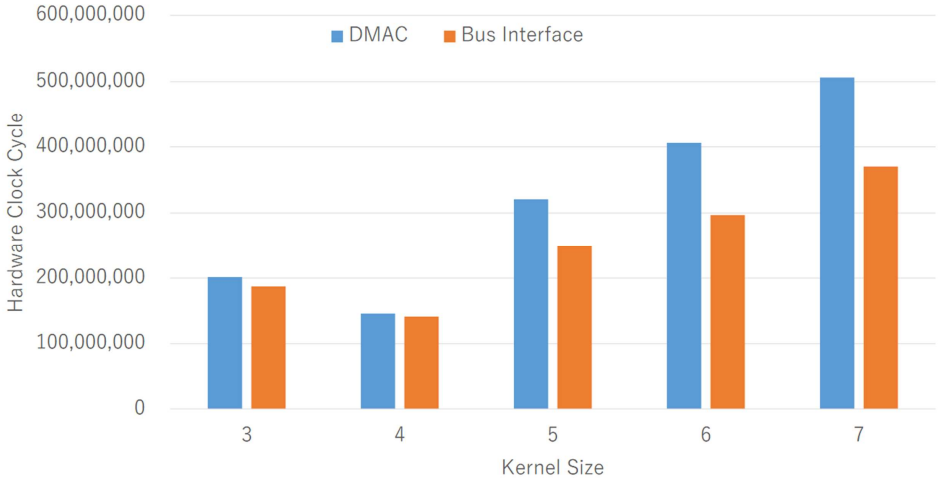

Figure 12: Bus Interface vs DMAC (IMAGENET)

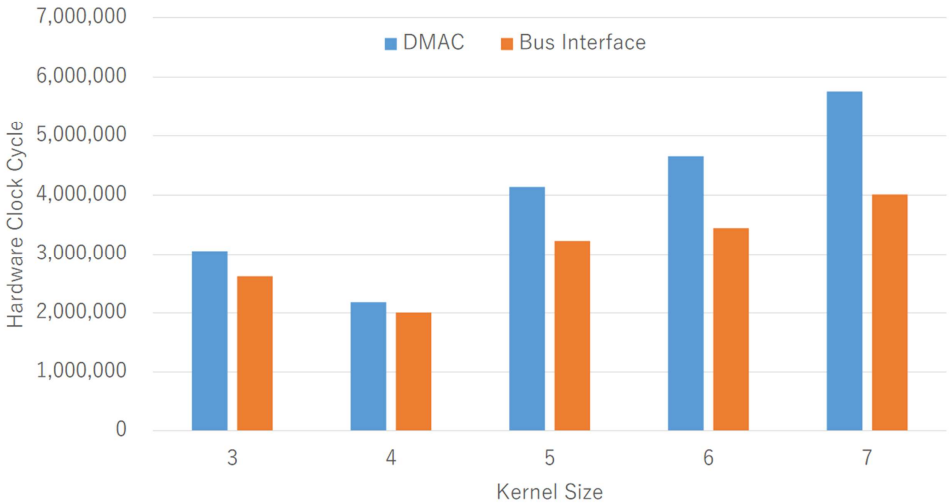

Figure 13: Bus Interface vs DMAC (SVHN)

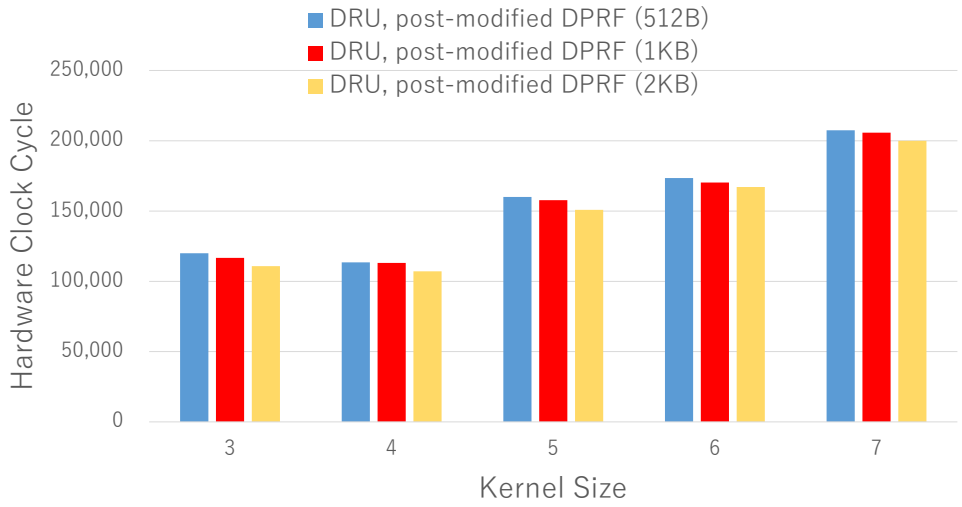

Figure 14: Post-Modified Data DPRF Size(CIFAR-10)

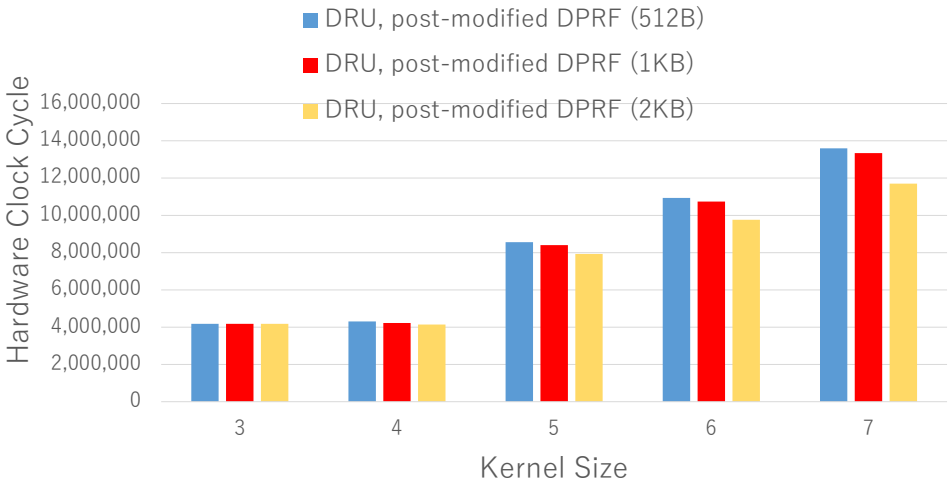

Figure 15: Post-Modified Data DPRF Size (ImageNet)

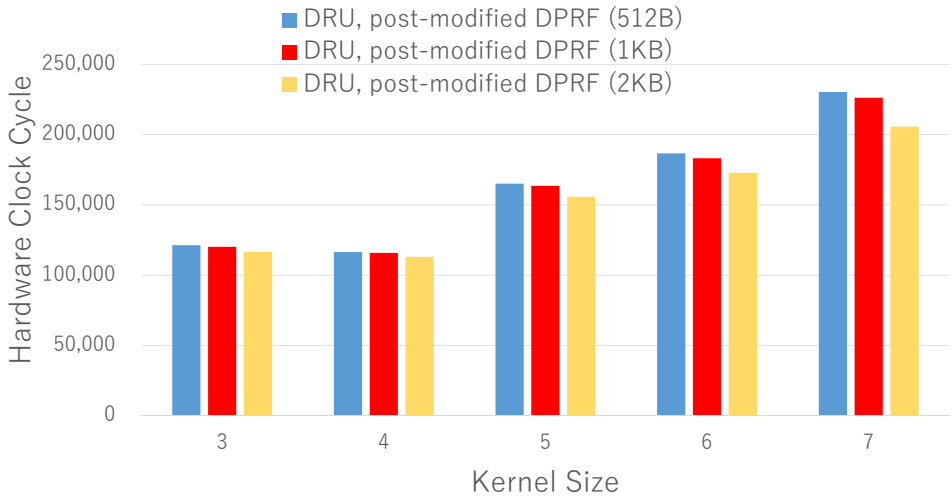

Figure 16: Post-Modified Data DPRF Size (SVHN)

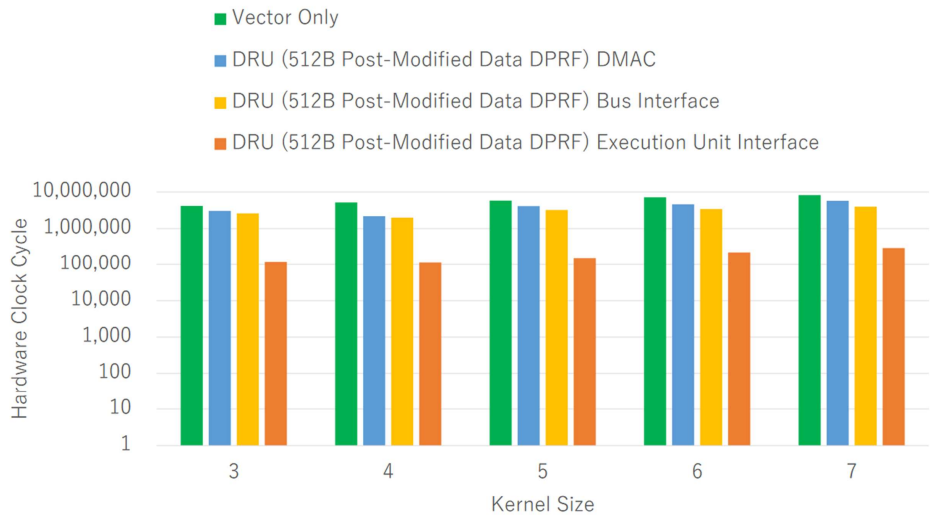

Figure 17: Versus Vector Only (CIFAR-10)

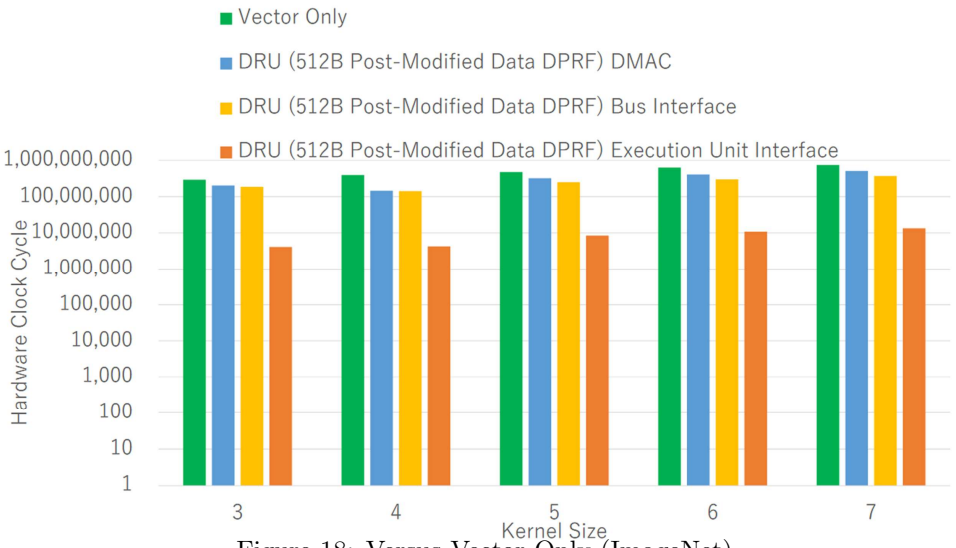

Figure 18: Versus Vector Only (ImageNet)

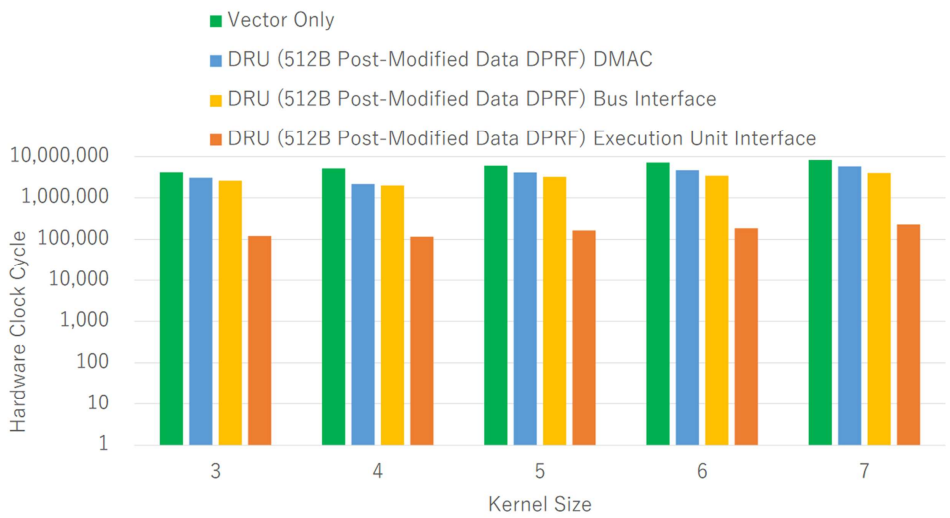

Figure 19: Versus Vector Only (SVHN)

size. Approaches to mitigate the impact of memory access have shown an enormous impact from this evaluation.

Lastly, cell area evaluation is done for the DRU in comparison with the SRMTP SoC cell area in Table 3. Post-Modified Data DPRF is changed in size to analyze the cell area difference when implemented on the DRU. Each configuration of the DRU with differing post-modified data DPRF size is compared in implementation on the SRMTP. 512B Post-Modified Data DPRF covers only 12.7% of the total cell area of the SRMTP, achieving high computation efficiency without much cell area overhead.

Table 3: DRU Cell Area

Module Name	cell area(um^2)	DRU coverage of SRMTP
SRMTP	54,112K	-
SRMTP + DRU (512B post-modified DPRF)	61,978K	12.7%
SRMTP + DRU (1KB post-modified DPRF)	62,325K	13.1%
SRMTP + DRU (2KB post-modified DPRF)	63,026K	14.1%

5.4 Analysis

About half the execution clock cycle of computing with vector only is achieved for DRU execution unit interface disabled. The execution unit interface not only drastically improves execution throughput but also does not use memory resources which increases with larger k_{size}.

The DRU with execution unit interface improves in performance compared to the performance of the vector only results for every k_{size}. We can check the computation time increase proportionally to the k_{size} in the vector only results. Memory access can be seen as a significant overhead affecting computation time, as referred to in Equation 1. The DRU performances of smaller k_{size} of 3 to 5 have an approximately 38 times increase in performance on smaller image sets of CIFAR-10 and SVHN. On ImageNet with a large data size image set, approximately 74 times increase in performance was observed. The performance increases approximately to 40 times on the smaller image sets with the k_{size} of 6 and 7. However, on a large image set of ImageNet, the performance degrades to about 60 times throughput. We can observe that a small data size does not cause as much stall in the rearrangement pipeline with additional reads to main memory using the DMAC. In the case of large data size, the DMAC read often causes a pipeline stall. Additionally, k_{size} affects the occupation rate of the post-modified data DPRF, causing pipeline stall as the data read bandwidth from the execution interface is lower than the rearranged bandwidth.

Smaller Post-modified DRPF sizes of 1KB and 512B affect execution time by an average of 1.08 times for k_{size} of 3 to 5. A smaller DPRF is occupied quickly with larger k_{size} of 6 to 7, increasing the execution time by a few thousand clock cycles, with 1.16 times increase in average. Compared with the execution time of the SRMTP vector only, this increase in a few thousand clock cycles is minor, only a 0.001% increase in execution time.

Therefore, it is optimal to apply a small post-modified DPRF on the SRMTP. Small DPRF results in only 12.7% of the total cell area, making the DRU applicable to simple systems as a hardware unit to increase computation efficiency.

6 Conclusion

The popularity of AI and graphics applications has increased the demand for efficient computations for high throughput. Deep learning techniques which use NNs have been widely applied to the applications and have shown success.

CNN, a type of feedforward NN, extracts features from input data with its multiple convolution layers, which have the most computational load. The output feature maps are computed by convolutions between the input data array and a kernel which slides a set stride amount against all input data. Data-parallel execution units compute convolutions as MAC operations for higher throughput in contemporary architectures.

However, due to the features of convolutions, memory access to a discontinuous memory address is required. General memory access instructions cannot access the whole input data equal to the size of the kernel in one instruction, requiring multiple memory access for one convolution. Additionally, general memory access induces access to invalid data for computation inside data-parallel execution units. Since data-parallel execution units compute every data in parallel, computation resources are wasted on invalid data. Therefore, problems of low computation density and multiple memory access reduce the computation efficiency of convolution. Especially systems such as embedded systems with limited resources suffer from these wasted computational resources.

The Data Rearrange Unit (DRU), a new functional unit, gathers and rearranges the valid data for efficient high-density computations. Rearranged data is placed into continuously accessible data blocks resulting in high-density computations with its use. DRU comprises DMAC, DC, and DD sub-blocks for efficient memory access, rearrangement, and writing back computed data. The DMAC allows burst access to the input data in main memory to be stored in a temporary buffer, improving spatial localization of the rearrangement process.

The DRU prepares three interfaces to access rearranged data, including DMAC, bus interface, and the execution unit interface. The former two achieve easy access to main memory done in parallel to the rearrangement process. It applies to most architectures by only connecting the DRU to the memory bus. The execution unit interface directly bypasses rearranged data to the execution unit for efficient computation not involving load and store operations. Minor changes to the execution unit of adding a writeback unit accomplish a designated data path between the DRU and the execution unit. The DRU's rearrangement process and the execution unit's computations are pipelined to perform highly efficient convolutions with only a minor hardware change. Users of the DRU only need to work around setting convolution parameters to compute any convolution.

Evaluation results show that the DRU improves computation throughput on the SRMTP with execution unit interface by up to 94 times compared to an only vector unit execution. The DRU on average only occupies about 12.7% of the total cell area in the SRMTP, providing a simple architecture applicable in many systems.

Acknowledgement

This research is supported by Adaptable and Seamless Technology transfer Program through Target-driven R&D (ASTEP, AS2815003R) from Japan Science and Technology Agency (JST).

References

[1] Andrew Anderson, Aravind Vasudevan, Cormac Keane, and David Gregg. High-performance low-memory lowering: Gemm-based algorithms for dnn convolution. In *2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 99–106, 2020.

[2] A. Canziani, Adam Paszke, and Eugenio Culurciello. An analysis of deep neural network models for practical applications. *ArXiv*, abs/1605.07678, 2016.

[3] Kumar Chellapilla, Sidd Puri, and Patrice Simard. High performance convolutional neural networks for document processing. 10 2006.

[4] Yiran Chen, Yuan Xie, Linghao Song, Fan Chen, and Tianqi Tang. A survey of accelerator architectures for deep neural networks. *Engineering*, 6, 01 2020.

[5] Yu-Hsin Chen, Tushar Krishna, Joel S. Emer, and Vivienne Sze. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE Journal of Solid-State Circuits*, 52(1):127–138, 2017.

[6] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. Dadiannao: A machine-learning supercomputer. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 609–622, 2014.

[7] Vijay Daultani, Subhajit Chaudhury, and Kazuhisa Ishizaka. Convolutional neural network layer reordering for acceleration. 2016.

[8] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, Los Alamitos, CA, USA, jun 2016. IEEE Computer Society.

[9] Alex Krizhevsky. Learning multiple layers of features from tiny images. 2009.

[10] Lin Li, Jianhao Hu, Qiu Huang, and Wanting Zhou. Bit-serial systolic accelerator design for convolution operations in convolutional neural networks. *IEICE Electronics Express*, 17:20200308–20200308, 10 2020.

[11] Zewen Li, Fan Liu, Wenjie Yang, Shouheng Peng, and Jun Zhou. A survey of convolutional neural networks: Analysis, applications, and prospects. *IEEE Transactions on Neural Networks and Learning Systems*, pages 1–21, 2021.

[12] Dongbao Liang, Jiale Xiao, Yangbin Yu, and Tao Su. *A CNN Hardware Accelerator in FPGA for Stacked Hourglass Network*, pages 101–116. 09 2020.

[13] Chih-Ting Liu, Tung-Wei Lin, Yi-Heng Wu, Yu-Sheng Lin, Heng Lee, Yu Tsao, and Shao-Yi Chien. Computation-performance optimization of convolutional neural networks with redundant filter removal. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 66(5):1908–1921, 2019.

[14] Joshua Misko, Shrikant Jadhav, and Youngsoo Kim. Extensible embedded processor for convolutional neural networks. *Scientific Programming*, 2021:1–12, 04 2021.

[15] Shota Nakabeppu, Yosuke Ide, Masahiko Takahashi, Yuta Tsukahara, Hiromi Suzuki, Haruki Shishido, and Nobuyuki Yamasaki. Space responsive multithreaded processor (srmtp) for spacecraft control. In *2020 IEEE Symposium in Low-Power and High-Speed Chips (COOL CHIPS)*, pages 1–3, 2020.

[16] Yuval Netzer, Tao Wang, Adam Coates, Alessandro Bissacco, Bo Wu, and Andrew Ng. Reading digits in natural images with unsupervised feature learning. *NIPS*, 01 2011.

[17] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

[18] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv 1409.1556*, 09 2014.

[19] Kazutoshi Suito, Rikuhei Ueda, Kei Fujii, Takuma Kogo, Hiroki Matsutani, and Nobuyuki Yamasaki. The dependable responsive multithreaded processor for distributed real-time systems. *IEEE Micro*, 32(6):52–61, 2012.

[20] Yaohung M. Tsai, Piotr Luszczek, Jakub Kurzak, and Jack Dongarra. Performance-portable autotuning of opencl kernels for convolutional layers of deep neural networks. In *2016 2nd Workshop on Machine Learning in HPC Environments (MLHPC)*, pages 9–18, 2016.

[21] Xing Wang, Him Wai Ng, and Jie Liang. Lapped convolutional neural networks for embedded systems. In *2017 IEEE Global Conference on Signal and Information Processing (GlobalSIP)*, pages 1135–1139, 2017.

[22] Maurice Yang, Mahmoud Faraj, Assem Hussein, and Vincent Gaudet. Efficient hardware realization of convolutional neural networks using intra-kernel regular pruning. 03 2018.

[23] Zichao Yang, Marcin Moczulski, Misha Denil, Nando De Freitas, Le Song, and Ziyu Wang. Deep fried convnets. In *2015 IEEE International Conference on Computer Vision (ICCV)*, pages 1476–1483, 2015.