TETRIS: Scalable and Efficient Neural Network Acceleration with 3D Memory

Mingyu Gao Jing Pu Xuan Yang Mark Horowitz Christos Kozyrakis
Stanford University
{mgao12,jingpu,xuany,horowitz,kozyraki}@stanford.edu

Abstract

The high accuracy of deep neural networks (NNs) has led to the development of NN accelerators that improve performance by two orders of magnitude. However, scaling these accelerators for higher performance with increasingly larger NNs exacerbates the cost and energy overheads of their memory systems, including the on-chip SRAM buffers and the off-chip DRAM channels.

This paper presents the hardware architecture and software scheduling and partitioning techniques for TETRIS, a scalable NN accelerator using 3D memory. First, we show that the high throughput and low energy characteristics of 3D memory allow us to rebalance the NN accelerator design, using more area for processing elements and less area for SRAM buffers. Second, we move portions of the NN computations close to the DRAM banks to decrease bandwidth pressure and increase performance and energy efficiency. Third, we show that despite the use of small SRAM buffers, the presence of 3D memory simplifies dataflow scheduling for NN computations. We present an analytical scheduling scheme that matches the efficiency of schedules derived through exhaustive search. Finally, we develop a hybrid partitioning scheme that parallelizes the NN computations over multiple accelerators. Overall, we show that TETRIS improves the performance by 4.1x and reduces the energy by 1.5x over NN accelerators with conventional, low-power DRAM memory systems.

CCS Concepts **• Computer systems organization → Neural networks**

Keywords 3D memory, neural networks, acceleration, dataflow scheduling, partitioning

1. Introduction

Modern computer intelligence problems, including voice recognition, object detection, and scene labeling, can be solved with unprecedented accuracy using deep *neural networks* (NNs) [27]. However, their high computation requirements (500K operations per pixel) and large memory footprints (up to 100s of MBytes) make NNs a challenging workload for programmable computer systems. Hence, recent research has focused on building domain-specific accelerators for NN computations, which mostly use spatial arrays of narrow-datawidth processing elements and achieve 100x improvements in performance and energy efficiency compared to general-purpose platforms [5, 7, 12, 18, 36].

Ideally, we would like to continue scaling the performance and efficiency of NN accelerators in order to achieve real-time performance for increasingly complicated problems. In general, the size of state-of-the-art NNs has been increasing over the years, in terms of both the number of layers and the size of each layer [19, 26, 41]. As a result, the memory system is quickly becoming the bottleneck for NN accelerators. To feed larger numbers of processing elements, the accelerators need to use large on-chip SRAM buffers and multiple DDRx channels, both of which are expensive in terms of power and cost (chip area or pin count).

Advances in through-silicon-via (TSV) technology have enabled 3D memory that includes a few DRAM dies on top of a logic chip [20, 22, 44]. Compared with the conventional 2D DRAM, 3D memory provides an order of magnitude higher bandwidth with up to 5x better energy efficiency by replacing the off-chip traces with hundreds of vertical connections [21]. Hence, 3D memory is an excellent option for the high throughput, low energy requirements of scalable NN accelerators. There are already proposals that place several NN accelerator arrays on the logic layer of a 3D memory stack to improve performance [25].

This paper focuses on improving the performance scalability and energy efficiency of inference tasks on large-scale NNs by going beyond naively combining 3D memory with NN accelerator arrays. At the hardware level, we take advantage of the high bandwidth and low access energy of 3D memory to rebalance the NN accelerator chip, using

ASPLOS '17, April 08 - 12, 2017, Xi'an, China

ISBN 978-1-4503-4465-4/17/04…$15.00
DOI: http://dx.doi.org/10.1145/3037697.3037702

more area for processing elements and less area for on-chip SRAM buffers. This allows us to achieve both higher performance *and* better energy efficiency. We also move simple accumulation operations close to the data locations (DRAM banks) in order to reduce memory accesses and improve performance and energy. At the software level, we build an analytical model to quickly derive optimized dataflow schedules, while previous NN designs required exhaustive search for the optimal schedules [7, 45]. Finally, we also develop an efficient hybrid partitioning scheme for NNs that allows us to exploit the high parallelism inherent in 3D memory stacks.

We combine these hardware and software optimizations in the design of TETRIS, an NN accelerator that uses an eight-die HMC memory stack organized into 16 vaults (vertical channels). Each vault is associated with an array of 14×14 NN processing elements and a small SRAM buffer. We demonstrate that a single TETRIS vault improves the performance and energy efficiency by 37% and 40% respectively over a larger, 16×16 array with a 4 times larger buffer and one 2D low-power DDRx channel. Using all 16 vaults in the 3D stack, TETRIS achieves 4.1x and 1.5x performance and energy improvements respectively over the conventional 2D design, even if we scale its memory channels to four. We show that TETRIS improves computational density by optimally using area for processing elements and on-chip buffers, and that moving partial computations to DRAM dies is beneficial. Moreover, our analytically derived dataflow schedules are equally efficient to the exhaustively searched optimal schedules using previous approaches. Finally, the proposed hybrid partitioning scheme improves the performance and energy efficiency by more than 10% over simple heuristics as we parallelize NN computations across multiple stacks. Overall, we develop and evaluate all of the elements necessary to make 3D memory a practical substrate for scalable and efficient NN accelerators.

The rest of this paper is organized as follows: Section 2 provides the background on NN acceleration and demonstrates the memory challenges for scaling NN accelerators. Section 3 presents the hardware architecture for TETRIS and Section 4 develops the software scheduling and partitioning schemes for it. Section 6 evaluates TETRIS against state-of-the-art 2D and previous 3D NN accelerator designs using the methodology in Section 5. Section 7 discusses the related work and Section 8 concludes the paper.

2. Background and Motivation

2.1 Deep Neural Networks

Advances in deep learning have made multi-layer neural networks (NNs) a popular approach for solving machine intelligence problems, as they have greatly exceeded the accuracy of traditional methods for many tasks such as recognition, localization, and detection [27]. An NN is composed of a pipeline or a directed acyclic graph (DAG) of layers. In the inference task, each layer performs computations on a set of neurons and feeds the results forward to later layers according to the network connections. The first layer accepts the input data (e.g., an image), and the last layer generates the output as an encoded tag or a probability vector for the category of the input (e.g., picture of a dog).

NNs can have different types of layers. In the computer vision domain, convolutional neural networks (CNNs) are mostly composed of convolutional (CONV) layers. The neurons in a CONV layer are organized into a set of 2D feature maps (fmaps). A small number of CONV layers at the front of the CNN work on large fmaps, e.g. 112×112 or 56×56, followed by a very deep stack of CONV layers processing a large set (up to 1024) of small (e.g., 14×14) fmaps per layer. Each CONV layer performs 2D convolutions between input fmaps (ifmaps) and trained filter weights, and then a non-linear activation function, such as ReLU or sigmoid, is applied on the sum of the convolution results to generate the output fmaps (ofmaps). This computation is typically performed on a batch of fmaps for better efficiency. The formal definition is:

$$\mathbf{O}[b][v] = f\left(\sum_{u=0}^{N_\mathrm{i}-1} \mathbf{I}[b][u] * \mathbf{W}[u][v] + \mathbf{B}[v]\right) \tag{1}$$
$$0 \le v \le N_\mathrm{o} - 1, \quad 0 \le b \le N_\mathrm{b} - 1$$

where $\mathbf{O}$, $\mathbf{I}$, and $\mathbf{W}$ are the ofmaps, ifmaps, and filter weights, each of which has four dimensions (2D image, number of fmaps, and batch). "$*$" denotes a 2D convolution operation, and N_i, N_o, and N_b are the number of ifmaps, ofmaps and the size of batch, respectively. $\mathbf{B}$ is a 1D bias.[1] f is the non-linear activation function.

Fully-connected (FC) layers also widely exist in many NNs (e.g., at the end of a CNN). An FC layer operates on lower-dimensional ifmaps and weights, and calculates their inner products to generate ofmaps. Mathematically, the computation can also be framed using Equation 1, by restricting each ofmap size to be 1×1. Pooling (POOL) and normalization (NORM) layers are also common, but do not use trained weights and are fast to process through streaming, thus we focus on CONV and FC layers.

Many studies have shown that NNs become more effective as the networks become deeper. For example, ResNet [19], the ILSVRC 2015 winner, has 152 layers. Furthermore, the composition of CONV layers in CNNs becomes modularized by using small filters (e.g., 3×3), but the number of fmaps in each layer is usually large (256 to 1024). Therefore, we expect future NNs will have increasing numbers of layers, and the CONV layers will have large numbers of fmaps. Both trends increase the NN size.

2.2 Neural Networks Acceleration

Given their high accuracy with challenging applications, there are now many industrial and academic efforts to accel-

[1] Mathematically $\mathbf{B}$ can be merged into $\mathbf{W}$, so we ignore it in this paper.

erate NNs of various scales, including designs based on custom ASICs, reconfigurable logic, multi-core systems, GPUs, and cluster-based solutions [1, 5, 15, 23, 47]. We focus on the domain-specific accelerators for NN inference. Such accelerators are likely to be deployed widely in both server and client (mobile or IoT) systems. NN training is usually done offline using clusters of GPUs [1].

Recent NN accelerators are typically spatial architectures with a large number of processing elements (PEs) for multiply-accumulate (MAC) operations [10, 30, 42]. The PEs are organized in a 2D array, with a custom on-chip interconnect between the PEs and the memory hierarchy [5, 7, 12]. Such spatial architectures are particularly suitable for NNs, as the 2D layout can capture the locality and communication in the tensor-based patterns (Equation 1). As a result, they can offer 100x higher performance over general-purpose processors for inference tasks, while only consuming 200 to 300 mW of power [5, 8].

2.3 Memory Challenges for NN Acceleration

The memory system, including the on-chip buffers and off-chip main memory, is frequently the bottleneck for NN accelerators. A typical CONV layer in start-of-the-art NNs may contain several hundreds of fmaps ($N_\mathrm{i}, N_\mathrm{o} = 100 - 500$), with each fmap being hundreds to thousands of pixels. Even with a low-precision 16-bit datawidth, the total size of the ifmaps, ofmaps, and filter weights can easily exceed megabytes. For example, the largest CONV layer in VGGNet [41] has 6.4 MB ofmaps and the largest FC layer has more than 102 million filter weights. The large volume of data makes it challenging to supply a large number of PEs efficiently. Recent research has focused on alleviating these memory bottlenecks. Large on-chip SRAM caches or buffers of up to a few MBytes are commonly used to capture the locality in the fmap and filter weight accesses in order to reduce the main memory traffic [8, 12]. Prefetching is also effective for NN workloads as the access patterns are statically known [25]. On the software side, better dataflow scheduling [7] and effective loop blocking and reordering techniques [45] have been proposed. They typically use heuristics or exhaustive search to find the schedules that minimize the off-chip accesses and maximize on-chip reuse in the NN computations.

While there is strong interest to scale the NN accelerators to achieve real-time performance on deeper and larger NNs, large PE arrays exacerbate the memory challenges, rendering the optimizations discussed above insufficient. First, the on-chip buffers, which are already at the MByte level [7], consume significant chip *area* (up to 70%), greatly increasing the cost and decreasing the computational density. Increasing the PE count would require even larger buffers in future NN accelerators. Second, even with an optimized on-chip buffer design, the required DRAM bandwidth can still reach 20 to 30 GBps for large PE arrays (see Figure 1), requiring several standard DDR3 channels that consume high *dynamic power*. Last but not least, there will be significant *static power consumption* from both the multiple off-chip DRAM channels and the large on-chip SRAM buffers. Even if we use low-power SRAM and DRAM technologies, the static power is still non-negligible (see Figure 1). Overall, relying solely on large on-chip buffers is not only an expensive approach in terms of both area and power, but is also insufficient to reduce the high bandwidth requirement and energy consumption of DRAM main memory.

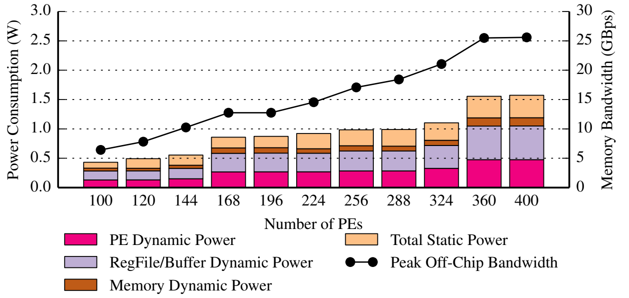


Figure 1. Memory bandwidth and power consumption when scaling the PE array size running VGGNet [41] on Eyeriss with the optimal scheduling. Logic runs at 500 MHz. 70% of on-chip area is used for buffers. The minimum number of LPDDR3-1600 channels are used in each case to supply the required bandwidth. The jumps in the scaling are due to mismatches between the dimensions of PE array and fmaps.

To demonstrate the memory bottlenecks of NN scaling, we take the state-of-the-art Eyeriss accelerator, currently using 168 PEs [8], and scale the PE array size. As shown in Figure 1, SRAM registers/buffers and DRAM contribute more than 60% of the total power. While the optimized scheduling minimizes the dynamic DRAM power, the static memory power becomes significant with larger arrays, even when low-power DRAM is used. When the PE array size scales to 400, the off-chip bandwidth exceeds 25 GBps despite using a 1.5 MB on-chip buffer. This requires four LPDDR3-1600 x32 chips and leads to overall system power of 1.5 W, six times higher than the current Eyeriss chip. Using DDR3/4 channels would result in even worse power consumption. Since we constantly fetch data from DRAM, it cannot go into the low power mode.

Recognizing the memory bottlenecks, several prior NN designs proposed to fully eliminate DRAM [12, 18]. However, we believe that a system with high bandwidth DRAM would be more flexible and general to use, since it would impose no limitation on the NN size. Future NNs will likely become even deeper, with increasing numbers of fmaps in each layer, making it harder to fit the entire model and intermediate data of all layers in on-chip SRAM. Switching between layers in an NN or between several NNs sharing the same hardware (e.g., image and voice recognition in parallel) makes DRAM unavoidable for practical NN accelerators that can support many large models.

2.4 Opportunities and Challenges with 3D Memory

3D memory [44] is a promising technology for the memory system of NN accelerators. It vertically integrates multiple DRAM dies on top of a logic layer within a single package by leveraging low-capacitance through-silicon vias (TSVs). The two well-known realizations of 3D memory are Micron's Hybrid Memory Cube (HMC) [20, 21] and JEDEC High Bandwidth Memory (HBM) [22, 28]. Nowadays, each 3D memory stack can use thousands of TSVs and provide an order of magnitude higher bandwidth (160 to 250 GBps) with 3 to 5 times lower access energy than DDR3 [21]. In addition, the large number of TSVs can be organized into multiple, independently-operated channels that exploit memory-level parallelism.

The Neurocube design has demonstrated the feasibility and performance benefits of using HMC for NN accelerators [25]. Nevertheless, to fully exploit the benefits of 3D memory, there are several challenges to address. First, given the characteristics of 3D memory, it is worth revisiting the on-chip buffer design. The lower cost of main memory access allows for smaller on-chip buffers with different use. Second, 3D integration technology also provides opportunities to rethink where computations are executed, potentially moving some operations closer to the actual memory locations. Third, once the memory and compute hardware changes, we need new approaches for dataflow scheduling for NN computations. Finally, the 3D memory stack with multiple vertical channels naturally creates a highly parallel system that requires efficient partitioning of NN computations. Neurocube fetched all ifmap data from 3D memory through specialized controllers without capturing reuse in the on-chip buffers, and only used a simple partitioning scheme [25]. Neither approaches was optimal.

3. TETRIS Architecture

TETRIS is an NN accelerator optimized for use with state-of-the-art 3D memory stacks. Its architecture also exploits the opportunities for near-data processing to move parts of the NN computations into the memory system [4].

3.1 Baseline System Architecture

We use Micron's Hybrid Memory Cube (HMC) [20, 21] as the 3D memory substrate of TETRIS.[2] Figure 2 shows the hardware architecture of TETRIS. The HMC stack (Figure 2 left) is vertically divided into sixteen 32-bit-wide *vaults* [21], which are similar to conventional DDRx channels and can be accessed independently. The vault channel bus uses TSVs to connect all DRAM dies to the base logic die. Each DRAM die contains two *banks* per vault (Figure 2 right top). Each bank is an array of DRAM cells. On data access, the global datalines transfer data from the internal DRAM cell arrays to the global sense-amplifiers (SAs) at the bottom of the bank, which amplify it and relay it to the channel TSV data bus. While accesses to different banks can overlap, all banks in a vault share the same TSVs.

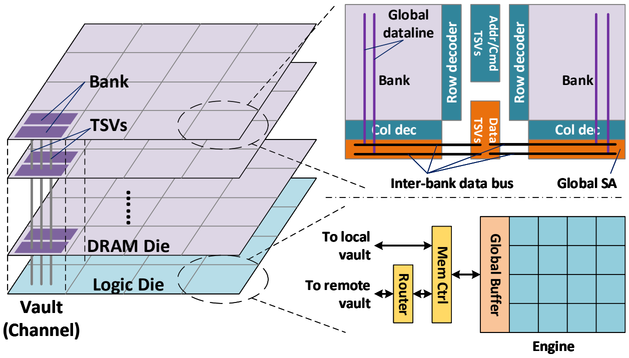


Figure 2. TETRIS architecture. Left: HMC stack. Right top: per-vault DRAM die structure. Right bottom: per-vault logic die structure.

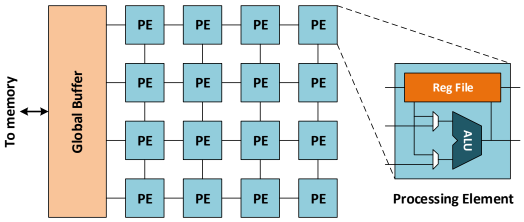


Figure 3. NN engine associated with one vault in TETRIS.

The original HMC logic die implements the vault memory controllers, a crossbar network connecting all vaults, the off-chip link SerDes, and other testing circuits [21]. Similar to previous studies [16, 25], we incorporate the NN accelerators by replacing the crossbar network with a 2D mesh network-on-chip (NoC). An NN engine is placed in each vault, connected to the vault memory controller as shown in Figure 2 (bottom right). The router can forward non-local accesses to other vaults through the NoC. Figure 3 shows the detailed structure of the NN engine associated with each vault. The engine is similar to a single Eyeriss accelerator [7]: Hundreds of PEs are connected through a dedicated network into a 2D array. Each PE contains a 16-bit fixed-point ALU and a small local register file of 512 to 1024 bytes. A global buffer is shared by all PEs to store and reuse data from memory. While the size of each engine is similar to one 2D NN accelerator, the collective number of PEs in all vaults in a stack is much higher. Multiple vault engines can be used to process a single NN layer in parallel (see Section 4.2).

[2] We prefer HMC to HBM because HMC connects to the host processor using packet-based serial links, which is convenient to program and control.

Operation	Energy (pJ/bit)	Relative Cost
16-bit PE [12]	0.2	1
256 kB SRAM [29]	1.2	6
2D DRAM random	15.0	75
2D DRAM sequential	4.6	23
3D DRAM random	5.1	26
3D DRAM sequential	4.2	21

Table 1. Energy comparison between PE, SRAM buffers, 2D and 3D DRAM. PE and SRAM buffers use 45 nm process. 2D DRAM is based on 16 Gb LPDDR3 [32]. 3D DRAM is based on the methodology in Section 5. We only consider internal access without off-chip transfer for 3D DRAM.

3.2 Memory Hierarchy for TETRIS

Recent NN accelerators assumed that the energy cost of DRAM access is one or two orders of magnitude higher than the access to on-chip SRAM [7, 12, 18]. The difference is due to the energy overheads of high-speed data transfers on PCB traces (on-die termination, PLL/DLL logic) [31] and the coarse granularity of DDRx access [43]. Regardless of the width of data needed, a whole DRAM row of 1 to 2 kB must be *activated* in the sense-amplifiers and 64 bytes are transfered on the DDRx channel. Hence, NN accelerators use large, on-chip SRAM buffers/registers to reduce DRAM accesses. However, these buffers take most of the chip area, 87% in ShiDianNao [12] and 70% in Eyeriss [7], despite the relatively small PE arrays, 8 × 8 and 16 × 16, respectively. They also consume significant power (see Figure 1).

3D memory has much lower energy consumption compared to 2D commodity DRAM. The vertical TSVs eliminate off-chip transfers for the NN engines on the logic die. Table 1 compares 2D DRAM and 3D memory internal access energy with a 256 kB SRAM buffer and the PE used in our study. While random access to 2D DRAM is 12 times more expensive than SRAM, 3D memory significantly reduces this cost. Moreover, the sequential patterns for accessing the tensor-based data in NN computations amortize the overheads of coarse-grain DRAM accesses. These two observations lower the necessity of large SRAM buffers.

Another reason to rebalance the resources used for SRAM buffers and PE arrays in 3D NN engines is the limited area of the logic die in 3D memory. With 10 to 16 GBps per vault [20], each engine needs more than 200 PEs to fully utilize this bandwidth (see Figure 1). The HMC logic die allows for roughly $50 - 60\ \mathrm{mm}^2$ of additional logic beyond the existing circuits [21], or $3.5\ \mathrm{mm}^2$ per vault. This is an extremely tight budget for a large PE array. For comparison, the Eyeriss NN accelerator took $12\ \mathrm{mm}^2$ for a 168 PE array with 182 kB of SRAM at 65 nm [8].

Overall, we argue that NN accelerators with 3D memory should use *larger* PE arrays (higher performance) with *smaller* SRAM buffers (lower area). Figure 4 shows the execution time and energy consumption with different PE array sizes in one vault. Since the area budget is fixed, to add more PEs we have to lower the global buffer size. We use the optimal dataflow schedules to achieve the best data reuse in the global buffer (see Section 5). Initially, the performance scales with the number of PEs until the reduced buffer size becomes the bottleneck (area-constrained design). While the energy spent on DRAM accesses slowly increases with smaller buffers, most of the dynamic energy is consumed by the NN engine. Moreover, the static energy decreases significantly with the improved performance. The most efficient design has 196 PEs with 133 kB buffer per vault. The buffer/PE area ratio is about 1:1, roughly half that of 2D Eyeriss that uses 70% of its area for buffers [7].

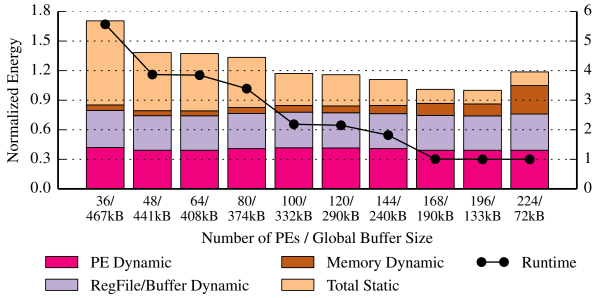


Figure 4. Energy consumption and execution time with different sizes of PE arrays and global buffers in a 3D NN engine running VGGNet with optimal scheduling. Register file per PE is 512 B. Total area budget is $3.5\ \mathrm{mm}^2$.

Note that we do not reduce the register file size in the PEs (512 bytes) as it can capture most of the data reuse given the efficient row stationary dataflow schedule [7]. With 196 PEs, this means that the SRAM buffer is only 1.35x larger than the aggregated register file capacity. Hence, the global buffer is not large enough to capture much additional data reuse beyond the register files, and it makes sense to bypass it for some types of data. Section 4 presents dataflow scheduling that realizes buffer bypass.

3.3 In-Memory Accumulation

The reduced global buffer capacity makes it important to minimize DRAM accesses. We focus on the accesses for ofmaps. The reduced buffer capacity may force the ofmaps to be swapped out to DRAM before being fully accumulated through all of the ifmaps. Hence, they must be fetched again to complete the accumulation, generating twice the traffic compared with the read-only ifmaps and weights. Since the succeeding convolution or inner-product computations do not depend on the partially accumulated ofmaps, we can treat any newly-generated values from the remaining ifmaps as *updates* and push them towards the partially accumulated ofmaps, rather than fetching the ofmaps again.

3D memory enables the implementation of such *in-memory accumulation*. The vault memory controller is

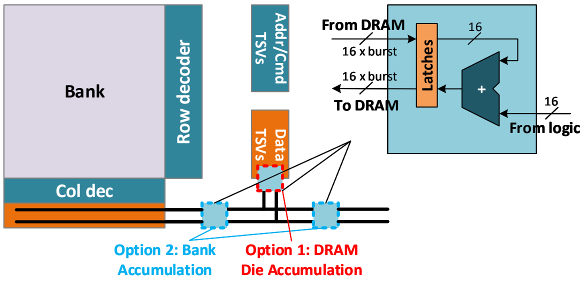


Figure 5. Two alternatives for in-memory accumulation: DRAM die accumulation and bank accumulation.

within the stack and can be fully customized. Unlike a DDRx channel that spreads values across multiple DRAM chips, a 3D vault sends all bits to the same DRAM bank, allowing the accumulation to be applied locally without carry propagation between chips or banks.

In-memory accumulation provides several benefits. First, it eliminates half of the ofmap memory traffic, saving memory bandwidth and thus improving performance. Second, the reduced ofmap data transfers also save energy on the vertical TSV channels. Third, compared with separate read and write accesses, the combined update access executes the two operations back-to-back in DRAM, resulting in better row buffer utilization and reduced peripheral logic activity (e.g., sharing row and column address decoding), which reduce both latency and energy consumption.

The key question becomes where we should place the accumulation logic. We discuss the tradeoffs of four options:

Memory controller accumulation: We can easily place the accumulators in the memory controllers on the logic die, without modifying the DRAM chips. However, this option does not eliminate any DRAM accesses and may introduce latency overheads by increasing the controller occupancy. Hence, we do not evaluate it.

DRAM die accumulation: We can place the accumulation logic close to the TSV driver block on the DRAM dies as shown in Figure 5. Assuming 32-bit vault channels with 8x burst, we need two 16-bit adders operating in a SIMD manner and two 128-bit latches that buffer the full size of data in a burst. We introduce an "update" command that works as follows. The address is first sent to the bank and the old values are read out to the latches. Next, the update values are supplied through the data bus to the accumulation logic which performs the additions. Finally, we switch the bank into write mode and write the updated values back to the bank. The data transfer between the bank and the TSV block and the bank mode switch between read and write may introduce slight latency overheads compared with normal read/write operations.

The accumulation logic is outside of the DRAM banks, so it does not affect the memory array layout. Since the logic is implemented in DRAM technology, its latency will be longer, but still small enough (sub-ns) compared to the DRAM array access latency. In addition, there is sufficient memory-level parallelism to hide its latency. We estimate the area for one accumulator (one 16-bit adder and one 128-bit latch) to be 663 μm^2 at 45 nm, indicating only 0.038% area overhead for two such accumulators per vault. Even if we consider the lower area efficiency of implementing logic in DRAM technology, the cost is negligible.

Bank accumulation: We can go one step further and place the accumulators in each DRAM bank. The two banks per die per vault can now perform updates in parallel without blocking the TSV bus for the duration of the update operation. To gather the data bits spread across the wide global sense-amplifier stripe, we reuse the inter-bank data bus and place the accumulators close to the TSV block, as shown in Figure 5. Since the accumulators are still at the bank periphery, they do not impact the cell arrays. The area overhead is 2x higher than DRAM die accumulation due to the replication, but is still negligible.

Subarray accumulation: Recent research has proposed to leverage the shared bitlines in each DRAM cell subarray within a bank to implement operations such as copy [38] and bitwise AND/OR [39]. While this approach can eliminate the need to read data out of a bank, it is not practical for accumulation as it applies operations at the granularity of full DRAM rows (1 to 2 kB) and the adder cells would introduce significant overheads and layout changes. Hence, we do not evaluate this approach.

In Section 6, we evaluate the performance and energy implications of DRAM die accumulation and bank accumulation. The RowClone proposal [38] considered the system-level implications, including issues such as address alignment, coherence (flush and invalidation) and memory consistency. These issues are not critical for our design as we use a custom NN accelerator that operates directly on physical memory. The accelerator uses scratchpad buffers, not coherent caches.

4. TETRIS Scheduling and Partitioning

The efficiency of an NN accelerator depends heavily on the scheduling (ordering) of the computations. Scheduling is particularly important for TETRIS since it uses small on-chip buffers, which could potentially increase the accesses to DRAM. Moreover, since TETRIS includes one NN accelerator per vault, we need to effectively partition and parallelize the NN workloads across vaults.

4.1 Dataflow Scheduling

Although the dataflow of NN computations for inference is well-defined, the large volume of data makes it difficult to simultaneously buffer the ifmaps, ofmaps, and filter weights in the multi-level memory hierarchy and optimize for their access order. Recall that each NN layer has N_i ifmaps. Each ifmap contributes to N_o ofmaps through convolutions or inner-products with different 2D filters. Such computations repeat for N_b times per batch.

The scheduling problem can be decomposed into two subproblems. First, we decide how to best map one 2D convolution computation of a particular group of ifmap $\mathbf{I}[b][u]$, filter $\mathbf{W}[u][v]$, and ofmap $\mathbf{O}[b][v]$ onto the PE array (*mapping* problem). Next, we decide the order between the $N_\mathrm{i} \times N_\mathrm{o} \times N_\mathrm{b}$ 2D convolutions and how to buffer the data in order to maximize on-chip reuse (*ordering* problem). Note that these two problems are orthogonal and can be solved separately. The mapping problem exploits the PE array interconnect and register files, while the ordering problem focuses on data buffering in the global buffer.

For the mapping problem, we leverage the state-of-the-art *row stationary* dataflow proposed for the Eyeriss design [7]. Row stationary maps the 1D convolution primitive onto a single PE to utilize the PE register file for local data reuse. It also carefully orchestrates the 2D convolution dataflow on the 2D array interconnect so that the data propagation between PEs remains local. Row stationary also supports mapping layers with different sizes onto the fixed-size PE array through folding (breaking large fmaps) and replication (processing multiple fmaps in parallel).[3]

While the row stationary dataflow provides an efficient solution to the mapping problem, the ordering problem remains unsolved. This problem involves blocking and reordering for the three loops (N_i, N_o, N_b) to reduce memory traffic. Previous work has used a cost model to capture the multi-level memory hierarchy (registers, global buffer, and main memory), and framed it as an optimization problem [7, 45]. However, the optimization problem is non-convex, and no general, closed-form solution exists. While exhaustive search can be used, searching large design spaces requires hours to days. Moreover, the optimal solution varies significantly with different layers [45], making it difficult to quickly deploy new NNs with many layers onto accelerators.

While the ordering problem is difficult in its general form, it is actually simpler to analyze with the memory hierarchy of TETRIS. As discussed in Section 3.2, TETRIS uses a larger PE array and a smaller global buffer. The buffer capacity is just slightly higher than that of all the register files in the PE array. Hence, for any register-allocated data, it is not profitable to also store copies in the global buffer since it is not large enough to capture further reuse patterns. Moreover, sequential accesses to 3D DRAM can be served at high bandwidth and are not particularly expensive in terms of energy (Table 1). Therefore, we propose to bypass the global buffer in TETRIS for two of the three input streams in the convolution computations, and utilize its full capacity to store data for just one stream in order to maximize the reuse of this stream in the global buffer beyond the register files. We call this approach *bypass ordering* and explore three variants: IW bypass (avoid the global buffer for ifmaps and filters), OW bypass (avoid the global buffer for ofmaps and filters), and IO bypass (avoid the global buffer for ifmaps and ofmaps).

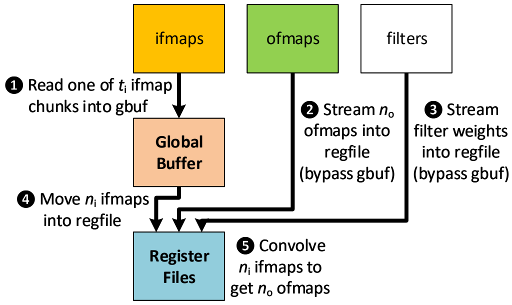

ifmaps
ofmaps
filters
❶ Read one of t_i ifmap chunks into gbuf
Global Buffer
❷ Stream n_o ofmaps into regfile (bypass gbuf)
❸ Stream filter weights into regfile (bypass gbuf)
❹ Move n_i ifmaps into regfile
Register Files
❺ Convolve n_i ifmaps to get n_o ofmaps

Figure 6. The diagram of OW bypass ordering with batching omitted. See Algorithm 1 for a detailed description.

Figure 6 explains OW bypass as an example. We split the N_i ifmaps into multiple chunks and fill the global buffer with one chunk at a time (❶). For each ifmap chunk, the ofmaps are streamed directly from DRAM into the register files (❷). Since each ifmap-ofmap pair uses a different 2D filter, the filters have little reuse outside of batching and are also streamed directly from DRAM (❸). The register files are used exactly the same way as row stationary dataflow dictates: the ifmaps in the global buffer will be streamed in (❹) to process with the ofmaps and filters stored in the register files (❺). Recall that the numbers of ifmaps, ofmaps, and the batch size are N_i, N_o, N_b, respectively. Assuming the blocking factors on the ifmaps and batch are t_i and t_b, the overall OW bypass ordering algorithm is formally presented in Algorithm 1, including the batching details. The IW bypass and IO bypass algorithms are similar; they use the global buffer to store the ofmaps and filters respectively.

Bypass ordering is significantly simpler than the general loop blocking and reordering schemes explored in [45] and can be implemented using a simple hardware controller (FSM). Moreover, we can analytically derive the optimal scheduling parameters without exhaustive search. With the parameters defined above and the schedule shown in Algorithm 1, the total number of accesses to DRAM is

$$A_\mathrm{DRAM} = 2 \times N_\mathrm{b} N_\mathrm{o} S_\mathrm{o} \times t_\mathrm{i} + N_\mathrm{b} N_\mathrm{i} S_\mathrm{i} + N_\mathrm{o} N_\mathrm{i} S_\mathrm{w} \times t_\mathrm{b} \quad (2)$$

where S_i, S_o, S_w are the sizes of one 2D ifmap, ofmap, and filter, respectively. With OW bypass, each ifmap is only accessed once. The ofmaps are read and written t_i times and the filters are streamed t_b times. The constraints are further summarized as

$$\begin{cases} \frac{N_\mathrm{b}}{t_\mathrm{b}} \times \frac{N_\mathrm{i}}{t_\mathrm{i}} \times S_\mathrm{i} \leq S_\mathrm{buf} \\ 1 \leq t_\mathrm{b} \leq N_\mathrm{b}, \quad 1 \leq t_\mathrm{i} \leq N_\mathrm{i} \end{cases} \quad (3)$$

[3] Folding and replication during mapping will affect the numbers of ifmaps, ofmaps, and batch size in the ordering problem. For example, if we replicate 2 ofmaps on the same physical PE array, N_o will reduce by half. We take this into account in the models.

```
for $b_1 \leftarrow 1, t_\mathrm{b}$ do
    // divide batch $N_\mathrm{b}$ into $t_\mathrm{b}$ pieces, each with $n_\mathrm{b}$
       fitting in registers.
    for $i_1 \leftarrow 1, t_\mathrm{i}$ do
        // $t_\mathrm{i}$ ifmap chunks, each time read one chunk
           into global buffer.
        for $o_2 \leftarrow 1, \frac{N_\mathrm{o}}{n_\mathrm{o}}$ do
            // stream $n_\mathrm{o}$ ofmaps to registers each
               time.
            for $i_2 \leftarrow 1, \frac{N_\mathrm{i}}{t_\mathrm{i} n_\mathrm{i}}$ do
                // move $n_\mathrm{i}$ buffered ifmaps into
                   registers each time.
                // now $n_\mathrm{o}$ ofmaps and $n_\mathrm{i}$ ifmaps in
                   registers, each with batch $n_\mathrm{b}$.
                for $o_3 \leftarrow 1, n_\mathrm{o}$ do
                    for $i_3 \leftarrow 1, n_\mathrm{i}$ do
                        for $b_3 \leftarrow 1, n_\mathrm{b}$ do
                            // do computation with all
                               data in registers using
                               row stationary.
                        end
                    end
                end
            end
        end
    end
end
```

Algorithm 1: Pseudocode for OW bypass ordering. Loop subscripts 1, 2, and 3 describe blocking at the level of the global buffer, register file, and PEs, respectively.

given that each time the global buffer stores $N_\mathrm{i}/t_\mathrm{i}$ ifmaps with $1/t_\mathrm{b}$ of the batch size N_b.

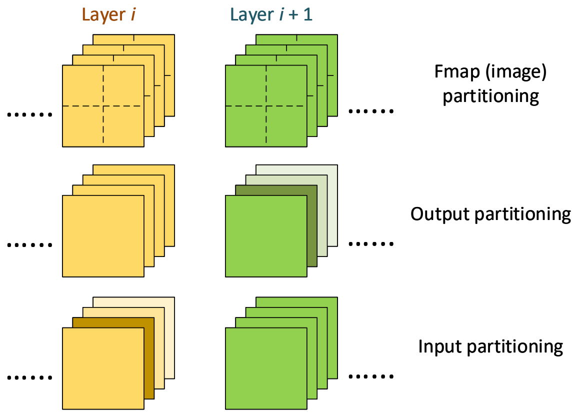
Unlike the general ordering problem [45], minimizing Equation 2 for TETRIS under the constraints of Equation 3 can be formalized as a convex optimization to solve for t_i and t_b. Furthermore, we can obtain an analytical solution for OW bypass:

$$t_\mathrm{i} = N_\mathrm{i}\sqrt{\frac{S_\mathrm{w} S_\mathrm{i}}{2 S_\mathrm{o} S_\mathrm{buf}}}, \quad t_\mathrm{b} = N_\mathrm{b}\sqrt{\frac{2 S_\mathrm{i} S_\mathrm{o}}{S_\mathrm{w} S_\mathrm{buf}}} \tag{4}$$

There are similar solutions for IW and IO bypass ordering.

These solutions allow us to optimally schedule any NN layer on TETRIS. The optimal values for t_i and t_b in Equation 4 also provide interesting insights. When the fmap size is large and the filter size is small ($S_\mathrm{o} \gg S_\mathrm{w}$), we need large t_b (smaller batch) but small t_i (larger fmap chunks). The cost of refetching the filters is relatively small compared to that of refetching the fmaps. Hence, we should reduce the number of times we stream the fmaps (i.e., t_i) at the cost of reducing the filter reuse across batches.

Bypass ordering also benefits from the in-memory accumulation discussed in Section 3.3. When the ofmaps bypass the global buffer, each ofmap is read and written t_i times. With in-memory accumulation, the updates to the ofmaps are pushed directly to the memory, which eliminates the need for ofmap reads and further reduces the cost of bypassing the global buffer. Accordingly, the factors of two in Equation 2 and Equation 4 will be removed.



Scheme	Ifmap	Ofmap	Weight
Batch	Fully partition	Fully partition	Fully replicate
Fmap	With overlap	Fully partition	Fully replicate
Output	Broadcast	Fully partition	Fully partition
Input	Fully partition	Broadcast	Fully partition

Figure 7. NN partitioning scheme comparison.

4.2 NN Partitioning across Vaults

The 3D memory stack in TETRIS contains multiple vaults, each with a separate NN accelerator array (see Figure 2). In addition to processing different NNs or layers in each vault, we can divide large NN layers across the vaults to process them in parallel. We first present a taxonomy of the different partitioning schemes summarized in Figure 7. Then, we systematically explore and find the optimized scheme.

Batch partitioning (data-parallelism): The simplest scheme is to use the multiple NN accelerators to process multiple input images in parallel, effectively dividing a batch of images across vaults [11]. While good for throughput, it requires the NN model to be replicated in each vault, which is a significant capacity challenge for large NNs. Moreover, parallelism is limited by the batch size. This scheme is less attractive for latency-sensitive, real-time applications since it does not improve the latency for inference on each image.

Fmap (image) partitioning: If the fmap is large (e.g., 112 × 112), we can partition it into smaller tiles as shown in Figure 7. The smaller fmap tile will fit better into the PE array in each vault and reduce the need for folding in the row stationary dataflow [7]. Moreover, If the ifmaps and ofmaps in a CONV layer use the same fmap partitioning, most data accesses will be local within each vault due to the locality of 2D convolution. However, the filters need to be replicated across all vaults.

Output partitioning: As each layer usually has multiple ofmaps, we can partition the ofmaps across vaults. For example, we can divide the N_o ofmaps into 16 groups and each vault processes $N_\mathrm{o}/16$ ofmaps. Since each ofmap uses different filters, the filter weights can be fully partitioned.

Since all ifmaps contribute to all ofmaps, all ifmaps must be sent to all vaults, which requires remote vault accesses.

Input partitioning: Similar to output partitioning, we could also partition the N_i ifmaps across vaults. The difference is where the computations take place. However, as discussed in Section 3.3, access to ofmaps generates both read and write traffic, and thus is more critical than the ifmap access. Hence, it is better to use output partitioning that avoids the remote accesses for ofmaps, rather than using input partitioning that avoids the remote accesses for ifmaps.

Neurocube [25] used a simple heuristic to minimize remote vault accesses, i.e., fmap partitioning for CONV layers and output partitioning for FC layers. This heuristic does not necessarily lead to the best performance and energy. In addition to the number of remote accesses, we should consider the impact of the partitioning scheme on the total number of memory accesses and the data reuse in the PE register files. Because the fmaps are tiled with fmap partitioning, each vault needs to load the same filters into the PE registers, while only reusing them across the smaller fmap tile. In contrast, output partitioning keeps the whole fmap in one vault, loads the filters once, and uses them across the whole fmap which results in higher filter reuse. Moreover, output partitioning can be combined with the OW bypass ordering in Section 4.1, which requires just one round of ifmap reads, minimizing the cost of ifmap remote accesses.

Therefore, we consider a *hybrid* partitioning scheme that strikes a balance between the benefits of fmap partitioning (minimizing remote accesses) and those of output partitioning (better on-chip data reuse to minimize total DRAM accesses). Such a hybrid scheme also captures the intention of the Neurocube heuristic: for the first few CONV layers in common CNNs, the access to large fmaps dominates and the weight reuse is not significant, thus we should mostly use fmap partitioning; for the FC layers, the filter weights are much larger than the fmaps, so output partitioning is preferred to maximize weight reuse.

To find the optimized partitioning scheme, we propose a simple cost model in terms of overall memory access energy:

$$E_\text{access} = A_\text{DRAM} \times e \times (1 + \beta r) \tag{5}$$

where e is the energy for one DRAM access to the local vault, β is the energy penalty for one remote vault access, and r is the percentage of accesses to remote vaults. A_DRAM is the total number of main memory accesses. Fmap partitioning minimizes r but results in larger A_DRAM, while output partitioning has smaller A_DRAM by sacrificing r.

Each NN layer can be potentially parallelized using a combination of fmap partitioning and output partitioning. Adjacent layers are coupled because the partitioning scheme used for the previous layer determines the layout of ifmaps of the next layer (see Figure 7). Assuming ℓ layers and c partitioning combinations per layer, we must consider the overall memory access energy cost for c^ℓ scenarios. As ℓ can be 20 to 100 and c can be 4 to 8, the cost is prohibitive.

	AlexNet [26]	ZFNet [46]	VGG16 [41]	VGG19 [41]	ResNet [19]
# CONVs	5	5	13	16	151
# FCs	3	3	3	3	1
Fmap size	0.58/1.32	2.4/3.4	6.4/27.0	6.4/28.0	1.6/40.0
Filter size	74/124	150/214	204/276	204/286	4.6/110
Year	2012	2013	2014	2014	2015

Table 2. The NNs used in our evaluation. We show the fmap and filter sizes in Mbytes for the largest layer and the whole NN (largest/total). We use 16-bit fixed-point data.

We leverage two strategies to reduce the difficulty of partitioning exploration. First, we use a greedy algorithm that explores the partitioning options for layer i without backtracing, assuming the partitioning of the first $i-1$ layers in the optimal scheme is independent of that for the later layers. The first CONV layer is assumed to only use fmap partitioning. This allows us to reduce the number of choices to $c \times \ell$, roughly several thousands in common cases. Second, we apply the bypass ordering in Section 4.1 to the partitioned layers in each vault. This allows us to determine A_DRAM analytically without time-consuming exhaustive search. Although partitioned layers are smaller, the solutions are still close to optimal as we show in Section 6.3.

4.3 NN Partitioning across Multiple Stacks

We can use the same methodology to partition inference tasks across multiple 3D stacks with NN accelerators. Multiple stacks allow us to scale to arbitrarily large NNs. For inter-stack partitioning, the remote access cost β in Equation 5 is significantly higher. Moreover, the remote access percentage r also increases with more stacks. This implies that we should mostly use fmap partitioning to minimize remote accesses for multi-stack scenarios.

We leave a thorough exploration of multi-stack partitioning to future work. Current 3D stacking technology supports 4 to 8 GB with up to 256 GBps per stack. Hence, a single stack is sufficient for inference with large NNs. Multi-stack configurations may be more interesting for training, where the memory footprint increases significantly [1].

5. Methodology

Workloads. We use several state-of-the-art NN workloads to evaluate TETRIS, summarized in Table 2. They are medium to large scale with several hundreds of MBytes memory footprints. In particular, we include the ILSVRC 2015 winner, ResNet, one of the largest NNs nowadays with 152 layers [19]. AlexNet, ZFNet, and ResNet (despite its large depth) mostly use small fmaps no more than 50 × 50, except for the first few layers. The two VGGNet variants contain large fmaps in the early layers, 56 × 56 to 224 × 224, which dominate performance. All NNs have large numbers of fmaps at the later layers (up to 2048)

System models and simulation. We use 45 nm technology for the logic of 2D baseline NN accelerators and the 3D NN engines. We model the NN compute engine after Eyeriss [7, 8] (Figure 3). Both the 2D and 3D engines run at 500 MHz. The area and power consumption of each PE are scaled from [8, 12], assuming $0.01\ \mathrm{mm}^2$ at 45 nm and 3.2 pJ for each 16-bit multiply-accumulate operation including control overheads. The area and power of the register files and the global buffers at different capacities are modeled using CACTI-P [29] that has improved leakage power models over previous versions. For the 3D case, the total area budget for additional logic of each vault is assumed to be $3.5\ \mathrm{mm}^2$, which constrains the PE array size and the global buffer capacity. In the 2D baselines we assume no limitation for the chip area, but the available DRAM bandwidth is limited by the pin count: at most 4 DDR3/LPDDR3 channels can be connected to one accelerator chip.

The 2D baselines use 16 Gb, LPDDR3-1600, x32 chips. Each channel provides 6.4 GBps bandwidth. The memory timing and power parameters are obtained from datasheets [32] and the system power is calculated using [31]. For 3D memory, we use an HMC stack [20, 21], assuming sixteen 32-bit-wide vaults in each stack, providing 8 GBps bandwidth per vault. The power parameters are modeled using [43] and calibrated with prior literature on HMC designs [17, 21]. For remote accesses to other vaults, the NoC power is estimated using ORION 2.0 [24].

Given the timing characteristics of the NN accelerators and the memory channels/vaults, we use zsim [37] to evaluate the performance and in particular the impact of the limited memory bandwidth. We extended zsim to model the logic die NoC interconnect latency and throughput. We generate memory address traces based on the schedule. We adjust the trace to incorporate prefetching, but perfect prefetching is impractical since it would require us to use half of the global buffer capacity for double buffering. Our methodology captures the latency of DRAM accesses. Once data is on-chip, all other accesses to the global buffer and register files and all PE operations proceed deterministically in fully predictable manners. We use the actual simulated time instead of the modeled ideal number of execution passes to compare performance and calculate static energy consumption.

Scheduling and partitioning. We implemented and released a scheduling tool[4] to schedule NN layers of arbitrary sizes onto the given-size physical PE array. The mapping phase uses row stationary dataflow with folding and replication to maximize register-file-level data reuse [7]. The ordering phase uses exhaustive search to find the best loop blocking and reordering [45]. The tool has been verified against the Eyeriss results [7]. We use its results as the globally optimal schedules and compare them against the results from the analytical solutions for bypass ordering (Section 4.1)

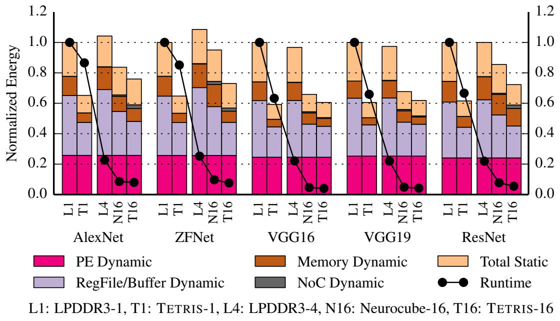


Figure 8. Performance and energy comparison between 2D and 3D NN architectures.

We also extended our scheduling tool to support NN partitioning (Section 4.2). It includes the base partitioning scheme from the Neurocube design [25], and the greedy algorithm for the hybrid partitioning scheme. Once layers are partitioned, we use bypass ordering for dataflow scheduling as explained above.

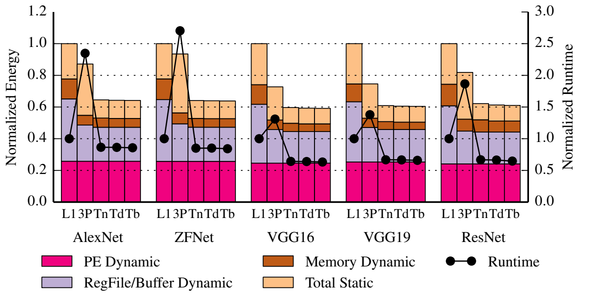
6. Evaluation

6.1 Overall Performance and Energy Comparison

Figure 8 compares the performance and energy consumption of TETRIS with three baselines. The 2D baselines use 1 and 4 LPDDR3 channels, respectively (L1 and L4). Their NN engine is a scaled-up version of Eyeriss [7], using a 16×16 PE array with a 1024-byte register file per PE and a 576 kB global buffer. There is one NN engine in the single-channel baseline. The 4-channel baseline has 4 NN engines for a total of 1024 PEs and 2.3 MB global buffer ($34\ \mathrm{mm}^2$). Both of the TETRIS designs, with 1 and 16 vaults, respectively (T1 and T16), use a 14×14 PE array per vault, with a 512-byte register file per PE and a 133 kB global buffer per vault (Figure 4). We assume bank in-memory accumulation and use the analytical scheduling solutions and hybrid partitioning schemes presented in Section 4 for both TETRIS designs. The 2D designs are limited by the off-chip memory bandwidth, while the 3D TETRIS designs are limited by the area in the logic die. We also compare to the 3D Neurocube design with 16 vaults (N16) [25]. For a fair comparison, we scale up its logic resources to be the same as in TETRIS, but use their proposed scheduling and partitioning schemes. As a result, any difference between TETRIS and Neurocube comes from software.

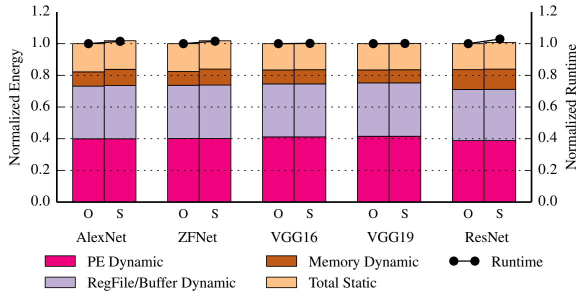
For the 2D NN accelerator, increasing the number of LPDDR3 channels and NN engines from 1 to 4 improves the performance by 3.9x to 4.6x (L1 vs. L4). However, the overall energy consumption remains roughly the same (97% to 108%), suggesting a 4-5x increase in power. In other words, the performance scaling also increases the cost (power consumption and pin count).

[4] `https://github.com/stanford-mast/nn_dataflow`.



L1: LPDDR3, 3P: 3D-Proj, Tn: TETRIS, Td: TETRIS w/ DieAccum, Tb: TETRIS w/ BankAccum

Figure 9. Effects of PE/buffer area rebalancing and in-memory accumulation in TETRIS. All results use 1 channel/vault.



O: Optimal, S: Solved

Figure 10. Comparison between optimal dataflow schedules and analytical solutions of bypass ordering. Both use 1 vault.

TETRIS with a single 3D vault (T1) leads to a 35% to 40% energy reduction over 1 LPDDR3 channel (L1). Its power consumption is actually lower despite improving the performance by up to 37%. Scaling to 16 vaults (T16) results in 12.9x better performance with 9.2% higher energy consumption on average over T1. Overall, the 16-vault TETRIS improves the performance by 4.1x and reduces the energy by 1.48x over the L4 baseline with 4 LPDDR3 channels. 3D memory provides the bandwidth necessary to scale the NN accelerator from 4 to 16 PE arrays in an energy-efficient manner. Compared with the scaled Neurocube design (N16), TETRIS (T16) improves the performance and energy by 20% and 15%, respectively, which establishes the need for better scheduling and partitioning schemes.

Note that in both 2D and 3D designs, parallelizing a layer across multiple PE arrays results in higher energy consumption on SRAM and DRAM memories. This validates the reduction in data reuse discussed in Section 4.2 and the need to co-optimize remote accesses and total memory accesses with the hybrid partitioning scheme. This co-optimization allows the 16-vault TETRIS design to have better energy characteristics over the 16-vault Neurocube.

The aggregated power consumption of the 16-vault TETRIS is on average 6.94 W at 500 MHz with a maximum of 8.42 W, which is comparable to the power consumption of previous studies that placed compute capabilities in HMC stacks [2, 16, 35]. Eckert et al. demonstrated that 10 W power consumption in 3D memory stacks is feasible even with low-end passive heat sinks [14] . Therefore, we believe that processing at full speed with all vaults in the stack is a practical choice to accelerate the NN workloads.

6.2 3D NN Hardware Evaluation

We now evaluate the impacts of rebalancing the on-chip PE/buffer area (Section 3.2) and in-memory accumulation (Section 3.3). Figure 9 shows the performance comparison and energy breakdown for five systems: the previous 1-channel LPDDR3 baseline (L1), a projected 3D design (3P), and three TETRIS designs with no in-memory accumulation (Tn), DRAM die accumulation (Td), and bank accumulation (Tb). The projected design uses the same PE-to-buffer area ratio as the 2D accelerator and simply scales down to a 10×11 array with 259 kB buffer to fit within the $3.5\ \mathrm{mm}^2$ area constraint.

Simply scaling down the 2D design to fit in the 3D constraints not only leads to a performance drop of up to 2.7x due to the smaller PE array, but also has limited (7% to 27%) energy savings from 3D memory due to the increased static energy. When the area budget is rebalanced (Tn), we are able to achieve up to 36% performance improvement over L1, taking advantage of the higher memory bandwidth in a 3D vault versus an LPDDR3 channel (8 GBps vs. 6.4 GBps). The energy savings are also higher, ranging from 35% to 40% compared to L1.

DRAM die accumulation (Td) reduces the DRAM dynamic energy by 6.3% over no in-memory accumulation (Tn) on average, and up to 10.4% with the deepest NN (ResNet), as it reduces the traffic on the vertical channel and activities in the DRAM peripheral logic (Section 3.3). The savings are lower for AlexNet and ZFNet, since their DRAM energy is mostly from operating on FC layers that in-memory accumulation does not help with. Bank accumulation (Tb) further improves the performance and static energy by up to 3.1% over Td. However, since we have already minimized DRAM accesses and applied effective prefetching, the overall performance and system energy improvements are limited.

6.3 Dataflow Scheduling and Partitioning

Figure 10 compares the analytically derived bypass ordering schedules in Section 4.1 to the optimal dataflow schedules obtained from exhaustive search. Unlike partitioning, ordering for each different layer is independent. Hence, we choose the best among IW, OW, and IO bypass schedules for each layer. There is hardly any difference between bypass and optimal schedules, less than 2.9% for runtime and 1.8% for energy. We have verified that for most layers in the NNs, the

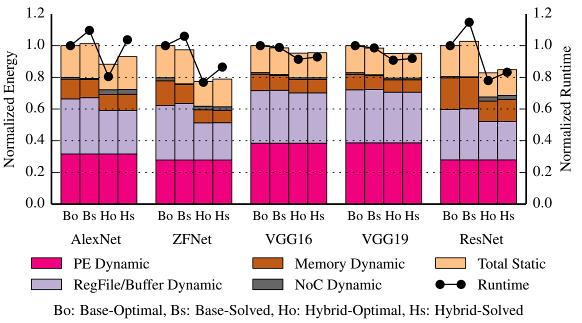


Figure 11. Comparison between the base partitioning used in [25] and the hybrid partitioning over 16 vaults in TETRIS.

two schedules are equivalent, demonstrating the effectiveness of our analytical approach.

Finally, Figure 11 compares the TETRIS hybrid partitioning scheme in Section 4.2 with the base partitioning scheme used in Neurocube [25], which exclusively uses fmap partitioning for CONV layers and output partitioning for FC layers. We present both results with exhaustive search and analytical solving as the scheduling approach, respectively. While the NoC energy increases slightly with hybrid partitioning, the SRAM and DRAM energy decreases due to better data reuse, leading to 13.3% performance and 10.5% energy improvements on average. Note that the analytical scheduling solutions are less optimized than the single-vault case (Figure 10), mostly due to the smaller fmap sizes after partitioning which make it easier to buffer data on-chip and less attractive to bypass.

7. Related Work

NN acceleration. There were several recent proposals for CNN accelerators using ASIC or FPGA platforms [33, 34]. Early designs [5, 6, 47] used parallel inner-product engines which could be easily programmed for different layer types. Neuflow [15], ShiDianNao [12], and Eyeriss [8] utilized 2D spatial PE arrays with specialized interconnects to match the 2D convolution pattern. Several techniques are commonly used to improve computational efficiency. Low-precision arithmetic reduces area and energy cost with minimum accuracy impact for NN inference [5, 36]. Dynamically pruning zero and/or small activations and weights at runtime avoids useless work [3, 36]. Static pruning compresses NNs into sparse formats to reduce memory footprints [18]. While our baseline compute engine is similar to Eyeriss, these techniques are orthogonal to the TETRIS memory hierarchy and can be integrated into our design.

NN and near-data processing. Most recent proposals for near-data processing (NDP) architectures focused on general-purpose processing [2, 17, 35]. HRL used a heterogeneous reconfigurable array that was similar to spatial NN engines [16]. ISAAC [40] and PRIME [9] utilized non-volatile ReRAM crossbars to not only store the weights but also perform in-situ analog dot-product operations. Neurocube proposed an NN accelerator for 3D memory stacks [25], with SIMD-style multiply-accumulate units in each vault to process the partitioned NNs. Each vault also had a specialized memory controller for prefetching. We extend Neurocube by using a 2D PE array with a global buffer as the compute engine in each vault, enabling us to explore more sophisticated dataflow scheduling. We also study in-memory accumulation and hybrid partitioning schemes. Finally, the RowClone design for in-memory bulk data copy [38], and its extension on bulk bitwise AND/OR operations [39] implemented simple operations inside DRAM arrays. Our in-memory accumulation leverages the same insights, but is less intrusive and allows for more powerful arithmetic operations.

NN scheduling and partitioning. NN scheduling was typically optimized using auto-tuning or heuristic-based loop blocking strategies [13, 34, 47]. Eyeriss summarized a taxonomy of CNN dataflows and proposed row stationary heuristic, but it still used exhaustive search for the ordering problem [7]. Yang et al. proposed a framework to optimize CNN scheduling in multi-level memory systems and pointed out the non-convexity of the problem. Thus, the optimal schedule could only be found by exhaustive search [45]. In contrast, bypass scheduling allows us to analytically derive near-optimal schedules that yield good performance in the restricted 3D memory environment. Yang et al. also explored optimal NN partitioning in their framework for uniformly distributed systems [45]. Neurocube used a simple heuristic for 2D partitioning [25]. Neither of them considered hybrid partitioning in one layer.

8. Conclusion

We presented TETRIS, a scalable and efficient architecture with 3D-stacked memory for neural network inference. TETRIS provides 4.1x performance improvement with 1.5x energy saving compared to an aggressive 2D NN accelerator design with 1024 processing elements and four, low-power DRAM channels. The performance and energy gains are achieved by rebalancing the use of area for processing elements and SRAM buffers, as well as introducing novel in-memory accumulation hardware features. We have also proposed a scheduling scheme for TETRIS that can be derived analytically and has equivalent efficiency to the optimal schedules derived from exhaustive search. Finally, we presented a hybrid partitioning scheme that parallelizes the NN layers across multiple vaults in the stack, further improving performance and efficiency.

Acknowledgments

The authors want to thank the anonymous reviewers for their insightful comments. This work was supported by the Stanford Pervasive Parallelism Lab, the Stanford Platform Lab, and NSF grant SHF-1408911.

References

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 265–283, 2016.

[2] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi. A Scalable Processing-in-memory Accelerator for Parallel Graph Processing. In *42nd International Symposium on Computer Architecture (ISCA)*, pages 105–117, 2015.

[3] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos. Cnvlutin: Ineffectual-Neuron-Free Deep Neural Network Computing. In *43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 1–13, 2016.

[4] R. Balasubramonian, J. Chang, T. Manning, J. H. Moreno, R. Murphy, R. Nair, and S. Swanson. Near-Data Processing: Insights from a MICRO-46 Workshop. *IEEE Micro*, 34(4): 36–42, 2014.

[5] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam. DianNao: A Small-Footprint High-Throughput Accelerator for Ubiquitous Machine-Learning. In *19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 269–284, 2014.

[6] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam. DaDianNao: A Machine-Learning Supercomputer. In *47th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO)*, pages 609–622, 2014.

[7] Y.-H. Chen, J. Emer, and V. Sze. Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks. In *43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 367–379, 2016.

[8] Y.-H. Chen, T. Krishna, J. Emer, and V. Sze. Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks. In *IEEE International Solid-State Circuits Conference (ISSCC)*, pages 262–263, 2016.

[9] P. Chi, S. Li, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie. PRIME: A Novel Processing-in-Memory Architecture for Neural Network Computation in ReRAM-Based Main Memory. In *43rd International Symposium on Computer Architecture (ISCA)*, pages 27–39, 2016.

[10] K. Choi. Coarse-Grained Reconfigurable Array: Architecture and Application Mapping. *IPSJ Transactions on System LSI Design Methodology*, 4:31–46, 2011.

[11] J. Dean, G. S. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, and A. Y. Ng. Large Scale Distributed Deep Networks. In *25th International Conference on Neural Information Processing Systems (NIPS)*, pages 1223–1231, 2012.

[12] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam. ShiDianNao: Shifting Vision Processing Closer to the Sensor. In *42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 92–104, 2015.

[13] A. Dundar, J. Jin, V. Gokhale, B. Martini, and E. Culurciello. Memory Access Optimized Routing Scheme for Deep Networks on a Mobile Coprocessor. In *2014 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6, 2014.

[14] Y. Eckert, N. Jayasena, and G. H. Loh. Thermal Feasibility of Die-Stacked Processing in Memory. In *2nd Workshop on Near-Data Processing (WoNDP)*, 2014.

[15] C. Farabet, B. Martini, B. Corda, P. Akselrod, E. Culurciello, and Y. LeCun. Neuflow: A Runtime Reconfigurable Dataflow Processor for Vision. In *2011 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, pages 109–116, 2011.

[16] M. Gao and C. Kozyrakis. HRL: Efficient and Flexible Reconfigurable Logic for Near-Data Processing. In *22nd IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 126–137, 2016.

[17] M. Gao, G. Ayers, and C. Kozyrakis. Practical Near-Data Processing for In-Memory Analytics Frameworks. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pages 113–124, 2015.

[18] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dall. EIE: Efficient Inference Engine on Compressed Deep Neural Network. In *43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 243–254, 2016.

[19] K. He, X. Zhang, S. Ren, and J. Sun. Deep Residual Learning for Image Recognition. *arXiv preprint arXiv:1512.03385*, 2015.

[20] Hybrid Memory Cube Consortium. Hybrid Memory Cube Specification 2.1, 2014.

[21] J. Jeddeloh and B. Keeth. Hybrid Memory Cube New DRAM Architecture Increases Density and Performance. In *2012 Symposium on VLSI Technology (VLSIT)*, pages 87–88, 2012.

[22] JEDEC Standard. High Bandwidth Memory (HBM) DRAM. JESD235A, 2015.

[23] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional Architecture for Fast Feature Embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[24] A. B. Kahng, B. Li, L.-S. Peh, and K. Samadi. ORION 2.0: A Fast and Accurate NoC Power and Area Model for Early-stage Design Space Exploration. In *Conference on Design, Automation and Test in Europe (DATE)*, pages 423–428, 2009.

[25] D. Kim, J. Kung, S. Chai, S. Yalamanchili, and S. Mukhopadhyay. Neurocube: A Programmable Digital Neuromorphic Architecture with High-Density 3D Memory. In *43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 380–392, 2016.

[26] A. Krizhevsky, I. Sutskever, and G. E. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In *25th International Conference on Neural Information Processing Systems (NIPS)*, pages 1097–1105, 2012.

[27] Y. LeCun, Y. Bengio, and G. Hinton. Deep Learning. *Nature*, 521(7553):436–444, 2015.

[28] D. U. Lee, K. W. Kim, K. W. Kim, H. Kim, J. Y. Kim, Y. J. Park, J. H. Kim, D. S. Kim, H. B. Park, J. W. Shin, J. H. Cho, K. H. Kwon, M. J. Kim, J. Lee, K. W. Park, B. Chung, and S. Hong. 25.2 A 1.2V 8Gb 8-channel 128GB/s High-Bandwidth Memory (HBM) Stacked DRAM with Effective Microbump I/O Test Methods Using 29nm Process and TSV. In *IEEE International Solid-State Circuits Conference (ISSCC)*, pages 432–433, 2014.

[29] S. Li, K. Chen, J. H. Ahn, J. B. Brockman, and N. P. Jouppi. CACTI-P: Architecture-Level Modeling for SRAM-based Structures with Advanced Leakage Reduction Techniques. In *2011 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 694–701, 2011.

[30] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins. ADRES: An Architecture with Tightly Coupled VLIW Processor and Coarse-Grained Reconfigurable Matrix. In *13th International Conference on Field Programmable Logic and Application (FPL)*, pages 61–70, 2003.

[31] Micron Technology Inc. TN-41-01: Calculating Memory System Power for DDR3. `https://www.micron.com/support/tools-and-utilities/power-calc`, 2007.

[32] Micron Technology Inc. Mobile LPDDR3 SDRAM: 178-Ball, Single-Channel Mobile LPDDR3 SDRAM Features. `https://www.micron.com/products/dram/lpdram/16Gb`, 2014.

[33] S. Park, K. Bong, D. Shin, J. Lee, S. Choi, and H.-J. Yoo. A 1.93TOPS/W Scalable Deep Learning/Inference Processor with Tetra-Parallel MIMD Architecture for Big-Data Applications. In *IEEE International Solid-State Circuits Conference (ISSCC)*, pages 1–3, 2015.

[34] M. Peemen, A. A. Setio, B. Mesman, and H. Corporaal. Memory-Centric Accelerator Design for Convolutional Neural Networks. In *31st International Conference on Computer Design (ICCD)*, pages 13–19, 2013.

[35] S. H. Pugsley, J. Jestes, H. Zhang, R. Balasubramonian, V. Srinivasan, A. Buyuktosunoglu, A. Davis, and F. Li. NDC: Analyzing the Impact of 3D-Stacked Memory+Logic Devices on MapReduce Workloads. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 190–200, 2014.

[36] B. Reagen, P. Whatmough, R. Adolf, S. Rama, H. Lee, S. K. Lee, J. M. Hernández-Lobato, G.-Y. Wei, and D. Brooks. Minerva: Enabling Low-Power, Highly-Accurate Deep Neural Network Accelerators. In *43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 267–278, 2016.

[37] D. Sanchez and C. Kozyrakis. ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-core Systems. In *40th International Symposium on Computer Architecture (ISCA)*, pages 475–486, 2013.

[38] V. Seshadri, Y. Kim, C. Fallin, D. Lee, R. Ausavarungnirun, G. Pekhimenko, Y. Luo, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry. RowClone: Fast and Energy-efficient in-DRAM Bulk Data Copy and Initialization. In *46th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO)*, pages 185–197, 2013.

[39] V. Seshadri, K. Hsieh, A. Boroumand, D. Lee, M. A. Kozuch, O. Mutlu, P. B. Gibbons, and T. C. Mowry. Fast Bulk Bitwise AND and OR in DRAM. *Computer Architecture Letters*, 14 (2):127–131, 2015.

[40] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar. ISAAC: A Convolutional Neural Network Accelerator with In-situ Analog Arithmetic in Crossbars. In *43rd International Symposium on Computer Architecture (ISCA)*, pages 14–26, 2016.

[41] K. Simonyan and A. Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[42] H. Singh, M.-H. Lee, G. Lu, N. Bagherzadeh, F. J. Kurdahi, and E. M. C. Filho. MorphoSys: An Integrated Reconfigurable System for Data-Parallel and Computation-Intensive Applications. *IEEE Transactions Computers*, 49(5):465–481, 2000.

[43] T. Vogelsang. Understanding the Energy Consumption of Dynamic Random Access Memories. In *43rd Annual ACM/IEEE International Symposium on Microarchitecture (MICRO)*, pages 363–374, 2010.

[44] C. Weis, N. Wehn, L. Igor, and L. Benini. Design Space Exploration for 3D-stacked DRAMs. In *Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1–6, 2011.

[45] X. Yang, J. Pu, B. B. Rister, N. Bhagdikar, S. Richardson, S. Kvatinsky, J. Ragan-Kelley, A. Pedram, and M. Horowitz. A Systematic Approach to Blocking Convolutional Neural Networks. *arXiv preprint arXiv:1606.04209*, 2016.

[46] M. D. Zeiler and R. Fergus. Visualizing and Understanding Convolutional Networks. In *13th European Conference on Computer Vision (ECCV)*, pages 818–833, 2014.

[47] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong. Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks. In *2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, pages 161–170, 2015.