DeftNN: Addressing Bottlenecks for DNN Execution on GPUs via Synapse Vector Elimination and Near-compute Data Fission

Parker Hill[†], Animesh Jain[†], Mason Hill[1†§], Babak Zamirai[†], Chang-Hong Hsu[†]
Michael A. Laurenzano[†], Scott Mahlke[†], Lingjia Tang[†], Jason Mars[†]

[†]University of Michigan, Ann Arbor
{parkerhh, anijain, zamirai, hsuch, mlaurenz, mahlke, lingjia, profmars}@umich.edu

[§]University of Nevada, Las Vegas
hillm3@unlv.nevada.edu

ABSTRACT

Deep neural networks (DNNs) are key computational building blocks for emerging classes of web services that interact in real time with users via voice, images and video inputs. Although GPUs have gained popularity as a key accelerator platform for deep learning workloads, the increasing demand for DNN computation leaves a significant gap between the compute capabilities of GPU-enabled datacenters and the compute needed to service demand.

The state-of-the-art techniques to improve DNN performance have significant limitations in bridging the gap on real systems. Current network pruning techniques remove computation, but the resulting networks map poorly to GPU architectures, yielding no performance benefit or even slowdowns. Meanwhile, current bandwidth optimization techniques focus on reducing off-chip bandwidth while overlooking on-chip bandwidth, a key DNN bottleneck.

To address these limitations, this work introduces DeftNN, a GPU DNN execution framework that targets the key architectural bottlenecks of DNNs on GPUs to automatically and transparently improve execution performance. DeftNN is composed of two novel optimization techniques – (1) synapse vector elimination, a technique that identifies non-contributing synapses in the DNN and carefully transforms data and removes the computation and data movement of these synapses while fully utilizing the GPU to improve performance, and (2) near-compute data fission, a mechanism for scaling down the on-chip data movement requirements within DNN computations. Our evaluation of DeftNN spans 6 state-of-the-art DNNs. By applying both optimizations in concert, DeftNN is able to achieve an average speedup of 2.1× on real GPU hardware. We also introduce a small additional hardware unit per GPU core to facilitate efficient data fission operations, increasing the speedup achieved by DeftNN to 2.6×.

[1]Work was conducted while at the University of Michigan.

MICRO-50, October 14–18, 2017, Cambridge, MA, USA

ACM ISBN 978-1-4503-4952-9/17/10…$15.00
https://doi.org/10.1145/3123939.3123970

CCS CONCEPTS

• **Computing methodologies** → **Neural networks**; • **Computer systems organization** → **Single instruction, multiple data**; • **General and reference** → *Performance*;

KEYWORDS

GPU Architecture, Deep Neural Networks, Memory Bandwidth, Performance Optimization

ACM Reference format:
Parker Hill, Animesh Jain, Mason Hill, Babak Zamirai, Chang-Hong Hsu, Michael A. Laurenzano, Scott Mahlke, Lingjia Tang, and Jason Mars. 2017. DeftNN: Addressing Bottlenecks for DNN Execution on GPUs via Synapse Vector Elimination and Near-compute Data Fission. In *Proceedings of MICRO-50, Cambridge, MA, USA, October 14–18, 2017,* 14 pages.
https://doi.org/10.1145/3123939.3123970

1 INTRODUCTION

As user demand for state-of-the-art technologies in the domains of computer vision, speech recognition, and natural language processing (NLP) continues to increase, system designers are tasked with supporting increasingly sophisticated machine learning capabilities [24, 25]. An important trend that impacts the design of current and future intelligent systems is the convergence of industry toward *deep learning* as the computational engine providing these services. Large companies, including Google [59], Facebook [14], and Microsoft [53], among others [39], are using deep neural networks (DNNs) as the primary technique underpinning machine learning for vision, speech, and NLP tasks.

With an increasing number of queries requiring DNN computation on the critical path, a significant challenge emerges vis-à-vis the large gap between the amount of computation required to process a DNN-based query relative to a traditional browser centric query such as web search [25]. Researchers have recently been investigating the role of accelerators such as ASICs and FPGAs to help bridge this gap [1, 5, 6, 20, 25, 30, 61]. However, these specialized hardware solutions require substantial adjustments across the hardware-software stack as well as re-designing and redeploying of servers which is an obstacle for wide-scale adoption. To avoid this burden on existing infrastructure, deep learning frameworks have embraced commodity accelerators like GPUs [8, 18, 27, 29, 35]. However, significant improvement beyond current GPU performance is needed to bridge the *scalability gap* [25] for DNN computation.

This work is driven by key insight that, much like biological neural networks, DNNs are intrinsically resilient to both minor numerical adjustments [12, 30, 56, 63] and eliding spurious neurons and synapses [21, 22]. This characteristic can be leveraged to achieve performance improvement. However, as we show later in the paper, reduction of computation and data movement does not directly translate to performance improvement. Techniques from prior work either create a mismatch between the algorithm and underlying architecture, or are not designed to address the real hardware bottlenecks, leaving two open challenges in the way of realizing performance benefits:

- **Limitation 1**: *Irregular Computation* – Network pruning [21, 22], a state-of-the-art machine learning technique that reduces the DNN topology focuses on reducing the memory footprint. However, their methodology fails to realize performance benefits on GPUs. Although this technique significantly reduces the amount of raw computation (i.e. floating-point operations), we show that the hardware-inefficient irregular DNN topology outweighs the benefits and results in substantial slowdown (up to 61×) due to increased branch divergence and uncoalesced memory access on GPUs. We present details on this limitation in §2.1. To achieve performance benefits, the challenge of reducing computation while aligning the reduced computation with underlying hardware must be addressed.
- **Limitation 2**: *Not Optimized for Bottleneck* – Our investigation identifies on-chip memory bandwidth to be the key bottleneck for DNN execution on GPUs. However, prior works focus on improving off-chip memory bandwidth using compression [58], removing non-contributing bits to increase the effective bandwidth. This technique, however, fails to provide significant speedups for DNNs (details in §2.2). We evaluate off-chip data packing and observe a speedup of less than 4%. On the other hand, compared to off-chip techniques, it is more challenging to perform on-chip compression because frequently reformatting data is difficult to achieve without introducing significant overhead.

This work introduces *DeftNN*, a GPU DNN execution framework that addresses these limitations. Firstly, *synapse vector elimination* reduces the total problem size by automatically locating and discarding non-contributing synapses in the DNN – those synapses having negligible impact on the output results – to improve performance. To address the limitation of irregular computation, our insight is that it is necessary to preserve existing architectural optimizations in original GPU-efficient applications. Utilizing this insight, synapse vector elimination applies a novel transformation to the DNN data layout, producing computations that efficiently leverage GPU hardware.

The second optimization, *near-compute data fission*, mitigates the GPU on-chip memory bandwidth bottleneck by optimizing the utilization of integer units during DNN execution. To address the prior work's limitation of providing only off-chip bandwidth optimization [58], as on-chip memory is closer to the functional units, we design novel techniques that can support low-overhead very fine-grained data conversion. The key insight that makes near-compute data fission feasible is that the focus of data conversion must be shifted from high compression ratio to low decompression overhead. In addition to the benefits achieved by our carefully optimized near-compute data fission technique on commodity hardware, we describe a small additional unit called the *Data Fission Unit* (DFU) that can be added to existing GPU hardware to obviate data fission overhead to realize additional benefits on future generations of GPU hardware. The specific contributions of this work are:

- **DeftNN.** We introduce DeftNN, a state-of-the-art GPU DNN execution framework. This framework automatically applies synapse vector elimination and near-compute data fission optimizations to existing DNN software applications to dramatically improve performance on today's GPUs.
- **Synapse Vector Elimination.** We introduce a DNN optimization technique for GPUs, synapse vector elimination, that shrinks the topology of the neural network. This method is guided by the insight that network pruning techniques in DNN systems must have computational regularity to achieve significant speedups. Our experiments show that synapse vector elimination achieves 1.5× average end-to-end speedups on a set of 6 state-of-the-art DNNs on real GPU hardware.
- **Near-compute Data Fission.** We introduce near-compute data fission, which improves performance by efficiently packing on-chip memory. To realize speedup, we find that the focus must be shifted from minimizing data size to minimizing unpacking overhead. We find that near-compute data fission provides 1.6× end-to-end speedup on a set of 6 DNNs on real GPU hardware available today by performing unpacking in software. We also introduce a lightweight hardware extension (<0.25% area overhead) to facilitate efficient unpacking, achieving an additional 1.4× speedup over software-only near-compute data fission.
- **Real System Evaluation of DeftNN.** We evaluate DeftNN on real GPU hardware across 6 state-of-the-art DNNs, covering both fully connected and convolutional layers – the most computationally intensive DNN layer types. We show that DeftNN achieves significant performance improvements yielding 2.1× speedups on average.

2 CHALLENGES

In this section, we describe the key ideas and challenges in applying real-system, GPU-based optimizations to DNNs.

2.1 Computation Elimination

Network pruning [21, 22] has been proposed to remove non-contributing synapses and neurons by removing those with near-zero values. These removed computations occur sporadically throughout the DNN topology, limiting benefits on commodity architectures.

GPU hardware, requiring contiguous data structures for efficient execution, presents a significant challenge when omitting arbitrary neurons or synapses. Specifically, for GPUs, branch divergence [15] and uncoalesced memory access [26] present two performance pitfalls for execution on noncontiguous data structures:

(1) **Branch divergence** is where some of the threads, partitioned into groups by hardware (e.g., warps in CUDA), need to execute different instructions than the other threads in its group [45]. The hardware is designed such that all of the threads in a group execute instructions in lockstep. This requires that divergent

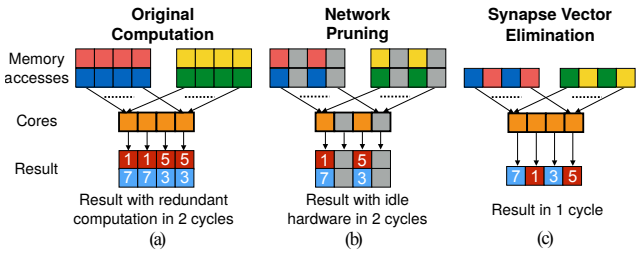

Figure 1: (a) Original DNN computation resulting in redundant computation, (b) network pruning [21, 22] resulting in underutilized hardware, and (c) synapse vector elimination showing efficient use of resources.

sections of code are executed sequentially, so omitted computation that occurs irregularly due to noncontiguous data structures results in idle hardware rather than more efficient execution.

(2) **Uncoalesced memory access** is a similar issue in the memory subsystem [26]. When multiple threads in a thread group issue memory instructions, requests to consecutive addresses, a result of contiguous data structures, can be grouped together to utilize a wide memory bus. Values stored in noncontiguous data structures are unlikely to have consecutive addresses, causing substantial underutilization of the memory bus.

The challenge faced by network pruning is illustrated in Figure 1. An example baseline computation is presented in Figure 1(a). The original computation takes two cycles to complete because there are eight inputs with four being completed each cycle. Figure 1(b) shows the computational pattern produced by network pruning where uncoalesced memory accesses, due to the sparse, noncontiguous data structure, prevent high utilization of the arithmetic cores. Therefore, it still takes two cycles to process four inputs, since only two inputs are being processed per cycle. In practice, this kind of sparse computation results in very poor hardware utilization because it fails to take advantage of the GPU's very wide vector units. To validate the necessity of addressing this challenge, we compare the performance of DNN inference subjected to contiguous and noncontiguous data structures. In our real-system GPU experiments, we observe that applying noncontiguous data structures, produced by network pruning, to DNN inference results in a slowdown of 61× (§5.8).

To efficiently reduce the DNN topology, we propose synapse vector elimination, which improves DNN performance during inference by discovering and removing performance-exploitable noncontributing synapses. Synapse vector elimination overcomes the sparsity challenge by applying dynamically transformed input data to the original hardware-efficient computation. As shown in Figure 1(c), synapse vector elimination reduces the total execution time by efficiently utilizing hardware resources on the transformed input. As shown in the diagram, our methodology results in only one cycle to process four inputs, since the removed and retained data is not interleaved. More details on our synapse vector elimination technique are presented in §4.1.

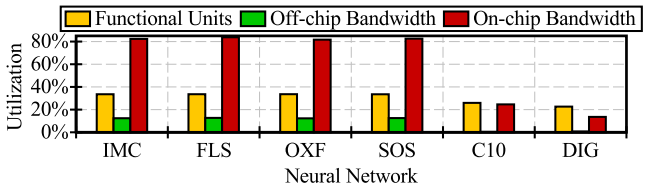

Figure 2: GPU utilization when processing DNNs, showing the on-chip memory bandwidth bottleneck.

2.2 On-chip Memory Bandwidth

In contrast to reducing the amount of work with synapse vector elimination, another approach to achieve speedup is to alleviate the DNN processing bottleneck on GPUs by effectively exchanging one hardware resource for another. There are three main hardware resources on a GPU that are susceptible to becoming a bottleneck: the functional units, the off-chip memory bandwidth, and the on-chip memory bandwidth. We present kernel-weighted average utilization metrics of these three components in Figure 2, which were produced by profiling 6 state-of-the-art DNNs (application details are presented in §5.1), using Caffe [29] and running on an Nvidia Titan X (Pascal) GPU.

The key takeaway from the figure is that the system is greatly limited by on-chip memory bandwidth. This utilization profile is a result of optimized matrix multiplication, the main underlying GPU kernel for DNN inference, which makes use of loop tiling [7]. Loop tiling optimization allows on-chip memory storage and registers to be traded for off-chip memory bandwidth and on-chip memory bandwidth, respectively. While there is sufficient on-chip memory storage to sufficiently reduce off-chip memory bandwidth, the on-chip memory bandwidth remains a bottleneck due to the limited number of registers available for loop tiling.

As an example, the state-of-the-art Titan X (Pascal) GPU provides 11 single-precision TFLOPS (i.e. 44 TB/s), but its on-chip memory bandwidth is limited to 3.6 TB/s ($frequency \times \#\ shared\ memory\ banks \times bus\ width$ = 1 GHz × 28 banks × 128 bytes) [51]. While loop tiling at the register level mitigates this throughput gap, on-chip memory bandwidth is still the limiting resource due to the limited number of registers available for tiling.

To alleviate the on-chip memory bandwidth bottleneck, unused functional unit cycles can be leveraged to compress on-chip memory. Unfortunately, the most recent GPU memory compression technique only applies to off-chip memory [58]. This technique works by compressing the data in off-chip memory, while storing the decompressed data in on-chip memory. Although this can reduce off-chip memory bandwidth, it provides no benefit for DNNs because on-chip memory bandwidth is the performance bottleneck.

Moving existing memory compression techniques closer to the functional units is more complex than simply applying the compression technique at a different place in the memory hierarchy. The central challenge when moving the compressed data closer to the compute is that the decompression overhead can outweigh the gains of reduced memory bandwidth and storage. The bandwidth for on-chip memory, however, is much greater than that of off-chip memory, making the size of the compressed data format less critical. The differences in proximity to functional units and available

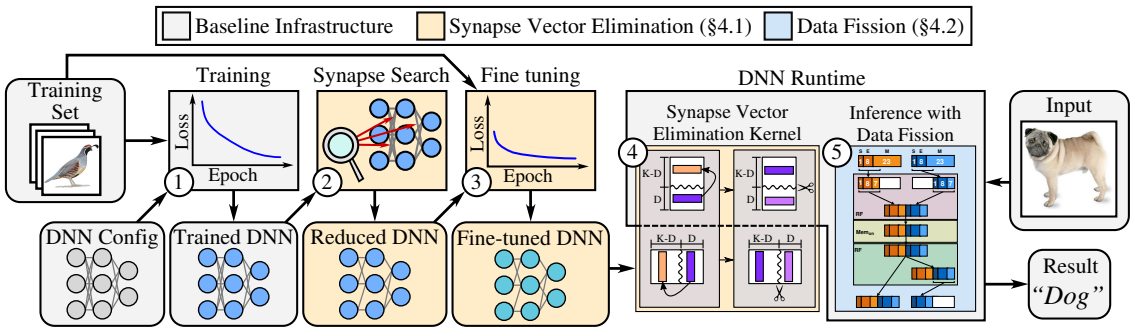

Figure 3: Overview of the DeftNN framework.

bandwidth cause a fundamental shift in the compression design space. While off-chip data packing focuses on larger reductions in memory bandwidth, a solution to this problem for DNNs must focus on minimizing decompression overhead.

Our near-compute data fission technique mitigates the GPU bottleneck in the system by targeting on-chip memory bandwidth. It realizes speedup by treating fission overhead as the paramount characteristic of the design. More details on our near-compute data fission technique are presented in §4.2.

3 SYSTEM OVERVIEW

In this section, we present an overview of the DeftNN system, a GPU DNN execution framework for optimizing DNN inference by tailoring it to the underlying architecture. The two optimizations, synapse vector elimination and near-compute data fission, are built upon a standard DNN software framework, comprising an offline training phase for optimizing the DNN topology and a runtime system. Together, the offline and runtime systems work in concert to apply optimizations automatically and transparently to unmodified DNN applications. An overview diagram of the DeftNN framework is presented in Figure 3.

① **Initial Training –** First, as in all DNN execution frameworks, a set of training inputs are used along with a DNN configuration that specifies the topology of the DNN. Using the training inputs, the DNN parameters are adjusted iteratively until the classification loss function converges. This process produces a trained DNN model.

② **Synapse Search –** After producing the baseline trained DNN model, DeftNN automatically performs a synapse search to find the non-contributing synapse vectors – groups of synapses that are architecturally efficient to eliminate on the GPU. This process, as detailed in §4.1.2, locates and removes non-contributing synapse vectors, which we define as any vector highly correlated with another vector. As illustrated in the figure, the synapse search results in a DNN model that has some set of synapse vectors eliminated from the computation.

③ **Fine Tuning –** Although the retained synapse vectors are chosen to be representative of those that were eliminated, the nuanced impact of the missing, eliminated synapse vectors can result in accuracy degradation if used directly. To remedy this, DeftNN uses fine tuning, a well-known technique used to refine the DNN weights by applying a small number of DNN training iterations [64]. This process allows the DNN model to fully recover accuracy that is lost from minor perturbations of the weights or topology. By using fine tuning after applying synapse vector elimination, DeftNN produces a DNN model with negligible loss in inference accuracy.

④ **Synapse Vector Elimination –** Beyond the training mechanism used by DeftNN to produce an efficient DNN model, DeftNN services DNN applications using a runtime system that seamlessly allows inputs formatted for the unoptimized DNN model to be applied to the DNN model from synapse vector elimination. The synapse vector elimination kernel, as shown in the figure, reorganizes input activation values prior to inference so that they can be applied to the optimized DNN model. A detailed description of the architecture-efficient synapse vector elimination kernel are provided in §4.1.1, though we note that the reorganization takes a maximum of 5% of kernel execution time (see Figure 6), an overhead dwarfed by the reduction in computation facilitated by synapse vector elimination.

⑤ **Near-compute Data Fission –** In addition to reducing the size of the DNN using synapse vector elimination, DeftNN optimizes the key GPU bottleneck, on-chip memory bandwidth, within the inference kernel using near-compute data fission. Near-compute data fission packs DNN weights and activations into on-chip memory by removing non-contributing bits from the numerical representation. Since this technique resides in the low-level computational DNN kernels at runtime, no further changes are required to the baseline infrastructure to utilize this optimization. A detailed description of near-compute data fission is presented in §4.2.

4 OPTIMIZATION TECHNIQUES

In this section, we describe two novel optimization techniques, synapse vector elimination and near-compute data fission. We present the key insights and challenges of both techniques when implemented on real GPU systems executing DNN workloads. In addition to our real-system implementation, we observe that near-compute data fission is amenable to acceleration and design a light-weight GPU hardware extension to mitigate overhead. Synapse vector elimination has minimal overhead when implemented in software, not warranting the costs of additional hardware.

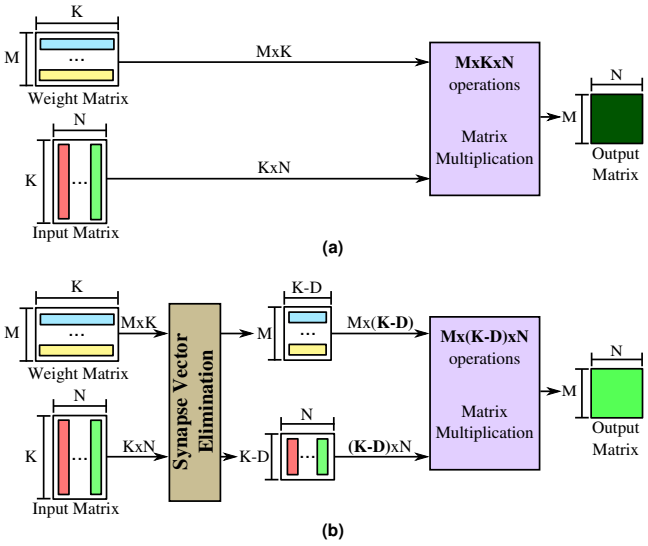

Figure 4: High-level view of synapse vector elimination, showing that (a) the exact computation is an $M \times K$ by $K \times N$ matrix multiplication, while (b) synapse vector elimination preprocesses the input to make the computation an $M \times (K - D)$ by $(K - D) \times N$ matrix multiplication.

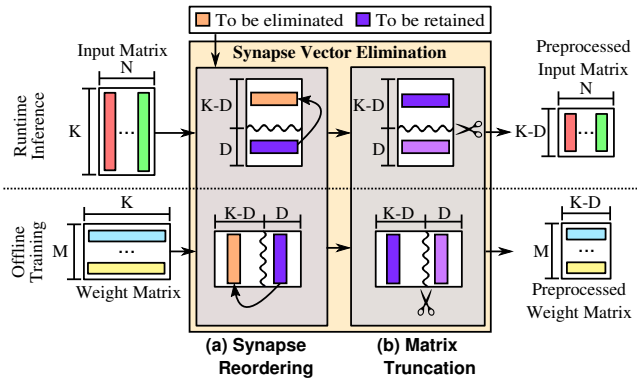

Figure 5: Internal workings of synapse vector elimination, showing compacting retained synapses so that the matrix can be trivially resized.

4.1 Synapse Vector Elimination

Synapse vector elimination removes non-contributing synapses from DNNs, thereby reducing the total computation required for the DNN to process its inputs. Previous network pruning techniques produce an inefficient mapping of operations to hardware. Specifically, these techniques modify the computational kernel to be irregular, limiting performance benefits due to branch divergence and uncoalesced memory access. Instead, synapse vector elimination retains a hardware-efficient design by transforming DNN inputs for similarly-structured but smaller DNN computations.

Many DNNs have a large number of synapses that can potentially be eliminated. Without considering the underlying architecture, the selection of non-contributing synapses is fairly straightforward: network pruning techniques simply select the synapses with the lowest weights [21, 22]. One of the insights motivating this work is that the granularity of synapses that should be removed is constrained by the architecture, thus the selection of synapses becomes a multi-dimensional optimization problem. We devise a novel search technique to solve this problem based on the correlation matrix formed by the architectural groups of synapse weights.

4.1.1 Architecture-efficient Design. Here, we present the GPU architecture-efficient design of synapse vector elimination, our technique that avoids the performance pitfalls associated with network pruning, as described in §2.1, by applying a preprocessing step to efficiently rearrange computation. First, we show a high-level view of the approach in Figure 4. The original neural network computation (Figure 4(a)) is carried out by multiplying the $M \times K$ weight matrix by the $K \times N$ output matrix of the previous layer. The synapse vector elimination variant of this operation (Figure 4(b)) preprocesses the input and weight matrices to reduce the total problem size. The weight matrix is preprocessed offline since it is reused many times, while the input matrix is preprocessed during runtime to allow seamless switching between the original computation and the synapse vector elimination optimized computation. These smaller matrices are then given to a standard matrix multiplication algorithm. The performance benefits are realized by applying an inexpensive transformation that reduces the size of the inner dimension (i.e. K in the figure) of the matrix multiplication.

There are two main steps for our synapse vector elimination transformation, as shown in Figure 5: synapse reordering and matrix truncation. First, synapse reordering efficiently repositions rows and columns of the neural network matrices so that they are easier to manipulate. Next, matrix truncation reduces the amount of computation required for matrix multiplication while preserving its uniform data structure. Finally, a correction factor is applied to the matrices to retain the scale of the output values.

Synapse Reordering. First, we reorder synapses to simplify the task of discarding unwanted synapses. The central goal of reordering is to preserve the data structure's uniformity without diminishing the gains of skipping synapses, so a quick method of grouping the retained and discarded synapses is necessary. We group the synapses (rows in the weight matrix and columns in the input matrix) together based on whether they will be discarded or retained. For brevity, we only describe the reordering of the weight matrix, but an equivalent reordering is applied to the transpose of the input matrix. Since the number of discarded synapse weights, D, is known before we start reordering synapses, we partition the matrix at column $K - D$ so that the $K - D$ columns on the left represent the retained synapse group and the D columns on the right represent the discarded one.

After defining this partition point, some of the synapses that are to be retained are already contained in the retained synapse partition. In fact, there is an equal number of synapses to be retained as discarded that are in the incorrect partition. By using this observation, we create a pairing between misplaced retained synapses and misplaced discarded synapses. The matrix is then reordered by swapping the two columns for each of these pairs. After swapping

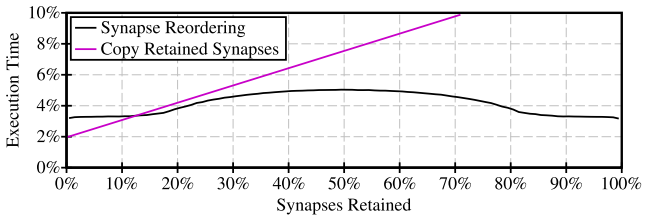

Figure 6: Overhead from synapse reordering compared to copying all retained synapses, showing that DeftNN substantially reduces the overhead of input transformation by reordering synapses at useful design points (>50% retained).

all of the misplaced columns, the retained and discarded synapses are strictly separated at column $K - D$.

To determine the benefits of our synapse reordering method compared to naively copying all retained synapses into a separate buffer, we evaluate the overhead of these two methods in Figure 6 as a percentage of end-to-end execution time for AlexNet (the same trend is present for all of our evaluated applications). In our experiments, we found that more than 50% of synapses are needed to retain accuracy. Using this discarding rate, we observe that synapse reordering is at least 1.4× faster than copying retained synapses. We find that it is impractical to design hardware for synapse reordering, since there is little overhead involved in synapse vector elimination.

Matrix Truncation. Next, we reduce the dimensions of the input matrices to reduce the required amount of computation. To describe the matrix truncation step, we supply the formula for computing the value of a neuron (i.e. a cell of the output matrix) in Equation 1, where *Out* is the output matrix, W is the weight matrix, *In* is the previous layer matrix, i is the input index (e.g., the convolution kernel index or the fully connected input vector index), and j represents the input neuron index.

$$Out_{i,j} = \sum_{k=1}^{K} W_{i,k} In_{k,j} \tag{1}$$

Note that, after reordering the matrices, the output of matrix multiplication is the same as it would be without reordering; only the order of the weighted sum is changed. Thus, the output is equivalently shown in Equation 2, where the $K - D$ retained synapses in the reordered matrices, W' and In', are summed first and then the D discarded synapses are summed.

$$Out_{i,j} = \sum_{k=1}^{K-D} W'_{i,k} In'_{k,s} + \sum_{k=K-D+1}^{K} W'_{i,k} In'_{k,j} \tag{2}$$

In order to remove the computation for the discarded synapses, the summation is stopped at $K - D$ instead of at K. In terms of the DNN matrix multiplication, we cut the last D columns from the input neuron matrix and the last D rows from the weight matrix.

Scale Adjustment. To compensate for the discarded synapses in each summation, we increase the magnitude of the retained synapses so that the expected value of the original and optimized results match. If we assume that the synapses are all drawn from a similar distribution, then the expected value is equal to the expected value of any single synapse multiplied by the number of synapses, shown in Equation 3.

$$\mathbf{E}(Out_{i,j}) = \mathbf{E}(\sum_{k=1}^{K} W_{i,k} In_{k,s}) = K\mathbf{E}(W_{i,x} In_{x,j}) \tag{3}$$

The expected value of this sum, after removing the discarded synapses, can be represented similarly (Equation 4).

$$\mathbf{E}(Out'_{i,j}) = \mathbf{E}(\sum_{k=1}^{K-D} W'_{i,k} In'_{k,s}) = (K - D)\mathbf{E}(W_{i,x} In_{x,j}) \tag{4}$$

To match the expected value from synapse vector elimination to the original expected value, the weighted sum is scaled by the ratio between the unadjusted expected value from synapse vector elimination and the original expected value. This produces our final expression for the synapse discarded summation, as shown in Equation 5.

$$Out''_{i,s} = \frac{\mathbf{E}(Out_{i,s})}{\mathbf{E}(Out'_{i,s})} \sum_{k=1}^{K-D} W'_{i,k} In'_{k,s} = \frac{K}{K-D} \sum_{k=1}^{K-D} W'_{i,k} In'_{k,s} \tag{5}$$

4.1.2 Synapse Search. Equipped with a method to efficiently discard synapses, we now describe how we find which synapses are not contributing to the final output. Trying all combinations of synapses is not tractable, since there are thousands of synapses and each synapse is a binary variable that can be either retained or discarded (i.e. there are $2^{\#synapses}$ possibilities). When reducing the DNN at a per-synapse granularity, prior works discard synapses with near-zero weights [21, 22], avoiding the need for a sophisticated search mechanism. Although this pruning strategy is effective for pruning sporadic synapses, our insight is that GPU-efficient optimizations must discard synapses in groups to exploit wide vector unit hardware. We evaluate the necessity of this insight by comparing to these prior works that discard sporadic synapses in §5.8. A synapse vector pruning search mechanism must choose to retain or discard each architectural group of synapses, referred to as synapse vectors, rather than single ones. This presents a new challenge, since no synapse vectors have enough near-zero weights to be discarded as is done in these prior works.

Instead of discarding synapses with weights nearest to zero, we aim to retain a subset of the synapse vectors that are representative of the entire set of synapses. To select contributing synapses, synapse vector elimination starts by computing the correlation matrix, ρ, for the synapse vectors, as shown in Equation 6, where S_x is the synapse vector for the group of synapses at index x.

$$\rho_{i,j} = \frac{covariance(S_i, S_j)}{\sqrt{var(S_i) var(S_j)}} \tag{6}$$

For each synapse vector, S_i, we find the set of synapse vectors that it can represent. We define S_i to be representative of S_j if the correlation between the two synapse vectors ($\rho_{i,j}$) is above the representative correlation threshold, α. This is shown in Equation 7.

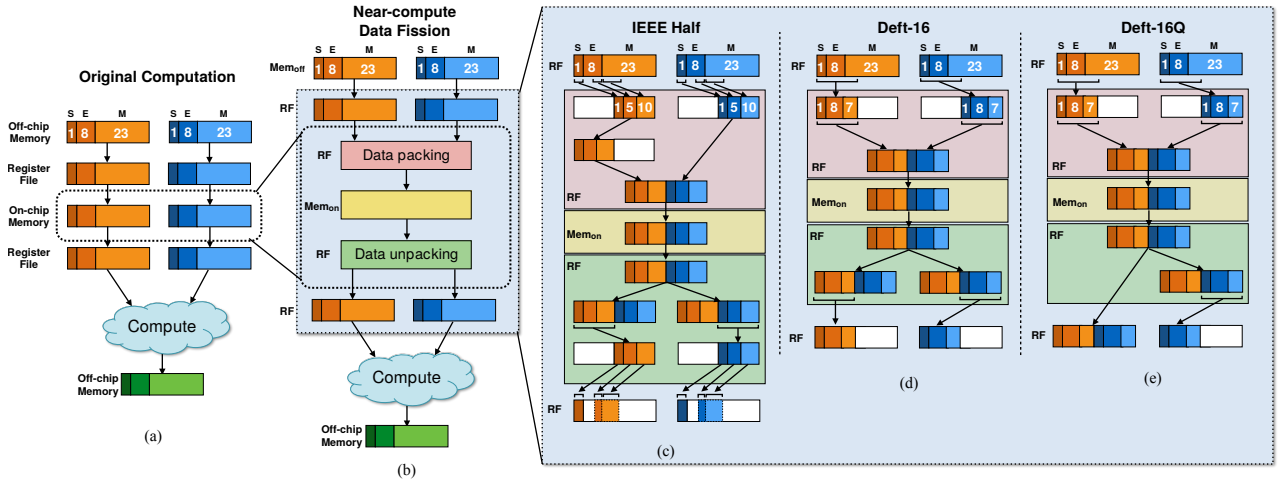

Figure 7: (a) The original design, (b) high level view of a design with near-compute data fission, (c) fission using the IEEE 754 single precision floating-point to the half precision variant, (d) fission using the Deft-16 floating-point format, and (e) fission using the optimized Deft-16Q floating-point conversion format.

$$R_i = \sum_{j=1}^{N} \begin{cases} 1 & \rho_{i,j} \geqslant \alpha \\ 0 & \text{otherwise} \end{cases} \tag{7}$$

The synapse vector that represents the most synapse vectors, R_i, is selected to be retained in the output DNN from synapse vector elimination, while the non-contributing synapse vectors represented by the retained one are removed. This process is repeated on the remaining synapse vectors, until all synapse vectors are either retained or discarded.

4.1.3 Exposing Further Performance Opportunities. In addition to selecting non-contributing synapse vectors, synapse vector elimination can be parameterized to discard marginally-contributing ones by adjusting the representative correlation threshold. As the correlation threshold is lowered, the number of synapse vectors that can be represented by a single synapse increases. This capability can be used to enact approximate computing, essentially shedding small amounts of accuracy to realize improved performance.

The number of such readily available performance-accuracy trade-off configurations is limited due to large DNN memory footprints, if each configuration is stored in memory separately. To greatly increase the flexibility of synapse vector elimination, applied to marginally-contributing synapses, DeftNN dynamically builds DNNs using combinations of layers that were trained with varying correlation thresholds. Each <layer, correlation threshold> pair is fine-tuned independently of the others, allowing arbitrary combinations of these pairs to be composed during runtime without requiring a new DNN model for each combination.

Given a performance or accuracy constraint, DeftNN must quickly select an appropriate set of correlation thresholds for each of the layers. To do this, DeftNN is configured to build a Pareto frontier of configurations during training and to select the configuration that is nearest to the user-specified goal during runtime. We evaluate the idea of using DeftNN for approximation in §5.6.

4.2 Near-compute Data Fission

In this section, we present our near-compute data fission technique, which fuses multiple values into a single value of lesser size in on-chip memory to improve effective bandwidth. Near-compute data fission directly improves performance, since DNN computation is bottlenecked by on-chip memory bandwidth.

Although on-chip memory bandwidth is the key limitation of DNN performance, fission at this level of the memory hierarchy requires very frequent data reformatting, causing excessive overhead, unless the data format is carefully chosen. We start with a standard CUDA-supported half precision format, but find that it is insufficient for near-compute data fission and devise a new format that results in far better performance due to reduced reformatting overheads. To exploit the non-contributing bits further, by reducing the reformatting overhead, we introduce hardware to allow conversion to narrower numerical representations.

4.2.1 Technique. Before addressing the challenge of efficient near-compute data fission, we describe each of the GPU components that are relevant to our near-compute data fission method. Figure 7(a) shows a baseline implementation, in a GPU context, with no fission. The data is first loaded from off-chip memory into registers. To improve performance, the values in registers are stored into an on-chip memory scratchpad for future reuse. The application reads from and computes on the data stored in scratchpad memory many times. Finally, the result is written to off-chip memory.

Figure 7(b) shows an extension of this baseline with near-compute data fission added to the system. As before, data is loaded from the off-chip memory into the register file. Instead of writing directly to the scratchpad memory, multiple values are fused into a single element. Similarly, each time the application reads from the scratchpad memory, fission is applied to the value before it is computed on. This process removes the non-contributing bits from the numerical representation in the on-chip memory. We do not store fused data

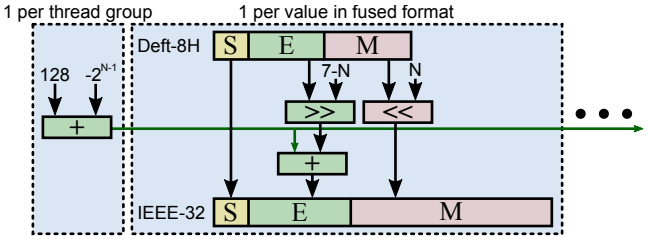

Figure 8: Fission in the DFU, showing hardware to apply fission to an 8-bit, variable exponent length (N) floating-point value to an IEEE single-precision value.

into the off-chip memory because, as shown before, the off-chip memory utilization is already very low.

4.2.2 Format Optimization. We investigate 3 near-compute data fission formats, as shown in the figure. The formatting process for these types are shown in Figure 7(c-e). All 3 of the fused data formats are reduced precision floating-point representations with a sign (S), a mantissa (M) that specifies the precision, and an exponent (E) that denotes dynamic range. In Figure 7(c), we start with the IEEE 754 half precision data format. In our evaluation, we find that this format results in excessive reformatting overhead, resulting in slowdown, due to the complex conversion taking several cycles.

The next technique shown in Figure 7(d), Deft-16, takes advantage of a special floating-point format defined to be the 16 most significant bits of the IEEE single precision floating-point format. This format is a data type with 8 exponent bits and 7 mantissa bits, which provides sufficient precision and dynamic range for DNN workloads. To apply fission to this value, only inexpensive shift and bitwise operations are necessary.

Finally, we observe that we can reduce another instruction from the fission process by allowing the most significant bits of one value to spill into the least significant bits of the other value. We call this the Deft-16 quick format (Deft-16Q) and show the set of operations for fission in Figure 7(e). Despite only reducing the fission process by a single logical AND instruction, we find substantial performance difference between Deft-16Q and Deft-16.

Nevertheless, while this optimized fission process can be applied to today's commodity hardware, its design is specific to 16-bit data and introduces a non-trivial amount of overhead. To address both of these limitations, we next describe a small additional hardware unit to perform the fission operation.

4.2.3 Hardware Acceleration. Since the fission operation is on the critical path when applying near-compute data fission, we introduce a lightweight GPU hardware extension called the *Data Fission Unit* (DFU) to accelerate fission. The DFU is replicated for each floating-point unit to maintain high throughput, so our central design goal of the DFU is minimizing area overhead. For this reason, the DFU is specialized for the data representations that are most likely to be beneficial. The DFU is specifically targeted to accelerate the fission of custom 8-bit floating-point and Deft-16Q representations.

ISA Extension. DFU fission operations are accessed with a parallel thread execution (PTX) [52] ISA extension. We add two new instructions to PTX, `dfu_cvt_16` and `dfu_cvt_8`, which provide the ability to invoke the 16-bit and 8-bit DFUs, respectively. The 16-bit DFU operation is parameterized with a source `.b32` (a 32-bit conversion-only data type in PTX) register and two contiguous `.f32` (a 32-bit floating-point data type in PTX) destination registers. The 8-bit DFU operation is similar, except it is parameterized by four destination registers and an immediate floating-point exponent bitwidth. The flexibility of variable-width exponent allows low-precision 8-bit values to be more versatile, outweighing the negligible area cost (provided in §5.5).

Architecture Integration. The `dfu_cvt` instructions are executed by the DFU, which is integrated into the microarchitecture as an extension of the ALU. This extension adds a DFU to each floating-point unit, so the conversion throughput is sufficiently high to provide enough data for all of the floating-point units. In §5.5, we find that the area-efficient design of the DFU requires only 0.22% area overhead when replicated for each floating-point unit. In addition to allocating a DFU for each floating-point unit, we increase the throughput of the DFU by requiring that the 32-bit floating-point destination registers are contiguous. Using contiguous registers allows the DFU to use 64-bit and 128-bit register write operations when writing two and four 32-bit values, as produced by 16-bit and 8-bit data fission, respectively.

Conversion Details. The hardware design for applying 16-bit fission in the Deft-16Q format is straightforward, as it requires only a single zero-padded bitwise shift to prepare two values for computation. The hardware for 8-bit fission is illustrated in Figure 8. The 8-bit floating-point representation is shown at the top of the figure, with the sign bit denoted by "S", the exponent bits denoted by "E", and the mantissa bits denoted by "M".

The size of the exponent can be adjusted from 7 bits to 1 bit, denoted by N in the figure, depending on the exponent length encoded into the DFU instruction. Floating-point representations encode the exponent with a fixed offset, the bias, based on the bit width of the representation. Since this bias is different between the 32-bit representation and the 8-bit DeftNN representations, the DFU finds the difference between the two biases (adder on the left side of the figure), and then adds this difference to the exponent bits of the fused values. Because the GPU architecture executes threads in each thread group in lockstep, we reuse the bias difference when applying fission to all of the fused values in a given thread group.

The mantissa bits, also of variable length, of the fused representation are shifted to the left, so that the most significant bit is aligned with the most significant bit of the 32-bit value. After alignment, the value is zero-padded to 23 bits and used as the mantissa of the 32-bit representation. The sign bit is directly transferred from the 8-bit representation to the 32-bit one. Leveraging the DFU, which provides single-cycle fission operations, we can significantly reduce the cost of performing DeftNN near-compute data fission.

5 EVALUATION

We evaluate DeftNN to determine its efficacy on improving DNN performance. We examine each of the key components of the system: synapse vector elimination, near-compute data fission, and the complete DeftNN runtime system.

Name	Neural Network	# Classes	Dataset
IMC	AlexNet [34]	1000	ImageNet [57]
FLS	Flickr Style [31]	20	FS-20 [31]
OXF	Flower Species [48]	102	Flower-102 [48]
SOS	SOS CNN [65]	5	SOSDS [65]
C10	CifarNet [33]	10	CIFAR-10 [33]
DIG	LeNet-5 [40]	10	MNIST [41]

Table 1: The set of benchmarks used to evaluate DeftNN.

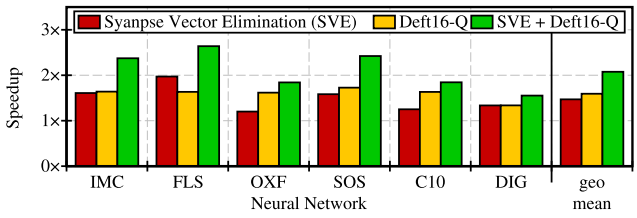

Figure 9: Speedup achieved by DeftNN when applying synapse vector discarding, data fission, and the combination of the two, showing the significant benefits of each technique and their efficacy when applied in concert.

5.1 Methodology

DeftNN is evaluated using a real-system GPU implementation with robust open source frameworks. Our implementation is built upon Caffe [29], a neural network framework developed by the Berkeley Vision and Learning Center. Caffe provides the high-level neural network functionality and offloads the computational kernels to BLAS. We use MAGMA [46] as the BLAS implementation, a fast open source CUDA implementation. We take measurements on a machine containing a Xeon E5-2630 v3 CPU and an Nvidia Titan X (Pascal) GPU, a configuration representative of state-of-the-art commodity hardware.

Table 1 summarizes the set of applications used in our evaluation. To gain an accurate representation of real DNN workloads, we use the trained neural network models deployed in Caffe and designed by the machine learning community. For each benchmark, we use the given machine learning task's canonical validation and training datasets. We randomly select 500 inputs from the validation set for speedup and accuracy measurements and use the entire training set for fine tuning.

5.2 Overall DeftNN System

We begin by evaluating the end-to-end real-system GPU performance characteristics of DeftNN when applying both synapse vector elimination and near-compute data fission. In these experiments, we follow the steps outlined in §3 to automatically and transparently optimize the 6 DNNs covered in the evaluation. Figure 9 shows the results of these experiments. We first observe that applying each of the two optimization techniques in isolation provides significant speedup, 1.5× and 1.6× geometric means across the applications for synapse vector elimination and near-compute data fission, respectively. When both techniques are applied, DeftNN provides an average speedup of 2.1×, showing the substantial performance benefit of deploying DeftNN.

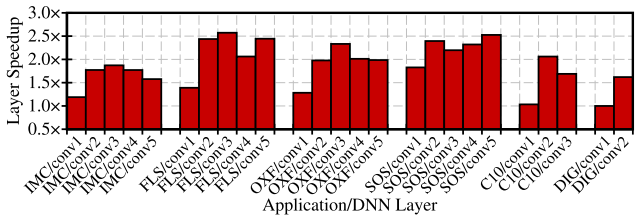

Figure 10: Per-layer speedup when applying synapse vector elimination, showing large performance improvements, particularly for the large DNNs (IMC, FLS, OXF, and SOS).

5.3 Synapse Vector Elimination

We next evaluate synapse vector elimination in isolation to provide insight into its workings and characteristics, focusing on the layer-by-layer speedup achieved by synapse vector elimination. We present the per-layer performance improvements in Figure 10, showing that synapse vector elimination is capable of optimizing nearly every individual layer across the DNNs. We note that C10 and DIG observe the smallest performance improvements from synapse vector elimination. These networks are the smallest of our evaluated applications in terms of the number of layers as well as the size of each layer. Small DNN topologies limit the available parallelism on GPU architectures, causing lower utilization of hardware. As a consequence, substantial reductions in the topology of the DNN may result in under-utilization of GPU resources and limit the speedup that can occur. Nevertheless, synapse vector elimination is able to improve the performance by at least 1.5× for all but the smallest layers (the input layers).

5.4 Near-compute Data Fission

To evaluate our near-compute data fission technique, we compare the three fission formats described in §4.2.2, the baseline computation, and the computation with 16-bit compute and storage. The baseline computation is produced without fission, which uses the IEEE 754 single precision 32-bit floating-point format for computation and storage throughout the memory hierarchy. Comparisons of these near-compute data fission strategies are shown in Figure 11.

Speedup. Figure 11(a) presents the speedup achieved during end-to-end DNN inference for each of the three fission formats. First, we consider the 16-bit computation and storage configuration, FP16 in the figure. We observe that 16-bit computation results in a 14× slowdown due to state-of-the-art GPUs having many more 32-bit ALUs than 16-bit ALUs.

Next, we consider the IEEE half precision format. The IEEE Fission results represent a CUDA mechanism to convert the fused IEEE half precision values into single precision values, which leverages specialized bit-convert hardware. Due to the limited amount of hardware allocated for these conversion instructions, the IEEE conversion process imposes significant overhead, resulting in slight slowdown rather than speedup. Our novel fission formats, Deft-16 and Deft-16Q, result in the same change in data size, but both

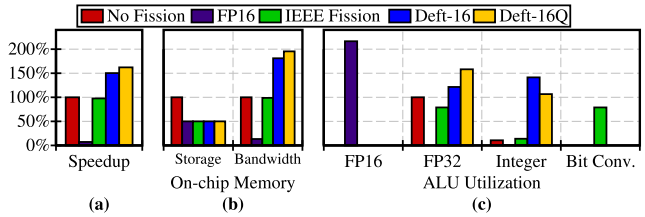

Figure 11: (a) Speedup from using 16-bit compute (FP16) and fission in three different formats (IEEE Fission, Deft-16, and Deft-16Q) compared to 32-bit storage and computation (No Fission) along with (b,c) pertinent profiling metrics, showing that Deft-16Q, achieves the highest performance because of improved effective on-chip memory bandwidth.

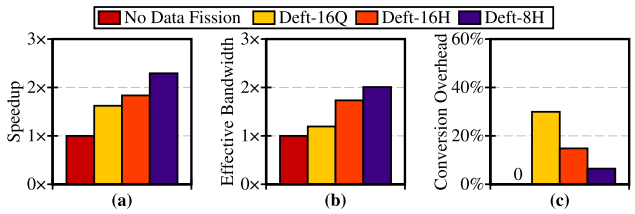

Figure 12: Comparison of no near-compute data fission, software-only fission (Deft-16Q), and hardware accelerated 16-bit (Deft-16H) and 8-bit (Deft-8H) fission, showing that (a) performance is improved as (b) effective on-chip bandwidth is increased with smaller representations, without (c) restrictive conversion overhead.

achieve over 50% improvement in end-to-end performance showing that the complexity of data type conversion is critical.

Profiling Details. All three of the near-compute data fission formats yield half of the storage requirements for on-chip memory, shown in Figure 11(b), since the values in each format occupy 16 bits instead of 32 bits. The key benefit of fusing data into on-chip memory is the increased effective on-chip memory bandwidth. In Figure 11(b), we note that the on-chip memory bandwidth is not equivalent to speedup, since the increase in register pressure required for format conversion causes registers to spill to on-chip memory. Spilling registers increases the total amount of data that must traverse the on-chip memory bus, so the measured effective bandwidth will exceed the speedup. As expected from the speedup of the other two fission formats, we see substantially increased on-chip memory effective bandwidth.

The ALU utilization, presented in Figure 11(c), shows the utilization of the four relevant functional units. The utilization of the functional units is normalized to the floating-point unit utilization of the configuration without fission. The single precision floating-point unit (FP32 in the figure), which is used for the core computation of the neural network, only serves as a rough proxy for performance due to fission using the floating-point unit. The integer and bit conversion functional units provide more insight into the speedup differences, representing overhead of data fission.

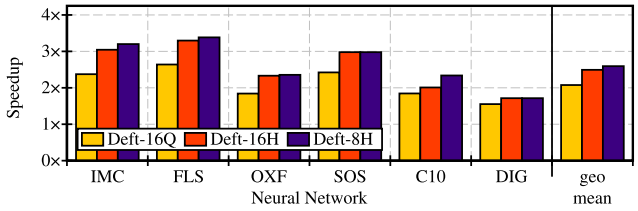

Figure 13: DeftNN runtime performance achieved by employing software-only (Deft-16Q) and hardware-accelerated (Deft-16H/8H) fission, showing substantial speedup via hardware-accelerated DeftNN.

5.5 Hardware-accelerated Data Fission

We now evaluate DeftNN with the addition of the DFU, a lightweight GPU architectural extension to accelerate near-compute data fission. We implemented and synthesized the DFU for an Nvidia Titan X (Pascal) using the ARM Artisan IBM SOI 45 nm library, showing that the DFU has an area overhead of $1.20mm^2$ (0.25% area overhead), and an active power consumption of 2.48W (0.99% power overhead).

We evaluate end-to-end performance of DeftNN atop a DFU-enabled Titan X using an in-house GPU simulation tool. This tool emulates end-to-end execution by modifying the GPU kernel to mimic the performance behavior of the modified hardware. Specifically, the fission instructions are automatically replaced by a set of instructions that have the same register dependencies, but throughput and latency characteristics matching the DFU (i.e. single cycle using single-precision floating-point ALUs).

Benefits Over Software Fission. We first evaluate the efficacy of the DFU by comparing it to software-implemented fission in isolation (i.e., no synapse vector elimination is involved). Figure 12(a) shows the speedup when the DNN computation is subjected to fission. Accelerated 16-bit fission (Deft-16H) yields a modest performance improvement over software-implemented fission (Deft-16Q), improving speedup from 1.6× to 1.8× by mitigating the overhead of performing the fission operations. The speedup of 8-bit accelerated fission (Deft-8H) is 2.3×, significantly higher than Deft-16H because the amount of data moved is dramatically reduced when using 8-bit values.

These sources of speedup are explored further in Figure 12(b) and (c). In (b), we observe comparable decreases in the effective on-chip memory bandwidth among the fission techniques. Moreover, in Figure 12(c) we observe that both of the hardware-accelerated configurations use less than half of the data conversion time compared to the software-only configuration. Although the DFU hardware for 16-bit conversion is far simpler than the hardware for 8-bit conversion, we note that it results in more total overhead due to the fact that twice as many conversions are made – only two values are produced per instruction using 16-bit conversions, rather than four values per instruction for 8-bit conversions.

End-to-end Performance. We next examine the impact of leveraging the DFU for accelerated data fission in the end-to-end DeftNN system. The speedup for all applications of the end-to-end system for Deft-16Q, Deft-16H and Deft-8H are presented in Figure 13,

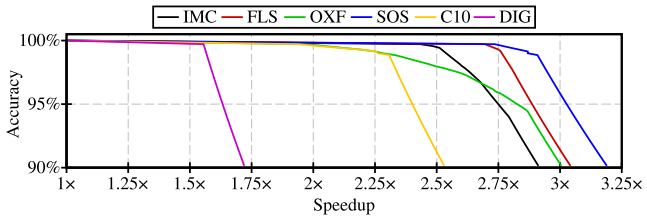

Figure 14: DeftNN Pareto frontiers, showing a range of advantageous operating points as the accuracy target is tuned.

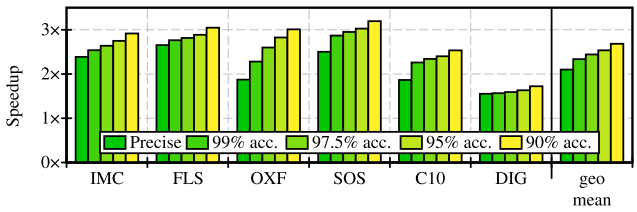

Figure 15: Speedup achieved by DeftNN at particular accuracy levels, showing that DeftNN exposes a range of useful design points for approximate computing.

which shows that the system improving performance substantially when leveraging the DFU to facilitate efficient near-compute data fission. We observe that the end-to-end speedup averages 2.1× with Deft-16Q and that it increases to 2.5× with Deft-16Q and 2.6× with Deft-8H. As Deft-16Q and Deft-16H have the same data movement characteristics, the difference between the two represents the removal of (most of) the overhead of performing data fission in software. The additional speedup achieved by Deft-8H is due to the substantial reduction in the amount of data moved compared to Deft-16Q and Deft-16H.

5.6 Performance-Accuracy Tradeoffs

In addition to removing only non-contributing synapses that do not impact accuracy, recall from §4.1.3 that synapse vector elimination parameterizations for higher performance, but slightly degraded accuracy, are also possible. The correlation parameter used for synapse vector elimination can be relaxed to allow the system to eliminate synapses that contribute in a small way to the output result. By relaxing the correlation parameter, we can be selective about the resulting DNN density and thus the amount of speedup achieved by synapse vector elimination. This section explores using this feature of synapse vector elimination within a runtime system that facilitates approximate computing, trading off small levels of accuracy for larger performance improvements.

Figure 14 presents the accuracy versus speedup Pareto frontier for each of the evaluated neural networks. As the correlation parameter is relaxed (going from left to right), the accuracy degradation is initially minimal due to the resilience of the DNN, but as more contributing synapses are removed the accuracy decreases more quickly. Beyond the precise configuration, where no loss in accuracy is permitted, which is used in the other sections of the evaluation, tuning the correlation parameter allows synapse vector elimination to achieve further speedups for small accuracy losses. We observe in Figure 15 that a spectrum of useful design points that are commonly focused on in approximate computing are available to the system [17, 37, 44], allowing DeftNN to service a wide range of use-cases where there is tolerance in end-user accuracy or a more aggressive performance target.

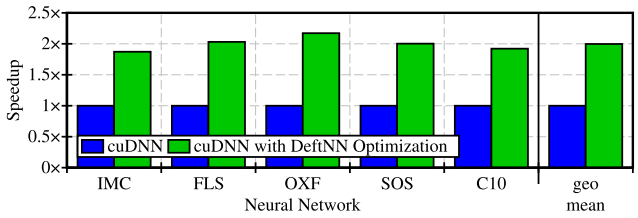

Figure 16: Speedup of cuDNN [7] with DeftNN optimizations, showing DeftNN provides similar speedup for cuDNN as it provides for MAGMA.

5.7 cuDNN with DeftNN Optimization

To demonstrate the applicability of DeftNN and its underlying optimization strategies, we apply DeftNN to cuDNN [49] by implementing the cuDNN convolution algorithm [7]. This algorithm is similar to the standard matrix multiplication algorithm, except that the preprocessing step of translating the DNN input and convolution weights into a matrix (known as `im2col`) is interleaved with the matrix multiplication. This optimization reduces off-chip memory storage requirements by lazily evaluating the contents of the input matrix. The only adjustment in synapse vector elimination to handle cuDNN is that, as the input matrix is being produced in on-chip memory, the synapse vector elimination takes place.

In Figure 16, we show the speedup achieved when DeftNN is applied to cuDNN. In applying DeftNN to cuDNN, we observe a geometric mean speedup of 2.0×, a similar speedup to what is achieved when applying DeftNN to MAGMA. Since DeftNN is similarly effective on both MAGMA and cuDNN algorithms, the fundamental GPU DNN bottlenecks being addressed by DeftNN are common to the most popular GPU DNN implementations.

5.8 Comparison to Prior Work

We next compare the novel optimizations introduced in this work and leveraged by DeftNN to prior work.

Network Pruning. Network pruning is a technique that iteratively prunes synapses from the neural network [22]. This technique reduces the number of synapses in the DNN, but it produces an irregular sparse matrix because it places no performance-aware constraints on which synapses are removed. In comparison, our synapse vector elimination technique maintains a regular dense matrix by removing entire rows or columns of synapses, allowing synapse vector elimination to map efficiently to GPU hardware.

We compare synapse vector elimination to network pruning for IMC in Figure 17, presenting network density achieved versus speedup. Network pruning [21] achieves matrix densities between

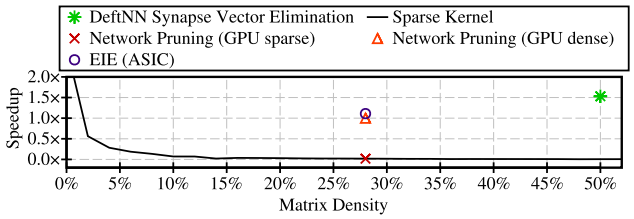

Figure 17: End-to-end speedup of DeftNN synapse vector elimination, software executed network pruning [22], and EIE [20] hardware-accelerated network pruning.

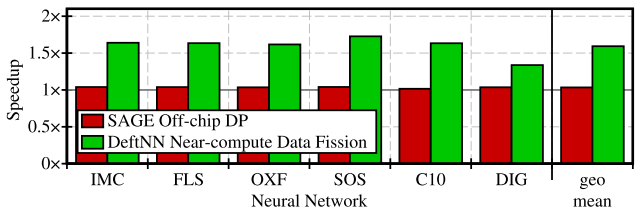

Figure 18: Comparison of DeftNN data fission to off-chip data packing [58].

9% and 100% across IMC's layers, and a weighted average of 28%. Using the network pruning approach for IMC, we execute the pruned networks using both dense and sparse kernels via cuBLAS [49] and cuSPARSE [50]. With the sparse kernel we observe that network pruning is 60× slower than the dense kernel baseline computation and 91× slower than synapse vector elimination, while with the dense kernel we observe that network pruning results in no speedup over the baseline. We performed a sweep of density levels on the sparse kernel, finding that the density must be reduced to 2.5% before the sparse kernel outperforms the dense kernel baseline. On the other hand, synapse vector elimination achieves 50% density and a 1.5× speedup on this kernel, illustrating the benefit of synapse vector elimination's architecturally-aware design.

Recent work also proposed EIE, a custom ASIC to execute network pruned DNNs [20]. While this ASIC achieves impressive results on fully connected layers, those components account for only a fraction of the end-to-end execution time in many modern DNNs, including for the IMC network. Figure 17 includes the density and speedup achieved by EIE for the end-to-end IMC network, which achieves a 1.1× speedup. Meanwhile, synapse vector elimination's speedup of 1.5× is achieved on a real GPU system.

Off-chip Data Packing. Off-chip data packing is similar to near-compute data fission, except data is packed into off-chip memory and unpacked into on-chip memory, saving off-chip memory storage and bandwidth [58]. Certain applications are able to benefit substantially from off-chip data packing, but we found that, for DNN applications, off-chip bandwidth is only slightly utilized, while on-chip bandwidth is saturated. We compare the performance improvements of off-chip and near-compute data fission in Figure 18. As expected, off-chip data packing yields modest speedups, since off-chip memory is already underutilized and the bottleneck lies elsewhere in DNN execution.

6 RELATED WORK

The computational requirements and applicability of deep neural networks [36] and convolutional neural networks [40] have prompted researchers to design novel DNN hardware [1–3, 13, 20, 32, 43, 54, 60, 62]. Some of these hardware designs specifically target memory bandwidth [4, 5]. Although these works can provide substantial speedup upon fabrication, our techniques can operate on current commodity hardware.

On the software side, there has been a lot of effort to efficiently implement DNNs on GPUs [7, 29, 34, 38, 47]. In addition to optimizing for GPUs, some work has looked at optimizing DNNs at the cluster level [9, 11, 23–25, 55?]. Further software approaches consider using different types of neural networks to improve performance [16]. Optimized algorithms can be applied in concert with our optimization techniques.

Many prior works improve performance by exploiting reduced precision [12, 30, 42, 56, 63]. Reducing precision is possible for both floating-point and fixed-point formats [10, 19]. These works all require substantial hardware modifications to operate. ACME [28], although requiring less modifications to hardware by design, still requires substantial overhead when applied to a high-throughput accelerator such as a GPU. The DFU in DeftNN requires <0.25% overhead to continuously provide the functional units with data, while scaling ACME to the same number of units would require over 19% overhead.

7 CONCLUSION

This paper describes *DeftNN*, a system for optimizing GPU-based DNNs. DeftNN uses two novel optimization techniques – synapse vector elimination, a technique that drops the non-contributing synapses in the neural network, as well as near-compute data fission, a mechanism for scaling down the data movement requirements within DNN computations. Our experimental results show that DeftNN can significantly improve DNN performance, improving performance by over 2.1× on commodity GPU hardware and over 2.6× when leveraging a small additional hardware unit that accelerates fission.

ACKNOWLEDGEMENT

We thank our anonymous reviewers for their feedback and suggestions. This work was supported by Google, Facebook, and the National Science Foundation under grants CCF-XPS-1438996, CCF-XPS-1628991, and IIS-VEC-1539011.

REFERENCES

[1] Jorge Albericio, Patrick Judd, Tayler Hetherington, Tor Aamodt, Natalie Enright Jerger, and Andreas Moshovos. 2016. CNVLUTIN: Ineffectual-Neuron-Free Deep Neural Network Computing. In *International Symposium on Computer Architecture (ISCA)*.

[2] Lukas Cavigelli, Michele Magno, and Luca Benini. 2015. Accelerating real-time embedded scene labeling with convolutional networks. In *Design Automation Conference (DAC)*.

[3] Srimat Chakradhar, Murugan Sankaradas, Venkata Jakkula, and Srihari Cadambi. 2010. A dynamically configurable coprocessor for convolutional neural networks. In *International Symposium on Computer Architecture (ISCA)*.

[4] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. 2014. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. In *Architecture Support for Programming Languages and Operating Systems (ASPLOS)*.

[5] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. 2014. Dadiannao: A machine-learning supercomputer. In *International Symposium on Microarchitecture (MICRO)*.

[6] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. 2016. Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks. In *International Symposium on Computer Architecture (ISCA)*.

[7] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cuDNN: Efficient primitives for deep learning. *arXiv:1410.0759*.

[8] Ronan Collobert, Clement Farabet, Koray Kavukcuoglu, and Soumith Chintala. 2015. torch. (2015). Retrieved August 25, 2017 from http://torch.ch/

[9] Francesco Conti and Luca Benini. 2015. A ultra-low-energy convolution engine for fast brain-inspired vision in multicore clusters. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*.

[10] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. 2014. Low precision arithmetic for deep learning. *arXiv:1412.7024*.

[11] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc extquotesingle aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, Quoc V. Le, and Andrew Y. Ng. 2012. Large scale distributed deep networks. In *Neural Information Processing Systems (NIPS)*.

[12] Zidong Du, Avinash Lingamneni, Yunji Chen, Krishna Palem, Olivier Temam, and Chengyong Wu. 2014. Leveraging the error resilience of machine-learning applications for designing highly energy efficient accelerators. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*.

[13] Clément Farabet, Berin Martini, Benoit Corda, Polina Akselrod, Eugenio Culurciello, and Yann LeCun. 2011. Neuflow: A runtime reconfigurable dataflow processor for vision. In *Computer Vision and Pattern Recognition Workshops (CVPRW)*.

[14] Klint Finley. 2015. Facebook open-sources a trove of AI tools. (2015). Retrieved August 25, 2017 from https://www.wired.com/2015/01/facebook-open-sources-trove-ai-tools/

[15] Wilson W. L. Fung, Ivan Sham, George Yuan, and Tor M. Aamodt. 2007. Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow. In *International Symposium on Microarchitecture (MICRO)*.

[16] Ross Girshick. 2015. Fast R-CNN. *arXiv:1504.08083*.

[17] Íñigo Goiri, Ricardo Bianchini, Santosh Nagarakatte, and Thu D Nguyen. 2015. ApproxHadoop: Bringing Approximations to MapReduce Frameworks. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[18] Google. 2015. TensorFlow. (2015). Retrieved August 25, 2017 from http://www.tensorflow.org/

[19] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. 2015. Deep Learning with Limited Numerical Precision. *arXiv:1502.02551*.

[20] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. 2016. EIE: efficient inference engine on compressed deep neural network. In *International Symposium on Computer Architecture (ISCA)*.

[21] Song Han, Huizi Mao, and William J Dally. 2015. Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding. *International Conference on Learning Representations (ICLR)*.

[22] Song Han, Jeff Pool, John Tran, and William Dally. 2015. Learning both Weights and Connections for Efficient Neural Network. *Neural Information Processing Systems (NIPS)*.

[23] Awni Y. Hannun, Carl Case, Jared Casper, Bryan Catanzaro, Greg Diamos, Erich Elsen, Ryan Prenger, Sanjeev Satheesh, Shubho Sengupta, Adam Coates, and Andrew Y. Ng. 2014. DeepSpeech: Scaling up end-to-end speech recognition. *arXiv:1412.5567*.

[24] Johann Hauswald, Yiping Kang, Michael A Laurenzano, Quan Chen, Cheng Li, Trevor Mudge, Ronald G Dreslinski, Jason Mars, and Lingjia Tang. 2015. DjiNN and Tonic: DNN as a service and its implications for future warehouse scale computers. In *International Symposium on Computer Architecture (ISCA)*.

[25] Johann Hauswald, Michael A. Laurenzano, Yunqi Zhang, Cheng Li, Austin Rovinski, Arjun Khurana, Ronald G. Dreslinski, Trevor Mudge, Vinicius Petrucci, Lingjia Tang, and Jason Mars. 2015. Sirius: An open end-to-end voice and vision personal assistant and its implications for future warehouse scale computers. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[26] Sunpyo Hong and Hyesoon Kim. 2009. An Analytical Model for a GPU Architecture with Memory-level and Thread-level Parallelism Awareness. In *International Symposium on Computer Architecture (ISCA)*.

[27] Intel. 2015. neon. (2015). Retrieved August 25, 2017 from https://github.com/NervanaSystems/neon

[28] Animesh Jain, Parker Hill, Shih-Chieh Lin, Muneeb Khan, Md E. Haque, Michael A. Laurenzano, Scott Mahlke, Lingjia Tang, and Jason Mars. 2016. Concise Loads and Stores: The Case for an Asymmetric Compute-Memory Architecture for Approximation. In *International Symposium on Microarchitecture (MICRO)*.

[29] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional Architecture for Fast Feature Embedding. *arXiv:1408.5093*.

[30] Patrick Judd, Jorge Albericio, Tayler Hetherington, Tor Aamodt, Natalie Enright Jerger, and Andreas Moshovos. 2016. Proteus: Exploiting Numerical Precision Variability in Deep Neural Networks. In *International Conference on Supercomputing (ICS)*.

[31] Sergey Karayev, Matthew Trentacoste, Helen Han, Aseem Agarwala, Trevor Darrell, Aaron Hertzmann, and Holger Winnemoeller. 2013. Recognizing image style. *arXiv:1311.3715*.

[32] Joo-Young Kim, Minsu Kim, Seungjin Lee, Jinwook Oh, Kwanho Kim, and Hoi-Jun Yoo. 2010. A 201.4 GOPS 496 mW real-time multi-object recognition processor with bio-inspired neural perception engine. In *Journal of Solid-State Circuits (JSSC)*.

[33] Alex Krizhevsky and Geoffrey Hinton. 2009. Learning multiple layers of features from tiny images. *Tech report, University of Toronto*.

[34] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Neural Information Processing Systems (NIPS)*.

[35] LISA lab. 2015. theano. (2015). Retrieved August 25, 2017 from http://deeplearning.net/software/theano/

[36] Hugo Larochelle, Dumitru Erhan, Aaron Courville, James Bergstra, and Yoshua Bengio. 2007. An empirical evaluation of deep architectures on problems with many factors of variation. In *International Conference on Machine learning (ICML)*.

[37] Michael A Laurenzano, Parker Hill, Mehrzad Samadi, Scott Mahlke, Jason Mars, and Lingjia Tang. 2016. Input responsiveness: using canary inputs to dynamically steer approximation. In *Programming Language Design and Implementation (PLDI)*.

[38] Andrew Lavin. 2015. maxDNN: An Efficient Convolution Kernel for Deep Learning with Maxwell GPUs. *arXiv:1501.06633*.

[39] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *Nature*.

[40] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*.

[41] Yann LeCun, Corinna Cortes, and Christopher JC Burges. 1998. The MNIST database of handwritten digits. (1998). Retrieved August 25, 2017 from http://yann.lecun.com/exdb/mnist/

[42] Yongsoon Lee, Younhee Choi, Seok-Bum Ko, and Moon Ho Lee. 2009. Performance analysis of bit-width reduced floating-point arithmetic units in FPGAs: a case study of neural network-based face detector. In *EURASIP Journal on Embedded Systems*.

[43] Boxun Li, Yuzhi Wang, Yu Wang, Yuanfeng Chen, and Huazhong Yang. 2014. Training itself: Mixed-signal training acceleration for memristor-based neural network. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*.

[44] Divya Mahajan, Amir Yazdanbakhsh, Jongse Park, Bradley Thwaites, and Hadi Esmaeilzadeh. 2016. Towards statistical guarantees in controlling quality tradeoffs for approximate acceleration. In *International Symposium on Computer Architecture (ISCA)*.

[45] Veynu Narasiman, Michael Shebanow, Chang Joo Lee, Rustam Miftakhutdinov, Onur Mutlu, and Yale N. Patt. 2011. Improving GPU Performance via Large Warps and Two-level Warp Scheduling. In *International Symposium on Microarchitecture (MICRO)*.

[46] Rajib Nath, Stanimire Tomov, and Jack Dongarra. 2010. Accelerating GPU kernels for dense linear algebra. In *High Performance Computing for Computational Science (VECPAR)*.

[47] Jiquan Ngiam, Adam Coates, Ahbik Lahiri, Bobby Prochnow, Quoc V Le, and Andrew Y Ng. 2011. On optimization methods for deep learning. In *International Conference on Machine Learning (ICML)*.

[48] M-E. Nilsback and A. Zisserman. 2008. Automated Flower Classification over a Large Number of Classes. In *Indian Conference on Computer Vision, Graphics and Image Processing (ICVGIP)*.

[49] Nvidia. 2017. cuBLAS. (2017). Retrieved August 25, 2017 from developer.nvidia.com/cublas

[50] Nvidia. 2017. cuSPARSE. (2017). Retrieved August 25, 2017 from developer.nvidia.com/cusparse

[51] Nvidia. 2017. GeForce GTX TITAN X, Specifications. (2017). Retrieved August 25, 2017 from http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-titan-x/specifications

[52] Nvidia. 2017. Parallel Thread Execution ISA Version 5.0. (2017). Retrieved August 25, 2017 from http://docs.nvidia.com/cuda/parallel-thread-execution

[53] Kalin Ovtcharov, Olatunji Ruwase, Joo-Young Kim, Jeremy Fowers, Karin Strauss, and Eric Chung. 2015. Accelerating Deep Convolutional Neural Networks Using Specialized Hardware. (2015). Retrieved August 25, 2017 from https://www.microsoft.com/en-us/research/publication/accelerating-deep-convolutional-neural-networks-using-specialized-hardware

[54] Eustace Painkras, Luis A Plana, Jim Garside, Steve Temple, Simon Davidson, Jeffrey Pepper, David Clark, Cameron Patterson, and Steve Furber. 2012. Spinnaker: a multi-core system-on-chip for massively-parallel neural net simulation. In *Custom Integrated Circuits Conference (CICC)*.

[55] Robert Preissl, Theodore M Wong, Pallab Datta, Myron Flickner, Raghavendra Singh, Steven K Esser, William P Risk, Horst D Simon, and Dharmendra S Modha. 2012. Compass: a scalable simulator for an architecture for cognitive computing. In *International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*.

[56] Brandon Reagen, Paul Whatmough, Robert Adolf, Saketh Rama, Hyunkwang Lee, Sae Kyu Lee, José Miguel Hernández-Lobato, Gu-Yeon Wei, and David Brooks. 2016. Minerva: Enabling low-power, highly-accurate deep neural network accelerators. In *International Symposium on Computer Architecture (ISCA)*.

[57] 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. 2015. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*.

[58] Mehrzad Samadi, Janghaeng Lee, D Anoushe Jamshidi, Amir Hormati, and Scott Mahlke. 2013. SAGE: Self-tuning approximation for graphics engines. In *International Symposium on Microarchitecture (ISCA)*.

[59] Kaz Sato, Cliff Young, and David Patterson. 2017. An in-depth look at Google's first Tensor Processing Unit (TPU). (2017). Retrieved August 25, 2017 from https://cloud.google.com/blog/big-data/2017/05/an-in-depth-look-at-googles-first-tensor-processing-unit-tpu

[60] Johannes Schemmel, Johannes Fieres, and Karlheinz Meier. 2008. Wafer-scale integration of analog neural networks. In *International Joint Conference on Neural Networks (IJCNN)*.

[61] Ali Shafiee, Anirban Nag, Naveen Muralimanohar, Rajeev Balasubramonian, John Paul Strachan, Miao Hu, R Stanley Williams, and Vivek Srikumar. 2016. ISAAC: A Convolutional Neural Network Accelerator with In-Situ Analog Arithmetic in Crossbars. In *International Symposium on Computer Architecture (ISCA)*.

[62] Olivier Temam. 2012. A defect-tolerant accelerator for emerging high-performance applications. In *International Symposium on Computer Architecture (ISCA)*.

[63] Vincent Vanhoucke, Andrew Senior, and Mark Z Mao. 2011. Improving the speed of neural networks on CPUs. In *Deep Learning and Unsupervised Feature Learning Workshop*.

[64] Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. 2014. How transferable are features in deep neural networks?. In *Neural Information Processing Systems (NIPS)*.

[65] Jianming Zhang, Shuga Ma, Mehrnoosh Sameki, Stan Sclaroff, Margrit Betke, Zhe Lin, Xiaohui Shen, Brian Price, and Radomír Měch. 2015. Salient Object Subitizing. In *Computer Vision and Pattern Recognition (CVPR)*.