Sparsity-Aware Caches to Accelerate Deep Neural Networks

Vinod Ganesan*, Sanchari Sen†, Pratyush Kumar*, Neel Gala‡, Kamakoti Veezhinathan* and Anand Raghunathan†
* Department of Computer Science and Engineering, IIT Madras, India
† School of Electrical and Computer Engineering, Purdue University
‡ InCore Semiconductors Pvt. Ltd
{vinodg,pratyush,kama}@cse.iitm.ac.in, {sen9,raghunathan}@purdue.edu, neelgala@incoresemi.com

***Abstract*—Deep Neural Networks (DNNs) have transformed the field of artificial intelligence and represent the state-of-the-art in many machine learning tasks. There is considerable interest in using DNNs to realize edge intelligence in highly resource-constrained devices such as wearables and IoT sensors. Unfortunately, the high computational requirements of DNNs pose a serious challenge to their deployment in these systems. Moreover, due to tight cost (and hence, area) constraints, these devices are often unable to accommodate hardware accelerators, requiring DNNs to execute on the General Purpose Processor (GPP) cores that they contain. We address this challenge through lightweight micro-architectural extensions to the memory hierarchy of GPPs that exploit a key attribute of DNNs, viz. sparsity, or the prevalence of zero values. We propose SparseCache, an enhanced cache architecture that utilizes a null cache based on a Ternary Content Addressable Memory (TCAM) to compactly store zero-valued cache lines, while storing non-zero lines in a conventional data cache. By storing address rather than values for zero-valued cache lines, SparseCache increases the effective cache capacity, thereby reducing the overall miss rate and execution time. SparseCache utilizes a Zero Detector and Approximator (ZDA) and Address Merger (AM) to perform reads and writes to the null cache. We evaluate SparseCache on four state-of-the-art DNNs programmed with the Caffe framework. SparseCache achieves 5-28% reduction in miss-rate, which translates to 5-21% reduction in execution time, with only 0.1% area and 3.8% power overhead in comparison to a low-end Intel Atom Z-series processor.**

I. Introduction

Deep Neural Networks (DNNs) have revitalized the field of Artificial Intelligence (AI) by achieving, or even surpassing, human accuracy levels in a variety of image, video, text and speech processing tasks. This has led to the widespread deployment of DNNs in a spectrum of real-world products and services. However, DNNs have very high compute and memory demands, even during inference, due to large network complexities. For example, state-of-the-art DNNs for image recognition require tens of Billions of multiply-accumulate (MAC) operations and hundreds of Mega-Bytes of memory, posing a challenge to their deployment in resource-constrained devices such as wearables and IoT sensors [1].

Prior research efforts have addressed the computational challenges of DNNs through efficient parallelization on multicores and GPUs [2], design of lower complexity and hardware friendly networks [3], approximate computing [4] and specialized hardware accelerators [5], [6]. In particular, DNN accelerators have garnered much interest, as is evident from their deployment in several recent products like Apple's A13 Bionic [7], Google's Tensor Processing Units (TPUs) [5], *etc*. However, these accelerators often prove to be prohibitively large in the context of highly resource-constrained devices such as wearables and IoT sensors, whose microcontrollers are typically limited to no more than a few mm^2 due to tight cost constraints. Therefore, many devices still execute DNNs on General-Purpose Processor (GPP) cores. For instance, 63% of Facebook's edge inference is performed on low power in-order ARM cortex A53 and A7 processors while only 23% is performed on GPUs and less than 5% on specialized accelerators [8]. Thus, optimizing DNN execution on GPPs is key to enabling the next generation of edge intelligence.

We accelerate DNNs on GPPs by developing small extensions to the memory hierarchy that take advantage of sparsity, or the presence of zero values in DNNs. Sparsity in DNNs can be primarily categorized into *static* and *dynamic* sparsity based on whether the locations of zero values remain constant or vary across different inputs. Weight sparsity, often introduced through model compression [9], is static in nature whereas activation sparsity, caused by the presence of ReLU (Rectified Linear Unit) layers, is dynamic.

Prior efforts that exploit sparsity to accelerate DNNs on GPPs largely focus on static sparsity [10], [11]. One recent effort [12] has proposed techniques to exploit both forms of sparsity in the compute pipeline of GPP cores by skipping instructions that are rendered redundant by zero operands. We observe that the opportunities engendered by sparsity extend well beyond the core pipeline. Specifically, we identify opportunities in the memory hierarchy due to sparsity, and demonstrate how to exploit them for performance improvements. We observe that when executing DNNs with sparsity, the data cache hierarchy contains a significant number of zero-valued lines (up to 60% for some DNN layers). Note that this property holds even when model compression techniques are used, since the model is invariably decompressed before execution, and additionally due to the dynamic sparsity arising from activation values. Developing low-overhead mechanisms to dynamically identify and compactly store these zero blocks can increase effective cache-capacity, reduce off-chip memory accesses and improve the efficiency of DNN execution.

We propose SparseCache, an augmented cache architecture to exploit sparsity in the memory hierarchy. It consists of a

This work was supported in part by the Indian Science and Engineering Research Board under the Overseas Visiting Doctoral Fellowship, by the Center for Computational Brain Research at IIT Madras and by C-BRIC, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program, sponsored by DARPA.

978-3-9819263-4-7/DATE20/©2020 EDAA

"null cache" for storing zero-valued cache lines, alongside a conventional data cache for storing the remaining data lines. The null cache is realized using a Ternary CAM (TCAM), and stores the addresses of zero-valued cache lines efficiently by exploiting the TCAM's innate ability to store multiple addresses in a single entry. We introduce two key micro-architectural extensions in the form of a Zero Detector and Approximator (ZDA) and a Address Merger (AM) for enabling reads and writes to SparseCache. The ZDA checks and allocates incoming zero-valued cache lines to the null cache, while also implementing controlled approximations for increasing the incidence of zeros. The AM unit dynamically merges the addresses of zero-valued cache lines with existing entries in the null cache to allow for compact storage. Finally, we enhance the replacement policy of SparseCache to preferentially retain the zero blocks in the null cache.

We evaluate SparseCache in the context of DNN inference on an in-order Intel Atom Processor and demonstrate a 5-28% reduction in cache miss-rate, which translates to 5-21% reduction in application-level execution time across a suite of 4 image-recognition benchmarks.

II. Sparsity in DNNs and Opportunities in the Memory Hierarchy

DNNs exhibit significant amounts of sparsity in their data-structures (weights and activations), which can be broadly classified into static and dynamic sparsity, based on whether the locations of zero values remain constant or vary across different inputs to the network. Static sparsity in weights mainly arises from the application of different pruning techniques [13] whereas dynamic sparsity in activations primarily arises from the presence of ReLU (Rectified Linear Units) layers, which zero-out negative values. The average levels of static and dynamic sparsity exhibited by popular DNNs for object recognition are 50% and 40% respectively [12]. In this work, we identify opportunities in the memory hierarchy of GPPs due to both forms of sparsity, and exploit them to achieve execution time benefits.

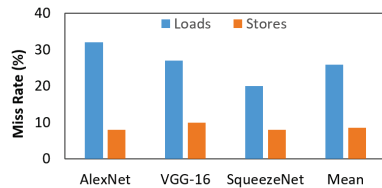

Fig. 1: Miss Rates for a 16KB Cache

In general, GPPs deployed in resource-constrained systems contain relatively small caches. Consequently, DNNs executing on these GPPs experience high miss rates and high execution times. As shown in Figure 1, the average load and store miss rates on a 16 KB, 4-way set-associative L1 data cache (representative of resource-constrained platforms) can be as high as 32% and 10% respectively, across 3 state-of-the-art image recognition DNNs, including one (SqueezeNet) that is specifically optimized for resource-constrained platforms. These miss rates are primarily caused by the large footprints of the activation and weight data structures. Addressing these high miss rates through the use of bigger caches leads to unacceptable increases in area and power. As an alternative approach, we propose low-overhead mechanisms to exploit sparsity and reduce the miss rate, thereby improving performance.

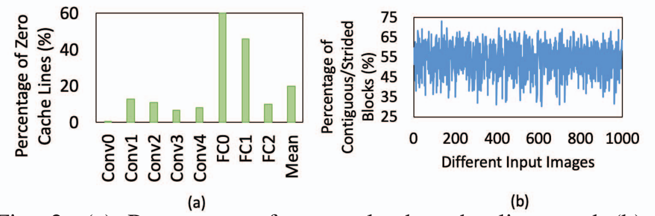

Fig. 2: (a) Percentage of zero-valued cache lines and (b) Percentage of contiguous/strided zero-valued cache lines in Deep Compressed AlexNet

Figure 2 (a) presents the average percentage of zero-valued cache lines due to the presence of zero weights and activations in the AlexNet network optimized with Deep Compression [9]. The values are averaged across 1000 different inputs on the same 16KB, 4-way set associative data cache. We observe that up to 60% of the cache lines in a layer can be zero-valued. Thus, developing a mechanism to store them compactly can free up a significant part of the cache, reducing the miss rate and execution time.

We also observe that a considerable number of these zero-valued cache lines have contiguous or power-of-two-strided addresses, *i.e.*, they are either located adjacent to another zero-valued cache line or at a distance of 2^n from another zero-valued cache line, where n is any integer between 0 and the number of address bits. Figure 2 (b) shows the fraction of such contiguous/strided zero-valued cache lines amongst all zero-valued cache lines corresponding to activations in AlexNet across 1000 different inputs. Although the exact fraction varies across different inputs, more than half of the zero-valued cache lines (average 53%) are contiguous/strided in nature. Ternary Content-Addressable Memories (TCAMs), with their ability to merge entries through appropriately placed dont-cares [14], are excellent candidates for storing the locations of these contiguous/strided zero-valued cache lines and are thus utilized in the proposed design of SparseCache.

III. SparseCache: Sparsity Aware Data Cache

In this section, we first present the key ideas behind SparseCache and explain its working within the framework of a general purpose processor. SparseCache exploits both dynamic and static sparsity in the memory hierarchy to improve the overall execution time of DNNs.

A. SparseCache: Overview

Figure 3 presents an overview of the micro-architectural extensions proposed in SparseCache. It consists of a conventional data-cache to store non-zero cache lines, alongside a TCAM-based null cache to exclusively store the addresses of zero-valued cache lines. Each entry in the null cache is capable of storing contiguous or strided address ranges of size 2^n, with n ranging from 0 to the number of address-bits.

To keep the overheads of the null cache low, we limit its size and design it to operate as a write-through cache, wherein all zero writes to the null cache are also propagated to the lower levels of the memory hierarchy. Two key micro-architectural extensions are proposed to allow reads and writes to the null cache.

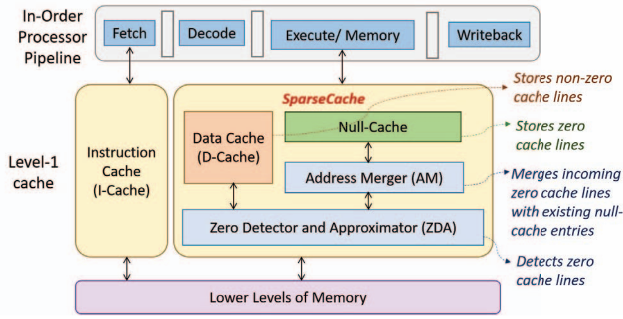

Fig. 3: SparseCache: Design Overview

1) Zero Detector and Approximator (ZDA): ZDA dynamically checks incoming cache lines to check whether they are zero-valued. Additionally, it supports an approximate mode of operation, wherein it modifies cache lines with more than a pre-specified number (α) of zero data words, to be entirely zero-valued, thereby boosting the incidence of zero-valued cache lines. The value of α is determined through profiling based on user-defined accuracy constraints.

2) Address Merger (AM): AM helps in storing contiguous/strided zero-valued cache line addresses compactly in the TCAM-based null cache. It identifies existing entries in the null cache that can be merged with an incoming address, and merges them to realize more compact storage.

We first describe how different operations are performed in SparseCache with the help of the above micro-architectural extensions. Subsequently, we describe the design of these extensions in detail.

B. SparseCache Operations

1) Reads with SparseCache: Figure 4 (a) illustrates the steps involved in a read operation in SparseCache. A read request is processed by accessing both the data cache and the null cache (in parallel, to minimize latency) and is identified to be a hit if the address is found in either of the caches. A hit in the data cache is serviced as a conventional cache hit operation, whereas a hit in the null cache is handled by simply returning zeros. Upon a miss in both the caches, the requested block is retrieved from a lower level of the memory hierarchy and passed through the ZDA to determine if it can be stored in the null cache. Zero-valued cache lines are subsequently sent to the AM unit to determine possible merges with existing entries in the null cache.

2) Writes with SparseCache: Figure 4 (b) illustrates the write operation with SparseCache. Write requests are processed differently based on the value of the written word, and the remaining values in the cache line it is written to. A non-zero valued word written to the data cache is processed as a conventional write hit. On the other hand, a zero-valued word written to the data cache is processed by reading the entire cache line and passing it through the ZDA to detect whether it can be stored in the null cache. A zero-valued word written to a line that is in the null cache does not require any operations. A non-zero word written to the null cache is processed by deleting the corresponding cache line from the null cache and transferring it to the data cache. We note that deleting a block could be expensive since it may require fragmenting a merged entry. However, non-zero writes to zero-valued addresses of activations or weights are very infrequent during DNN inference. Therefore, in our implementation, we simply delete the entire null cache entry, regardless of its size, when a non-zero word is written to any address belonging to the entry. A write miss in SparseCache is handled in a manner that is similar to a read miss.

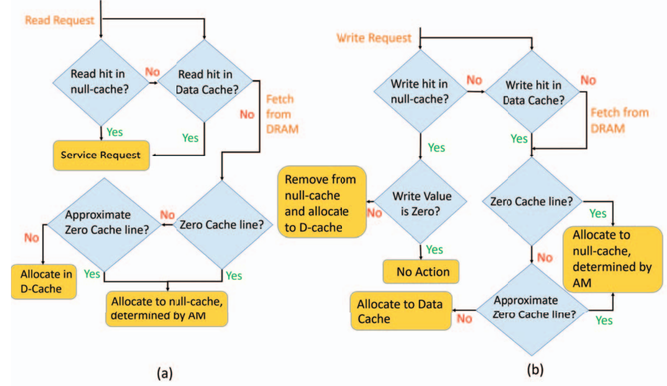

Fig. 4: (a) Reads (b) Writes with SparseCache

3) Replacements in SparseCache: Replacements in the data cache are performed by utilizing a pseudo-LRU (PLRU) policy. However, the same scheme is not sufficient for the null cache as its entries could be comprised of multiple cache lines merged into a single entry. Therefore, we propose a modified replacement policy that considers the size of the null cache entry, or the number of cache lines it represents, in addition to its recency. To that end, we maintain a state of $\log(addressbits) + 1$ bits per null cache entry; 1-bit for PLRU and $\log(addressbits)$ for capturing the size. During replacement, the least recently used candidate with the least size is chosen to be replaced.

C. SparseCache: Micro-architectural Extensions

Figure 5 details the SparseCache architecture, which can be a drop-in replacement for a conventional data cache without any modifications to the core pipeline or to software. We discuss the micro-architectural extensions of SparseCache and highlight the design choices that enable its implementation in a resource-constrained system.

1) Zero Detector and Approximator Unit (ZDA): The ZDA unit detects zeros in incoming cache lines and asserts its output, $IsZeroCacheline$, for exactly or approximately zero-valued cache lines, which can be stored in the null cache. As shown in Figure 5, each data-word in the cache line is compared to zero and the result is stored in a bit-vector with a '1' value corresponding to each zero data-word. This bit-vector is then passed through a population counter to yield the $ZeroCount$ for a particular cache line. In the exact mode of operation of the ZDA, $IsZeroCacheline$ is set if $ZeroCount$ equals the number of data words in the cache line, N. In the approximate mode of operation, the ZDA exploits the robustness of DNNs to introduce approximations in the activations and weights to increase the incidence of zero-valued cache lines. Specifically, it asserts $IsZeroCacheline$ for all cache lines with $ZeroCount$ greater than α and with non-zero data words having magnitudes less than β. The

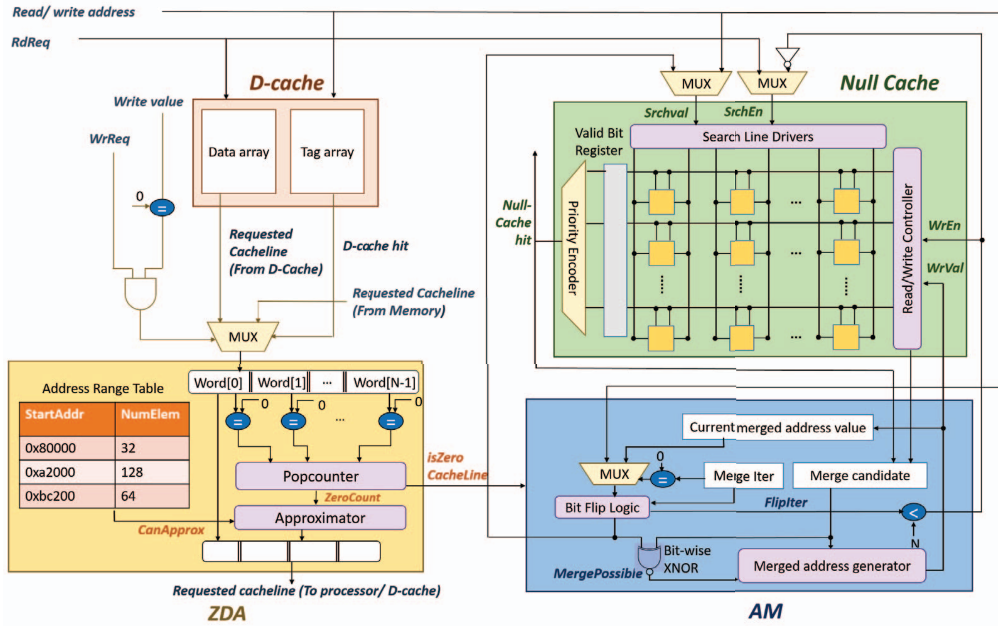

Fig. 5: SparseCache Architecture

parameters, α and β, are user-defined and may be derived by profiling the DNN with representative input data. This approximate mode of operation is restricted to specific address ranges (those that contain activations and weights) in order to ensure that sensitive data such as pointers are not affected. These address ranges are maintained in an Address Range Table (ART) within the ZDA and are written by the application program using a special instruction.

2) Address Merger (AM): The AM unit compactly stores the addresses of zero-valued cache lines in the null cache. It does so by exploiting the ability of TCAMs to store don't cares and merging incoming addresses with existing addresses in the null cache. The AM uses an iterative approach that is illustrated in Fig. 6 for a 4-bit zero-valued address of 0100. First, every bit in the address is flipped (sequentially, starting from LSB to MSB) before searching the TCAM for a possible match. A successful matched in the TCAM indicates a *Merge candidate* as the entry can be merged with the incoming address by introducing a *don't care* at the current location of the flipped bit. The corresponding merged address is generated by the *Merged address generator*. This merging can give rise to additional opportunities for merging, which are exploited by repeating the above process *Merge Iterations* times to form successive merged addresses, marked by the presence of additional Xs. In these subsequent iterations, the incoming address is replaced by the *Current merged address value* and the bit-flips are started from the merged bit of the previous iteration, while the remaining steps are unchanged. In each of these merge iterations, the previous merge candidate is invalidated in the null cache and after all iterations, the latest merged address is inserted and the corresponding replacement bits are updated.

To ensure that the overall operation of the SparseCache is not limited by the iterative search-and-merge operations performed by the AM, we interleave these operations with data access from a lower level of the memory hierarchy whenever possible. Specifically, on a cache miss, the AM unit forms the final merged address for the incoming address even before the

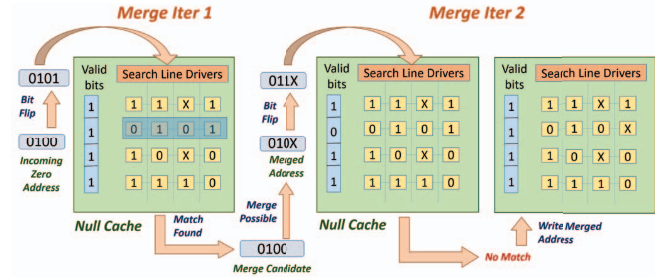

Fig. 6: Inserting addresses in the null cache

ZDA identifies the associated cache-line to be zero. In case the address is subsequently determined to be non-zero, the changes made by the AM unit are rolled back. This involves re-setting the valid bits of some of the entries to 1. The AM unit achieves this roll-back mechanism by maintaining a shadow valid bit register that stores the context of the null cache prior to the insertion operation.

IV. Experimental Methodology

In this section, we describe the methodology adopted in our experiments to evaluate the benefits of SparseCache.

A. Performance Evaluation

The performance benefits of SparseCache were evaluated with an in-order Intel Atom Z-series Processor [15] (system configurations similar to low-power Intel galileo boards [16]), using a custom x86 simulator developed with Intel's pin-tool [17]. The simulator was interfaced with the Caffe deep learning framework [18] to allow program traces from Caffe to be input to the simulator. Figure 7 presents the system configuration used in our experiments. The application-level execution time for DNN inference was measured on a system with SparseCache and compared against a baseline system utilizing a conventional data cache.

B. Area and Power overhead Evaluation

The area of the null cache was estimated from the TCAM details provided in [19], scaled to a 45nm technology and

the TCAM power consumption at 45nm for the given area was determined from the power estimation tool from [20]. The other micro-architectural extensions were implemented in Bluespec System Verilog (BSV) [21] and synthesized to NanGate's Open Cell library in 45nm technology. Their area and power consumption were measured using Synopsys Design Compiler. For our baseline implementation, we consider the core area of a representative in-order Intel Atom Z-series processor (estimated from die photographs) and its average power consumption [15].

Overall, we observe that the area and power overheads of SparseCache are minimal, amounting to just 0.1% and 3.8% of the baseline processor, respectively.

C. Benchmarks

We evaluated the benefits of SparseCache across four different image-recognition benchmarks listed in Table 7(b). The benchmarks were trained with different pruning techniques like Deep Compression [9] and Scalpel [10] to reduce the overall model size and make them suitable for deployment in resource-constrained platforms. All benchmarks demonstrate both static and dynamic sparsity. A constraint of $< 1\%$ accuracy degradation was imposed while determining the values of the parameters, α and β, used by the ZDA. We found that a TCAM size of 1KB size provided a good tradeoff between miss rate reduction *vs.* area and power overheads. Higher TCAM sizes yield diminishing returns while the area and power overheads increase.

Processor config.		Intel Atom based In-order Processor (x86)
Operating Frequency		400 MHz
SparseCache config.	L1 Data Cache	16KB, 4-way Set Associative, 32B/Block, 1 cycle hit latency
	Null-Cache	1KB, address only, TCAM, 1 cycle hit latency
Main Memory		100 cycle latency

(a)

Benchmark	Pruning Method	Dataset	No. of Layers
LeNet-5	Scalpel	MNIST	5
AlexNet	Deep-Compression	ImageNet	8
VGG-16	Deep-Compression	ImageNet	16
SqueezeNet	Deep-Compression	ImageNet	26

(b)

Fig. 7: (a) System Configuration, (b) Application Benchmarks

V. Results and Discussion

In this section, we present the results of our experiments that evaluate the benefits SparseCache.

A. Performance Improvements

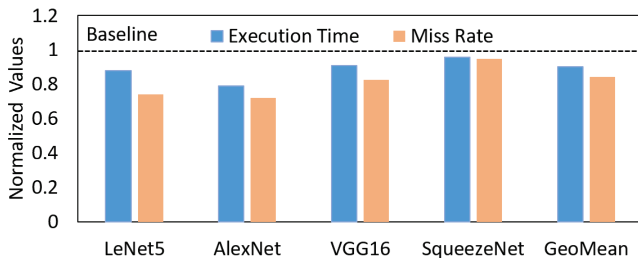

Fig. 8: Improvements in cache miss rate and application execution time

Figure 8 presents the normalized benefits of SparseCache in terms of both miss rate and execution time. We observe that SparseCache achieves 5-28% reduction in miss rate, which translates to 5-21% reduction in execution time, across the different benchmarks. Overall, the benefits are roughly proportional to the number of zero-valued cache lines. We note that miss-rate reductions will also translate into system-wide energy benefits as the number of off-chip memory accesses gets reduced.

1) Benefits per Layer: We present a layer-wise breakdown of execution time and miss rate reductions for Deep Compressed AlexNet and LeNet-5 in Figure 9. We observe that layers with higher sparsity in the network exhibit higher benefits. For instance, in AlexNet, the fully-connected layers (FC0, FC1 and FC2) achieve higher benefits due to higher levels of sparsity achieved by Deep Compression in those layers.

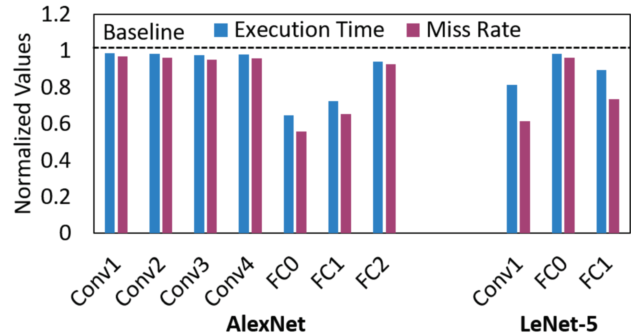

Fig. 9: Layer-wise improvement for AlexNet and LeNet-5

2) Comparison against a data-cache with 2× capacity: We compare the benefits of using the SparseCache against the benefits of using a bigger data cache, notwithstanding the fact that a 2× cache would require much higher overhead than SparseCache. We specifically consider a baseline system with 32 KB of L1 Data cache, which is double the 16KB data cache in SparseCache. As shown in Figure 10, the average execution times and miss-rates achieved by SparseCache are slightly better than this significantly higher area and power consuming baseline. For AlexNet, SparseCache provides 13% execution time and 20% miss-rate savings in comparison to the 2× larger cache.

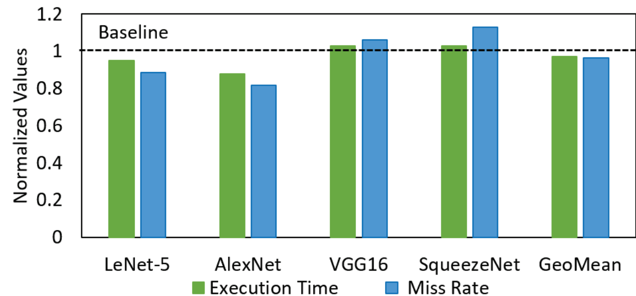

Fig. 10: Execution time and miss rate benefits over a 2× cache

B. Performance Scaling with Sparsity

We now study how the benefits of SparseCache scale with increasing levels of sparsity. To that end, we varied the sparsity in the weights and activations of the first fully-connected

layer (FC0) of AlexNet. The zeros were introduced using two different approaches. In the *Random Sparsity (RS)* approach, zeros were introduced in random locations across the layer, whereas in the *Guided Sparsity (GS)* approach, zeros were introduced in blocks of size equal to the cache-line size. As evident from Figure 11, SparseCache achieves stronger performance scaling under the GS approach due to the ability of the null cache to efficiently exploit sparsity aligned with cache lines. However, at higher sparsity levels, the difference between the two different approaches diminishes due to the prevalence of large, contiguous zero-valued memory regions in the RS approach as well.

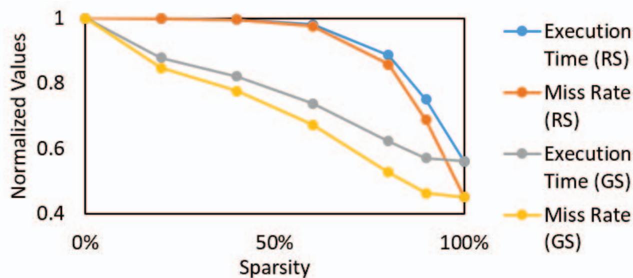

Fig. 11: Performance and miss-rate scaling with sparsity

VI. Related Work

Several prior efforts have proposed modified cache architectures to improve the execution of different workloads on GPPs by exploiting properties of the data stored. These efforts can be broadly grouped into the following three categories.

Frequent Value and Zero Value Caching have been investigated to improve cache efficiency [22]–[26]. These solutions either employ an auxiliary cache alongside the data-cache or have special ways of handling these frequent/zero values in the existing cache. However, they can exploit *only a fixed pattern of zeros, which are not exhibited by DNNs*. In contrast, *SparseCache* can exploit variable patterns, such as any power-of-2-sized contiguous or strided blocks.

Cache Compression improves effective cache capacity by compressing cache lines [27], [28]. However, it incurs high compression/decompression latency. In contrast, *SparseCache* has very low or negligible decompression latency because of its use of a very limited size TCAM.

Approximate Caches exploit the error-resiliency of applications for improvements in performance and energy [29]–[31]. This is achieved by approximately predicting on misses or by performing approximate reads and writes, improving performance and energy but not miss rate.

In contrast to these approaches, *SparseCache* increases the effective cache capacity by freeing up the L1 data cache from storing zero-valued lines, thereby providing higher performance benefits.

VII. Conclusion

As DNNs get increasingly deployed in different resource-constrained IoT edge devices and wearables, we need new approaches to speed up their execution with minimal hardware overheads. In this work, we accelerate DNN execution on GPPs by exploiting sparsity in the memory hierarchy. We propose SparseCache, an augmented cache architecture that compactly stores zero-valued cache lines in a TCAM-based null cache. Two micro-architectural extensions in the form of a Zero Detector and Approximator (ZDA) and Address Merger (AM) help in performing reads and writes to the SparseCache. We evaluate SparseCache on 4 state-of-the-art image-recognition DNNs. Our evaluations reveal that SparseCache successfully accelerates DNNs on GPPs with extremely low area and power overheads.

References

[1] S. Venkataramani, K. Roy, and A. Raghunathan, "Efficient embedded learning for iot devices," in *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 308–311, Jan 2016.

[2] A. Krizhevsky, "One weird trick for parallelizing convolutional neural networks," *arXiv preprint arXiv:1404.5997*, 2014.

[3] A. G. Howard *et al.*, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *arXiv:1704.04861*, 2017.

[4] S. Venkataramani *et al.*, "Approximate computing and the quest for computing efficiency," in *In Proc. DAC*, 2015.

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

[6] S. Venkataramani *et al.*, "Scaledeep: A scalable compute architecture for learning and evaluating deep networks," *ACM SIGARCH Computer Architecture News*, 2017.

[7] "A13 Bionic Chip brings extra hours of battery life & More: https://www.theverge.com/circuitbreaker/2019/9/10/20857177/apple-iphone-11-processor-a13-cpu-speed-graphics-specs," *theVerge*, 2019.

[8] C.-J. Wu *et al.*, "Machine learning at facebook: Understanding inference at the edge," in *Proc. HPCA*, IEEE, 2019.

[9] S. Han *et al.*, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *arXiv preprint arXiv:1510.00149*, 2015.

[10] J. Yu *et al.*, "Scalpel: Customizing DNN pruning to the underlying hardware parallelism," in *Proc. ISCA*, 2017.

[11] B. Liu *et al.*, "Sparse convolutional neural networks," in *Proc. IEEE CVPR*, 2015.

[12] S. Sen *et al.*, "SparCE: Sparsity Aware General-Purpose Core Extensions to Accelerate Deep Neural Networks," *IEEE TC*, 2018.

[13] S. Han *et al.*, "Learning both weights and connections for efficient neural network," in *NIPS 2015*.

[14] K. Pagiamtzis and A. Sheikholeslami, "Content-addressable memory (CAM) circuits and architectures: A tutorial and survey," *IEEE JSSC 2006*.

[15] T. R. Halfhill, "Intel's tiny atom," *Microprocessor Report*, vol. 22, no. 4, p. 1, 2008.

[16] M. C. Ramon, "Intel galileo and intel galileo gen 2," in *Intel® Galileo and Intel® Galileo Gen 2*, pp. 1–33, Springer, 2014.

[17] C.-K. Luk *et al.*, "Pin: building customized program analysis tools with dynamic instrumentation," in *Acm sigplan notices*, ACM, 2005.

[18] Y. Jia *et al.*, "Caffe: Convolutional architecture for fast feature embedding," in *Proc. MM*, ACM, 2014.

[19] S. Jeloka *et al.*, "A 28 nm configurable memory (TCAM/BCAM/SRAM) using push-rule 6T bit cell enabling logic-in-memory," *IEEE JSSC*, 2016.

[20] B. Agrawal and T. Sherwood, "Ternary CAM power and delay model: Extensions and uses," *IEEE TVLSI*, 2008.

[21] R. Nikhil, "Bluespec System Verilog: efficient, correct RTL from high level specifications," in *Proc. MEMOCODE*, IEEE, 2004.

[22] J. Yang *et al.*, "Frequent value compression in data caches," in *Proc. MICRO*, IEEE, 2000.

[23] Y. Zhang *et al.*, "Frequent value locality and value-centric data cache design," *ACM SIGPLAN Notices*, 2000.

[24] J. Dusser *et al.*, "Zero-content augmented caches," in *Proc. ICS*, ACM, 2009.

[25] M. M. Islam and P. Stenstrom, "Zero-value caches: Cancelling loads that return zero," in *IEEE PACT*, 2009.

[26] L. Villa *et al.*, "Dynamic zero compression for cache energy reduction," in *Proc. MICRO*, IEEE, 2000.

[27] G. Pekhimenko *et al.*, "Base-delta-immediate compression: Practical data compression for on-chip caches," in *Proc. IEEE PACT*, ACM, 2012.

[28] A. R. Alameldeen and D. A. Wood, "Adaptive cache compression for high-performance processors," *ACM ISCA*, 2004.

[29] J. S. Miguel *et al.*, "Load value approximation," in *Proc. MICRO*, 2014.

[30] J. S. Miguel *et al.*, "Doppelgänger: A cache for approximate computing," in *Proc. MICRO*, ACM, 2015.

[31] A. Ranjan *et al.*, "STAxCache: An approximate, energy efficient STT-MRAM cache," in *In Proc. DATE 2017*.