2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture

Neural Cache: Bit-Serial In-Cache Acceleration of Deep Neural Networks

Charles Eckert, Xiaowei Wang, Jingcheng Wang, Arun Subramaniyan,
Ravi Iyer†, Dennis Sylvester, David Blaauw, and Reetuparna Das
University of Michigan †Intel Corporation
{eckertch, xiaoweiw, jiwang, arunsub, dmcs, blaauw, reetudas}@umich.edu, ravishankar.iyer@intel.com

***Abstract*—This paper presents the *Neural Cache* architecture, which re-purposes cache structures to transform them into massively parallel compute units capable of running inferences for Deep Neural Networks. Techniques to do in-situ arithmetic in SRAM arrays, create efficient data mapping and reducing data movement are proposed. The *Neural Cache* architecture is capable of fully executing convolutional, fully connected, and pooling layers in-cache. The proposed architecture also supports quantization in-cache.**

Our experimental results show that the proposed architecture can improve inference latency by 18.3× over state-of-art multi-core CPU (Xeon E5), 7.7× over server class GPU (Titan Xp), for Inception v3 model. *Neural Cache* improves inference throughput by 12.4× over CPU (2.2× over GPU), while reducing power consumption by 50% over CPU (53% over GPU).

***Keywords*-Cache, In-memory architecture, Convolution Neural Network, Bit-serial architecture**

I. Introduction

In the last two decades, the number of processor cores per chip has steadily increased while memory latency has remained relatively constant. This has lead to the so-called memory wall [1] where memory bandwidth and memory energy have come to dominate computation bandwidth and energy. With the advent of data-intensive system, this problem is further exacerbated and as a result, today a large fraction of energy is spent in moving data back-and-forth between memory and compute units. At the same time, neural computing and other data intensive computing applications have emerged as increasingly popular applications domains, exposing much higher levels of data parallelism. In this paper, we exploit both these synergistic trends by *opportunistically* leveraging the huge caches present in modern processors to perform massively parallel processing for neural computing.

Traditionally, researchers have attempted to address the memory wall by building a deep memory hierarchy. Another solution is to move compute closer to memory, which is often referred to as processing-in-memory (PIM). Past PIM [2]–[4] solutions tried to move computing logic *near* DRAM by integrating DRAM with a logic die using 3D stacking [5]–[7]. This helps reduce latency and increase bandwidth, however, the functionality and design of DRAM itself remains unchanged. Also, this approach adds substantial cost to the overall system as each DRAM die needs to be augmented with a separate logic die. Integrating computation on the DRAM die itself is difficult since the DRAM process is not optimized for logic computation.

In this paper, we instead completely eliminate the line that distinguishes memory from compute units. Similar to the human brain, which does not separate these two functionalities distinctly, we perform computation directly on the bit lines of the memory itself, keeping data in-place. This eliminates data movement and hence significantly improves energy efficiency and performance. Furthermore, we take advantage of the fact that over 70% of silicon in today's processor dies simply stores and provides data retrieval; harnessing this area by re-purposing it to perform computation can lead to massively parallel processing.

The proposed approach builds on an earlier silicon test chip implementation [8] and architectural prototype [9] that shows how simple logic operations (AND/NOR) can be performed directly on the bit lines in a standard SRAM array. This is performed by enabling SRAM rows simultaneously while leaving the operands in-place in memory. This paper presents the *Neural Cache* architecture which leverages these simple logic operations to perform *arithmetic computation* (add, multiply, and reduction) *directly in the SRAM array* by storing the data in transposed form and performing bit-serial computation while incurring only an estimated 7.5% area overhead (translates to less than 2% area overhead for the processor die). Each column in an array performs a separate calculation and the thousands of memory arrays in the cache can operate concurrently.

The end result is that cache arrays morph into massive vector compute units (up to 1,146,880 bit-serial ALU slots in a Xeon E5 cache) that are one to two orders of magnitude larger than modern graphics processor's (GPU's) aggregate vector width. By avoiding data movement in and out of memory arrays, we naturally save vast amounts of energy that is typically spent in shuffling data between compute units and on-chip memory units in modern processors.

Neural Cache leverages opportunistic in-cache computing resources for accelerating Deep Neural Networks (DNNs). There are two key challenges to harness a cache's computing resources. First, all the operands participating in an in-situ operation must share bit-lines and be mapped to the same memory array. Second, intrinsic data parallel operations in DNNs have to be exposed to the underlying parallel hardware and cache geometry. We propose a data layout and execution model that solves these challenges, and harnesses the full potential of in-cache compute capabilities. Further, we find that thousands of in-cache compute units can be utilized by replicating data and improving data reuse.

2575-713X/18/$31.00 ©2018 IEEE
DOI 10.1109/ISCA.2018.00040

Techniques for low-cost replication, reducing data movement overhead, and improving data reuse are discussed.

In summary, this paper offers the following contributions:

- This is the first work to demonstrate in-situ arithmetic compute capabilities for caches. We present a compute SRAM array design which is capable of performing additions and multiplications. A critical challenge for enabling complex operations in cache is facilitating interaction between bit lines. We propose a novel bit-serial implementation with transposed data layout to address the above challenge. We designed an 8T SRAM-based hardware transpose unit for dynamically transposing data on the fly.
- Compute capabilities transform caches to massively data-parallel co-processors at negligible area cost. For example, we can re-purpose the 35 MB Last Level Cache (LLC) in the server class Intel Xeon E5 processor to support 1,146,880 bit-serial ALU slots. Furthermore, in-situ computation in caches naturally saves the on-chip data movement overheads.
- We present the *Neural Cache* architecture which re-purposes last-level cache to accelerate DNN inferences. A key challenge is exposing the parallelism in DNN computation to the underlying cache geometry. We propose a data layout that solves these challenges, and harnesses the full potential of in-cache compute capabilities. Further, techniques which reduce data movement overheads are discussed.
- The *Neural Cache* architecture is capable of fully executing convolutional, fully connected, and pooling layers in-cache. The proposed architecture also supports quantization and normalization in-cache. Our experimental results show that the proposed architecture can improve inference latency by 18.3× over state-of-the-art multi-core CPU (Xeon E5), 7.7× over server class GPU (Titan Xp), for the Inception v3 model. *Neural Cache* improves inference throughput by 12.4× over CPU (2.2× over GPU), while reducing power consumption by 59% over CPU (61% over GPU).

II. Background

A. Deep Neural Networks

Deep Neural Networks (DNN) have emerged as popular machine learning tools. Of particular fame are Convolutional Neural Networks (CNN) which have been used for various visual recognition tasks ranging from object recognition and detection to scene understanding. While *Neural Cache* can accelerate the broader class of DNNs, this paper focuses on CNNs.

CNNs consist of multiple convolution layers with additional pooling, normalization, and fully connected layers mixed in. Since convolutional layers are known to account for over 90% of inference operations [10], [11], we discuss them in this section. A convolutional layer performs a convolution on the filter and the input image. Filters have four dimensions, a height (R), width (S), channels (C), and batches of 3D filters (M). Inputs have three dimensions with a height (H), width (W), and channel (C). The filter is overlaid on the inputs and each pixel of the input is multiplied by

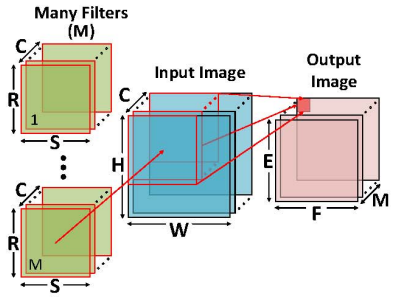

Figure 1: Computation of a convolution layer.

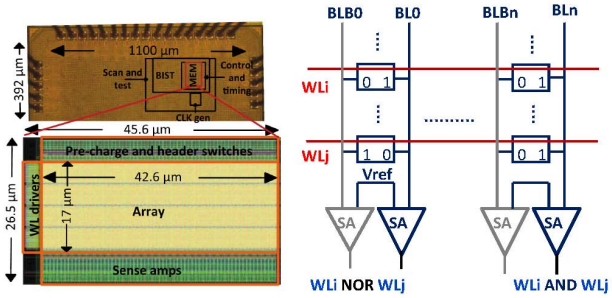

Figure 2: (a) Prototype test chip [8] (b) SRAM circuit for in-place operations. Two rows (WL_i and WL_j) are simultaneously activated. An *AND* operation is performed by sensing bit-line (BL). A *NOR* operation is performed by sensing bit-line complement (BLB).

the corresponding filter pixel and repeated across the M dimension. The results across the $R \times S$, i.e. the height and width, are accumulated together. Further, the channels are also reduced into a single element. Thus each window gives us an output of $1 \times 1 \times M$. The window is slid using a stride (U), increasing the stride will decrease the number of output elements computed. Eventually the sliding window will produce an output image with dimensions based on the height and width of the input, and the stride. The output's channel size is equivalent to the M dimension of the filter. The output image, after an activation function that varries across networks and layers, is fed as the input to the next layer.

The convolutions performed can be broken down to many different dimensions of matrix-vector multiplications. *Neural Cache* breaks down a convolution to vector-vector dot product (vector dimension $R \times S$), followed by reduction across channels (C). The different filter batches (M) are computed in parallel using the above step. A new input vector is used for each stride. In this paper we analyze the state-of-art Inception v3 model which has 94 convolutional sub-layers. *Neural Cache* is utilized to accelerate not only convolutional layers, but also pooling and fully connected layers.

B. Bit-Line Computing

Neural Cache builds on earlier work on SRAM bit line circuit technology [8], [12], [13] shown in Figure 2b. To compute in-place, two word-lines are activated

simultaneously. Computation (`and` and `nor`) on the data stored in the activated word-lines is performed in the analog domain by sensing the shared bit-lines. Compute cache [9] uses this basic circuit framework along with extensions to support additional operations: `copy`, bulk zeroing, `xor`, equality comparison, and search.

Data corruption due to multi-row access is prevented by lowering the word-line voltage to bias against the write of the SRAM array. Measurements across 20 fabricated $28nm$ test chips (Figure 2a) demonstrate that data corruption does not occur even when 64 word-lines are simultaneously activated during such an in-place computation. Compute cache however only needs two. Monte Carlo simulations also show a stability of more than six sigma robustness, which is considered industry standard for robustness against process variations. The robustness comes at the the cost of increase in delay during compute operations. But, they have no effect on conventional array read/write accesses. The increased delay is more than compensated by massive parallelism exploited by *Neural Cache*.

C. Cache Geometry

We provide a brief overview of a cache's geometry in a modern processor. Figure 3 illustrates a multi-core processor modeled loosely after Intel's Xeon processors [14], [15]. Shared Last Level Cache (LLC) is distributed into many slices (14 for Xeon E5 we modeled), which are accessible to the cores through a shared ring interconnect (not shown in figure). Figure 3 (b) shows a slice of the LLC. The slice has 80 32KB banks organized into 20 ways. Each bank is connected by two 16KB sub-arrays. Figure 3 (c) shows the internal organization of one 16KB sub-array, composed of 8KB SRAM arrays. Figure 3 (d) shows one 8KB SRAM array. A SRAM array is organized into multiple rows of data-storing bit-cells. Bit-cells in the same row share one word line, whereas bit-cells in the same column share one pair of bit lines.

Our proposal is to perform in-situ vector arithmetic operations within the SRAM arrays (Figure 3 (d)). The resulting architecture can have massive parallelism by repurposing thousands of SRAM arrays (4480 arrays in Xeon E5) into vector computational units.

We observe that LLC access latency is dominated by wire delays inside a cache slice, accessing upper-level cache control structures, and network-on-chip. Thus, while a typical LLC access can take $\sim$30 cycles, an SRAM array access is only 1 cycle (at 4 GHz clock [14]). Fortunately, in-situ architectures such as *Neural Cache* require only SRAM array accesses and do not incur the overheads of a traditional cache access. Thus, vast amounts of energy and time spent on wires and higher-levels of memory hierarchy can be saved.

III. Neural Cache Arithmetic

Compute cache [9] supported several simple operations (logical and copy). These operations are bit-parallel and do not require interaction between bit lines. *Neural Cache* requires support for more *complex operations (addition, multiplication, reduction)*. The critical challenge in supporting these complex computing primitives is facilitating interaction between bit lines. Consider supporting an addition operation which requires carry propagation between bit lines. We propose **bit-serial implementation** with **transposed data layout** to address the above challenge.

A. Bit-Serial Arithmetic

Bit-serial computing architectures have been widely used for digital signal processing [16], [17] because of their ability to provide massive bit-level parallelism at low area costs. The key idea is to process one bit of multiple data elements every cycle. This model is particularly useful in scenarios where the same operation is applied to the same bit of multiple data elements. Consider the following example to compute the element-wise sum of two arrays with 512 32-bit elements. A conventional processor would process these arrays element-by-element taking 512 steps to complete the operation. A bit-serial processor, on the other hand, would complete the operation in 32 steps as it processes the arrays *bit-slice by bit-slice instead of element-by-element*. Note that a bit-slice is composed of bits from the same bit position, but corresponding to different elements of the array. Since the number of elements in arrays is typically much greater than the bit-precision for each element stored in them, bit-serial computing architectures can provide much higher throughput than bit-parallel arithmetic. Note also that bit-serial operation allows for flexible operand bit-width, which can be advantageous in DNNs where the required bit width can vary from layer to layer.

Note that although bit-serial computation is expected to have higher latency per operation, it is expected to have significantly larger throughput, which compensates for higher operation latency. For example, the 8KB SRAM array is composed of 256 word lines and 256 bit lines and can operate at a maximum frequency of 4 GHz for accessing data [14], [15]. Up to 256 elements can be processed in parallel in a single array. A 2.5 MB LLC slice has 320 8KB arrays as shown in Figure 3. Haswell server processor's 35 MB LLC can accommodate 4480 such 8KB arrays. Thus up to 1,146,880 elements can be processed in parallel, while operating at frequency of 2.5 GHz when computing. By repurposing memory arrays, we gain the above throughput for near-zero cost. Our circuit analysis estimates an area overhead of additional bit line peripheral logic to be 7.5% for each 8KB array. This translates to less than 2% area overhead for the processor die.

B. Addition

In conventional architectures, arrays are generated, stored, accessed, and processed element-by-element in the vertical direction along the bit lines. We refer to this data layout as the bit-parallel or regular data layout. Bit-serial computing in SRAM arrays can be realized by storing data elements in a transpose data layout. Transposing ensures that all bits of a data element are mapped to the same bit line, thereby obviating the necessity for communication between bit lines. Section III-F discusses techniques to store data in a transpose layout. Figure 4 shows an example 12×4 SRAM array with transpose layout. The array stores two vectors

Figure 3: Neural Cache overview. (a) Multi-core processor with 8-24 LLC slices. (b) Cache geometry of a single 2.5MB LLC cache slice with 80 32KB banks. Each 32KB bank has two 16KB sub-arrays. (c) One 16KB sub-array composed of two 8KB SRAM arrays. (d) One 8KB SRAM array re-purposed to store data and do bit line computation on operand stored in rows (A and B). (e) Peripherals of the SRAM array to support computation.

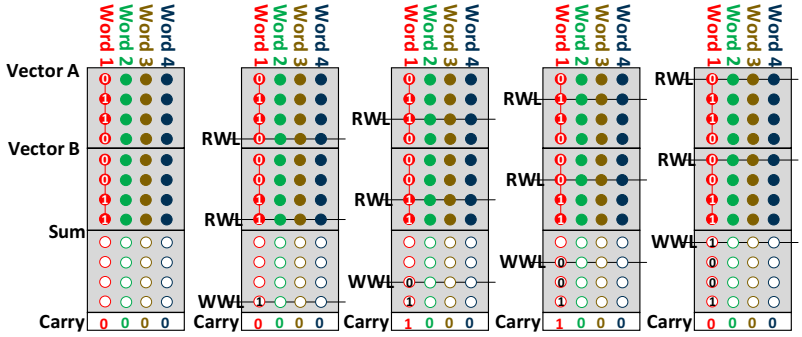

Figure 4: Addition operation

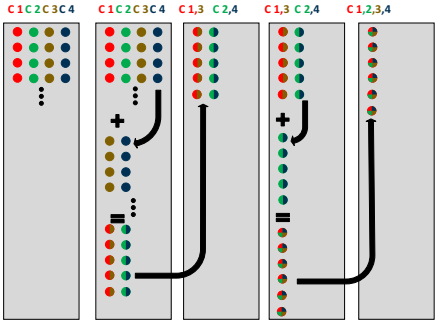

Figure 5: Reduction Operation

A and B, each with four 4-bit elements. Four word lines are necessary to store all bit-slices of 4-bit elements.

We use the addition of two vectors of 4-bit numbers to explain how addition works in the SRAM. The 2 words that are going to be added together have to be put in the same bit line. The vectors A and B should be aligned in the array like Figure 4. Vector A occupies the first 4 rows of the SRAM array and vector B the next 4 rows. Another 4 empty rows of storage are reserved for the results. There is a row of latches inside the column peripheral for the carry storage. The addition algorithm is carried out bit-by-bit starting from the least significant bit (LSB) of the two words. There are two phases in a single operation cycle. In the first half of the cycle, two read word lines (RWL) are activated to simultaneously sense and wire-and the value in cells on the same bit line. To prevent the value in the bit cell from being disturbed by the sensing phase, the RWL voltage should be lower than the normal VDD. The sense amps and logic gates in the column peripheral (Section III-E) use the 2 bit cells as operands and carry latch as carry-in to generate sum and carry-out. In the second half of the cycle, a write word line (WWL) is activated to store back the sum bit. The carry-out bit overwrites the data in the carry latch and becomes the carry-in of the next cycle. As demonstrated in Figure 4, in cycles 2, 3, and 4, we repeat the first cycle to add the second, third, and fourth bit respectively. Addition takes n + 1, to complete with the additional cycle to write a carry at the end.

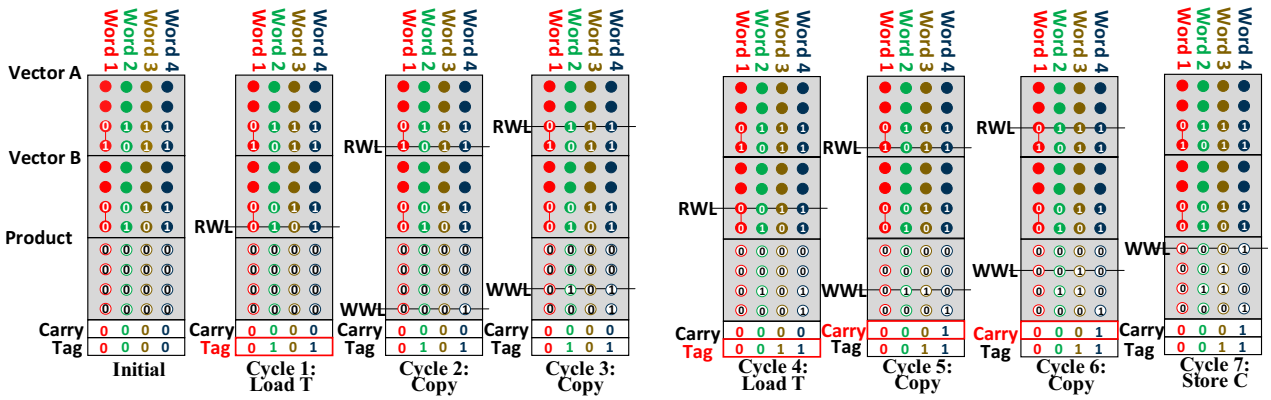

Figure 6: Multiplication Operation, each step is shown with four different sets of operands

C. Multiplication

We demonstrate how bit-serial multiplication is achieved based on addition and predication using the example of a 2-bit multiplication. In addition to the carry latch, an extra row of storage, the tag latch, is added to bottom of the array. The tag bit is used as an enable signal to the bit line driver. When the tag is one, the addition result sum will be written back to the array. If the tag is zero, the data in the array will remain. Two vectors of 2-bit numbers, A and B, are stored in the transposed fashion and aligned as shown in Figure 6. Another 4 empty rows are reserved for the product and initialized to zero. Suppose A is a vector of multiplicands and B is a vector of multipliers. First, we load the LSB of the multiplier to the tag latch. If the tag equals one, the multiplicand in that bit line will be copied to product in the next two cycles, as if it is added to the partial product. Next, we load the second bit of the multiplier to the tag. If tag equals 1, the multiplicand in that bit line will be added to the second and third bit of the product in the next two cycles, as if a shifted version of A is added to the partial product. Finally, the data in the carry latch is stored to the most significant bit of the product. Including the initialization steps, it takes n^2+5n-2 cycles to finish an n-bit multiplication. Division can be supported using a similar algorithm and takes $1.5n^2+5.5n$ cycles.

D. Reduction

Reduction is a common operation for DNNs. Reducing the elements stored on different bit lines to one sum can be performed with a series of word line moves and additions. Figure 5 shows an example that reduces 4 words, $C1$, $C2$, $C3$, and $C4$. First words $C3$ and $C4$ are moved below $C1$ and $C2$ to different word lines. This is followed by addition. Another set of move and addition reduces the four elements to one word. Each reduction step increases the number of word lines to move as we increase the bits for the partial sum. The number of reduction steps needed is $\log_2$ of the words to be reduced. In column multiplexed SRAM arrays, moves between word lines can be sped up using sense-amp cycling techniques [18].

When the elements to be reduced do not fit in the same SRAM array, reductions must be performed across arrays which can be accomplished by inter-array moves. In DNNs, reduction is typically performed across channels. In the model we examined, our optimized data mapping is able to fit all channels in the space of two arrays which sharing sense amps. We employ a technique called packing that allows us to reduce the number of channels in large layers (Section IV-A).

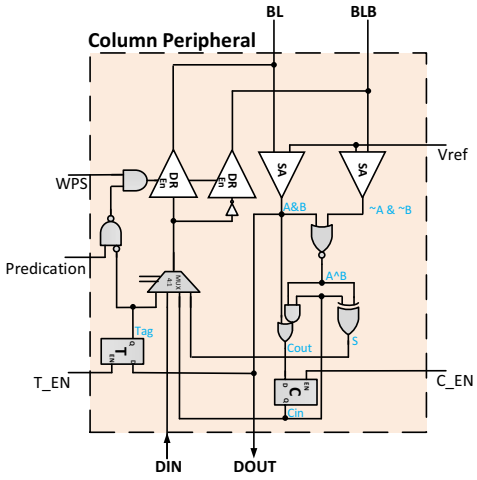

Figure 7: Bit line peripheral design

E. SRAM Array Peripherals

The bit-line peripherals are shown in Figure 7. Two single-ended sense amps sense the wire-and result from two cells, A and B, in the same bitline. The sense amp in BL gives result of $A \ \& \ B$, while the sense amp in BLB gives result of $A' \ \& \ B'$. The sense amps can use reconfigurable sense amplifier design [12], which can combine into a large differential SA for speed in normal SRAM mode and separate into two single-ended SA for area efficiency in computation mode. Through a NOR gate, we can get $A \oplus B$ which is then used to generate the sum $(A \oplus B \oplus C_{in})$ and Carry $((A \,\&\, B)$ + $(A \oplus B \,\&\, C_{in}))$. As described in the previous sections, C

and T are latches used to store carry and tag bit. A 4-to-1 mux selects the data to be written back among Sum, $Carry_{out}$, $Data_{in}$, and Tag. The Tag bit is used as the enable signal for the bit line driver to decide whether to write back or not.

F. Transpose Gateway Units

The *transpose* data layout can be realized in the following ways. First, leverage programmer support to store and access data in the *transpose* format. This option is useful when the data to be operated on does not change at runtime. We utilize this for filter weights in neural networks. However, this approach increases software complexity by requiring programmers to reason about a new data format and cache geometry.

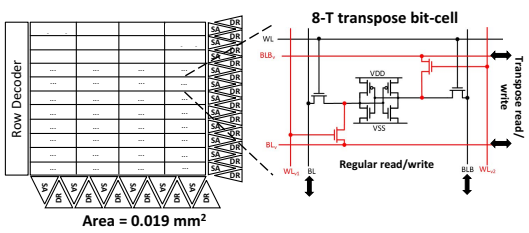

Figure 8: Transpose Memory Unit (TMU)

Second, design a few hardware *transpose memory units (TMUs)* placed in the cache control box (C-BOX in Figure 3 (b)). A TMU takes data in the *bit-parallel or regular* layout and converts it to the *transpose* layout before storing into SRAM arrays or vice-versa while reading from SRAM arrays. The second option is attractive because it supports dynamic changes to data. TMUs can be built out of SRAM arrays with multi-dimensional access (i.e., access data in both horizontal and vertical direction). Figure 8 shows a possible TMU design using an 8T SRAM array with sense-amps in both horizontal and vertical directions. Compared to a baseline 6T SRAM, the transposable SRAM requires a larger bitcell to enable read/write in both directions. Note that only a few TMUs are needed to saturate the available interconnect bandwidth between cache arrays. In essence, the transpose unit serves as a gateway to enable bit-serial computation in caches.

IV. Neural Cache Architecture

The *Neural Cache* architecture transforms SRAM arrays in LLC to compute functional units. We describe the computation of convolution layers first, followed by other layers. Figure 9 shows the data layout and overall architecture for one cache slice, modeled after Xeon processors [14], [15]. The slice has twenty ways. The last way (way-20) is reserved to enable normal processing for CPU cores. The penultimate way (way-19) is reserved to store inputs and outputs. The remaining ways are utilized for storing filter weights and computing.

A typical DNN model consists of several layers, and each layer consists of several hundred thousands of convolutions. For example, Google's Inception v3 has 20 layers, most of which have several branches. Inception v3 has $\approx$ 0.5 million convolutions in each layer on average. *Neural Cache* computes layers and each branches within a layer serially. The convolutions within a branch are computed in parallel to the extent possible. Each of the 8KB SRAM arrays computes convolutions in parallel. The inputs are streamed in from the reserved way-19. Filter weights are stationary in compute arrays (way-1 to way-18).

Neural Cache assumes 8-bit precision and quantized inputs and filter weights. Several works [19]–[21] have shown that 8-bit precision has sufficient accuracy for DNN inference. 8-bit precision was adopted by Google's TPU [22]. Quantizing input data requires re-quantization after each layer as discussed in Section IV-D.

A. Data Layout

This section first describes the data layout of one SRAM array and execution of one convolution. Then we discuss the data layout for the whole slice and parallel convolutions across arrays and slices.

A *single convolution* consists of generating *one* of the $E \times E \times M$ output elements. This is accomplished by multiplying $R \times S \times C$ input filter weights with a same size window from the input feature map across the channels. *Neural Cache* exploits channel level parallelism in a single convolution. For each convolution, an array executes $R \times S$ Multiply and Accumulate (MAC) in parallel across channels. This is followed by a reduction step across channels.

An example layout for a single array is shown in Figure 10 (a). Every bitline in the array has 256 bits and can store 32 1-byte elements in transpose layout. Every bitline stores $R \times S$ filter weights (green dots). The channels are stored across bit lines. To perform MACs, space is reserved for accumulating partial sums (lavender dots) and for scratch pad (pink dots). Partial sums and scratch pad take 3×8 and 2×8 word lines.

Reduction requires an additional 8×8 word lines as shown in Figure 10 (b). However the scratch pad and partial sum can be overwritten for reduction as the values are no longer needed. The maximum size for reducing all partial sums is 4 bytes. So to perform reduction, we reserve two 4 byte segments. After adding the two segments, the resultant can be written over the first segment again. The second segment is then loaded with the next set of reduced data.

Each array may perform several convolutions in series, thus we reserve some extra space for output elements (red dots). The remaining word lines are used to store input elements (blue dots). It is desirable to use as many word lines as possible for inputs to maximize input reuse across convolutions. For example in a 3×3 convolution with a stride of 1, 6 of the 9 bytes are reused across each set of input loads. Storing many input elements allows us to exploit this locality and reduce input streaming time.

The filter sizes ($R \times S$) range from 1-25 bytes in Inception v3. The common case is a 3×3 filter. *Neural Cache* data mapping employs *filter splitting* for large filters. The filters are split across bitlines when their size exceeds 9 bytes. The other technique employed is *filter packing*. For 1×1

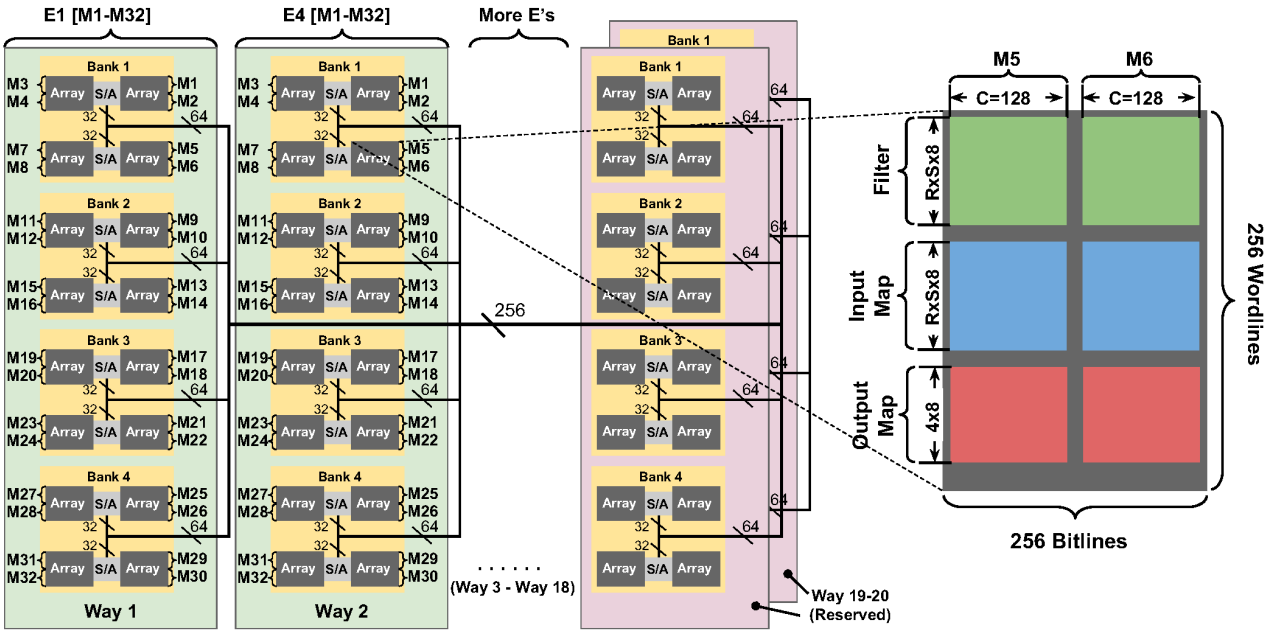

Figure 9: Neural Cache Data Layout for one LLC Cache Slice.

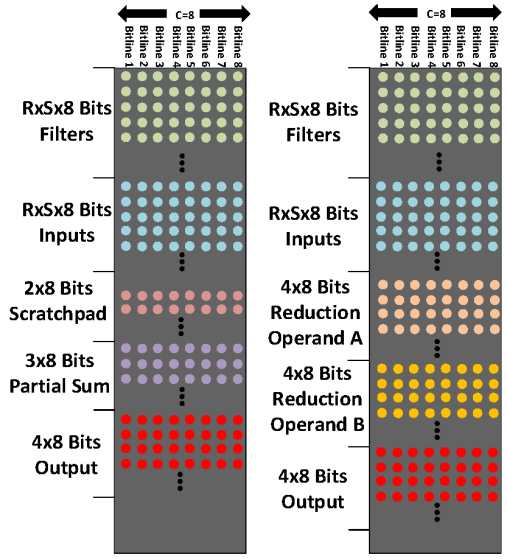

Figure 10: Array Data Layout (a) MACs (b) Reduction.

filters we compress multiple channels into the same bit line. Instead of putting a single byte of the filter, we can instead put 16 bytes of the filter. Since 1×1 has no input reuse, we only need one input byte at a time. By packing the filters, the number of reductions is decreased at no cost to input loading. More importantly, by packing all channels in the network it is guaranteed to fit within 2 arrays that share sense-amps, making reduction faster and easier.

When mapping the layers, first the new effective channel size after packing and filter splitting is calculated. This channel number is then rounded up to the nearest power of 2, by padding the extra channels with zero. Depending on parameters, filters from different output channels (M's) can be placed in the same array as well. For instance, the first layer of Inception v3 has so few channels, that all M's can be placed in the same array.

Finally, although all operations are accomplished at the bit level, each data element is stored as a multiple of a byte, although it may not necessarily require that many bits. This is done for simplicity, software programmability, and easier data movement.

B. Data Parallel Convolutions

Each layer in the network produces $M\times E\times E$ outputs, and each output requires one convolution. All the outputs can be produced in parallel, provided we have sufficient computing resources. We find that in most scenarios half an array or even a quarter of an array is sufficient to computes each output element. Thus *Neural Cache*'s massively parallel computing resources can be utilized to do convolutions in parallel.

Figure 11 shows the division of convolutions among cache slices. Each slice can be visualized as a massively parallel SIMD core. Each slice gets roughly an equal number of outputs to work on. Mapping consecutive output elements to the same slice has the nice property that the next layer can find much of its required input data locally within its reserved way. This reduces the number of inter-slice transfers needed when loading input elements across layers.

Figure 9 shows the layout of one slice for an example layer. This layer has parameters: $R\times S=3\times3$, $C=128$, $M=32$, $E=32$. Figure 9 (right) shows a single array. Each array can pack two complete filters ($R\times S\times C$). The array

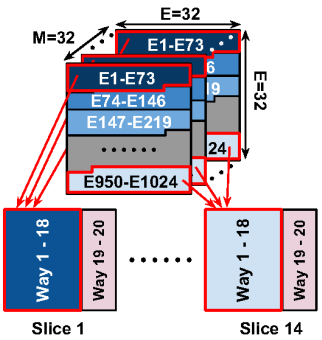

Figure 11: Partitioning of Convolutions between Slices.

shown in figure 9 packs $M=5$ and $M=6$ in the same array. Thus each array can compute two convolutions in parallel. Each way (4 banks, or 16 arrays) computes 32 convolutions in parallel ($E_i \ \forall \ M's \ M1 - M32$). The entire slice can compute 18×32 convolutions in parallel. Despite this parallelism, some convolutions need to be computed serially, when the total number of convolutions to be performed exceed the total number of convolutions across *all slices*. In this example, about 4 convolutions are executed serially.

When mapping inputs and filters to the cache, we prioritize uniformity across the cache over complete (i.e., 100%) cache compute resource utilization. This simplifies operations as all data is mapped in such a way that all arrays in the cache will be performing the exact same instruction at the same time and also simplifies data movement.

C. Orchestrating Data Movement

Data parallel convolutions require replication of filters and streaming of inputs from the reserved way. This section discusses these data-movement techniques.

We first describe the interconnect structure. Our design is based on the on-chip LLC [14] of the Intel Xeon E5-2697 processor. There are in total 14 cache slices connected by a bidirectional ring. Figure 9 shows the interconnect within a cache slice. There is a 256-bit data bus which is capable of delivering data to each of the 20 ways. The data bus is composed of 4 64-bit data buses. Each bus serves one quadrant. A quadrant consists of a 32KB bank composed of four 8KB data arrays. Two 8 KB arrays within a bank share sense-amps and receive 32 bits every bus cycle.

Filter Loading: We assume that all the filter data for a specific layer reside in DRAM before the entire computation starts. Filter data for different layers of the DNN model are loaded in serial. Within a convolution layer, regardless of the number of output pixels done in serial, the positions of filter data in the cache remain stationary. Therefore, it is sufficient to load filter data from *memory only once per layer*.

Across a layer we use M sets of $R \times S$ filters. By performing more than M convolutions, we will replicate filters across and within slices. Fortunately the inter-slice ring and intra-slice bus are both naturally capable of broadcasting, allowing for easy replication for filter weights. Each filter weight loaded from DRAM is broadcasted to all slices over the ring and all ways over the intra-slice bus.

In each array that actively performs computation, $R' \times S' \times 8$ word lines are loaded with filter data, where $R' \times S'$ is the equivalent filter dimension after packing and filter splitting, and 8 is the bit-serial factor due to 8-bits per element. *Neural Cache* assumes that filter weights are preprocessed to a transpose format and laid out in DRAM such that they map to correct bitlines and word-lines. Our experiments decode the set address and faithfully model this layout. Software transposing of weights is a one time cost and can be done cheaply using x86 SIMD shuffle and pack instructions [23], [24].

Input Data Streaming: We only load the input data of the first layer from DRAM, because for the following layers, the input data have already been computed and temporarily stored in the cache as outputs. In some layers, there are multiple output pixels to be computed in serial, and we need to stream in the corresponding input data as well, since different input data is required at any specific cache array for generating different output pixels.

Within each array that actively performs computation, $R' \times S' \times 8$ word lines of input data are streamed in, where $R' \times S'$ is the equivalent filter dimension after packing and filter splitting, and 8 is the bit-serial factor. When loading inputs from DRAM for the first layer, input data is transposed using the TMUs at C-BOX.

Since, each output pixel requires a different set of input data, input loading time can be high. We exploit duplicity in input data to reduce transfer time. For different output channels (M's) with the same output pixel position (i.e. same E_i for different M's), the input data is the same and can be replicated. Thus for scenarios were these different output channels are in different ways, the input data can be loaded in one intra-slice bus transfer. Furthermore, a large portion of input data can be reused across output pixels done in serial as discussed in Section IV-A. This helps reducing transfer time. We also observe that often the input data is replicated across arrays within a bank. We put a 64-bit latch at each bank, so that the total input transfer time can be halved.

Note, intra-bus transfers happen in parallel across different cache slices. Thus distributing E's across slices significantly reduces input loading time as well by leveraging intra-slice network bandwidth.

Output Data Management: One way of each slice (128 KB), is reserved for temporarily storing the output data. After all the convolutions for a layer are executed, data from compute arrays are transferred to the reserved way. We divide the computation into different slices according to the output pixel position. Contiguous output pixels are assigned to the same slice so that one slice will need neighboring inputs for at most $R \times E$ pixels. This design significantly reduces inter-slice communication for transferring outputs.

D. Supporting Functions

Max Pooling layers compute the maximum value of all the inputs inside a sliding window. The data mapping and

input loading would function the same way as convolution layers, except without any filters in the arrays.

Calculating the maximum value of two or more numbers can be accomplished by designating a temporary maximum value. The temporary maximum is then subtracted by the next output value and the resultant is stored in a separate set of word lines. The most significant bit of the result is used as a mask for a selective copy. The next input is then selectively copied to the maximum location based on the value of the mask. This process is repeated for the rest of the inputs in the array.

Quantization of the outputs is done by calculating the the minimum and maximum value of all the outputs in the given layer. The min can be computed using a similar set of operations described for max. For quantization, the min and max will first be calculated within each array. Initially all outputs in the array will be copied to allow performing the min and max at the same time. After the first reduction, all subsequent reductions of min/max are performed the same way as channel reductions. Since quantization needs the min and max of the entire cache, a series of bus transfers is needed to reduce min and max to one value. This is slower than in-array reductions, however unlike channel reduction, min/max reduction happens only once in a layer making the penalty small.

After calculating the min and max for the entire layer, the result is then sent to the CPU. The CPU then performs floating point operations on the min and max of the entire layer and computes two unsigned integers. These operations take too few cycles to show up in our profiling. Therefore, it is assumed to be negligible. The two unsigned integers sent back by the CPU are used for in-cache multiplications, adds, and shifts to be performed on all the output elements to finally quantize them.

Batch Normalization requires first quantizing to 32 bit unsigned. This is accomplished by multiplying all values by a scalar from the CPU and performing a shift. Afterwards scalar integers are added to each output in the corresponding output channel. These scalar integers are once again calculated in the CPU. Afterwards, the data is re-quantized as described above.

In Inception v3, **ReLU** operates by replacing any negative number with zero. We can write zero to every output element with the MSB acting as an enable for the write. Similar to max/min computations, ReLU relies on using a mask to enable selective write.

Avg Pool is mapped in the same way as max pool. All the inputs in a window are summed and then divided by the window size. Division is slower than multiplication, but the divisor is only 4 bits in Inception v3.

Fully Connected layers are converted into convolution layers in TensorFlow. Thus, we are able to treat the fully connected layer as another convolution layer.

E. Batching

We apply batching to increase the system throughput. Our experiments show that loading filter weights takes up about 46% of the total execution time. Batching multiple input images significantly amortizes the time for loading weights and therefore increases system throughput. *Neural Cache* performs batching in a straightforward way. The image batch will be processed sequentially in the layer order. For each layer, at first, the filter weights are loaded into the cache as described in Section IV-A. Then, a batch of input images are streamed into the cache and computation is performed in the same way as without batching. For the whole batch, the filter weights of the involved layer remain in the arrays, without reloading. Note that for the layers with heavy-sized outputs, after batching, the total output size may exceed the capacity of the reserved way. In this case, the output data is dumped to DRAM and then loaded again into the cache. In the Inception v3, the first five requires dumping output data to DRAM.

F. ISA support and Execution Model

Neural Cache requires supporting a few new instructions: in-cache addition, multiplication, reduction, and moves. Since, at any given time only one layer in the network is being operated on, all compute arrays execute the same in-cache compute instruction. The compute instructions are followed by move instructions for data management. The intra-slice address bus is used to broadcast the instructions to all banks. Each bank has a control FSM which orchestrates the control signals to the SRAM arrays. The area of one FSM is estimated to be 204 μm^2, across 14 slices which sums to 0.23 mm^2. Given that each bank is executing the same instruction, the control FSM can be shared across a way or even a slice. We chose not to optimize this because of the insignificant area overheads of the control FSM. *Neural Cache* computation is carried out in 1-19 ways of each slice. The remaining way (way-20) can be used by other processes/VMs executing on the CPU cores for normal background operation. Intel's Cache Allocation Technology (CAT) [25] can be leveraged to dynamically restrict the ways accessed by CPU programs to the reserved way.

V. Evaluation Methodology

Baseline Setup: For baseline, we use dual-socket Intel Xeon E5-2697 v3 as CPU, and Nvidia Titan Xp as GPU. The specifications of the baseline machine are in Table II. Note that the CPU specs are per socket. Note that the baseline CPU has the exact cache organization (35 MB L3 per socket) as we used in *Neural Cache* modeling. The benchmark program is the inference phase of the Inception v3 model [26]. We use TensorFlow as the software framework to run NN inferences on both baseline CPU and GPU. The default profiling tool of TensorFlow is used for generating execution time breakdown by network layers for both CPU and GPU. The reported baseline results are based on the unquantized version of Inception v3 model, because we observe that the 8-bit quantized version has a higher latency on the baseline CPU due to lack of optimized libraries for quantized operations (540 ms for quantized / 86 ms for unquantized). To measure execution power of the baseline, we use RAPL [27] for CPU power measurement and Nvidia-SMI [28] for GPU power measurement.

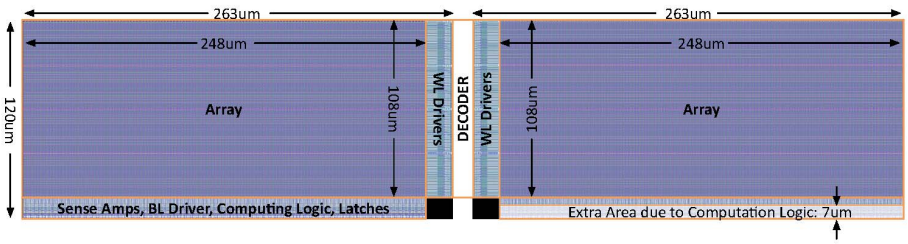

Figure 12: SRAM Array Layout.

Layer	H	$R \times S$	E	C	M	Conv	Filter Size / MB	Input Size / MB
Conv2D_1a_3x3	299	9	149	3	32	710432	0.001	0.256
Conv2D_2a_3x3	149	9	147	32	32	691488	0.009	0.678
Conv2D_2b_3x3	147	9	147	32	64	1382976	0.018	0.659
MaxPool_3a_3x3	147	9	73	0	64	0	0.000	1.319
Conv2D_3b_1x1	73	1	73	64	80	426320	0.005	0.325
Conv2D_4a_3x3	73	9	71	80	192	967872	0.132	0.407
MaxPool_5a_3x3	71	9	35	0	192	0	0.000	0.923
Mixed_5b	35	1-25	35	48-192	32-192	568400	0.243	0.897
Mixed_5c	35	1-25	35	48-256	48-256	607600	0.264	1.196
Mixed_5d	35	1-25	35	48-288	48-288	607600	0.271	1.346
Mixed_6a	35	1-9	17	64-288	64-384	334720	0.255	1.009
Mixed_6b	17	1-9	17	128-768	128-768	443904	1.234	0.847
Mixed_6c	17	1-9	17	160-768	160-768	499392	1.609	0.847
Mixed_6d	17	1-9	17	160-768	160-768	499392	1.609	0.847
Mixed_6e	17	1-9	17	192-768	192-768	499392	1.898	0.847
Mixed_7a	17	1-9	8	192-768	192-768	254720	1.617	0.635
Mixed_7b	8	1-9	8	384-1280	192-1280	208896	4.805	0.313
Mixed_7c	8	1-9	8	384-2048	192-2048	208896	5.789	0.500
AvgPool	8	64	1	0	2048	0	0.000	0.125
FullyConnected	1	1	1	2048	1001	1001	1.955	0.002

Table I: Parameters of the Layers of Inception V3.

CPU	**Intel Xeon E5-2697 v3**
Base Frequency	2.6 GHz
Cores/Threads	14/28
Process	22 nm
TDP	145 W
Cache	32 kB i-L1 per core, 32 kB d-L1 per core, 256 kB L2 per core, 35 MB shared L3
System Memory	64 GB DRAM, DDR4

GPU	**Nvidia Titan Xp**
Frequency	1.6 GHz
CUDA Cores	3840
Process	16 nm
TDP	250 W
Cache	3MB shared L2
Graphics Memory	12 GB DRAM, GDDR5X

Table II: Baseline CPU & GPU Configuration.

Modeling of Neural Cache: To estimate the power and delay of the SRAM array, the SPICE model of an 8KB computational SRAM is simulated using the 28 nm technology node. The nominal voltage for this technology is 0.9V. Since we activate two read word lines (RWL) at the same time in computation, we reduce the RWL voltage to improve bit cell read stability. But lowering RWL voltage will increase the read delay. We simulated for various RWL voltages and to achieve industry standard 6 sigma margin, we choose 0.66V as the RWL voltage. The total computing time is 1022ps. The delay of a normal SRAM read is 654ps given by the standard foundry memory compiler. So the computation SRAM delay is about 1.6× larger than normal SRAM read. Xeon processor cache arrays can operate at 4 GHz [14], [15]. We conservatively choose a frequency of 2.5 GHz for *Neural Cache* in the compute mode. Our SPICE simulations also provided the total energy consumption in one clock cycle for reading out 256 bits in SRAM mode or operating on 256 bit lines in computation mode. This was estimated to be 13.9pJ for SRAM access cycles and 25.7pJ for compute cycles. Since we model *Neural Cache* for the Intel Xeon E5-2697 v3 processor at 22 nm, the energy was scaled down to 8.6pJ for SRAM access cycles and 15.4pJ for compute cycles. The SRAM array layout is shown in Figure 12. Compute capabilities incur an area overhead of 7.5% increase due to extra logic and an extra decoder.

For the in-cache computation, we developed a cycle-accurate simulator based on the deterministic computation model discussed in Section IV. The simulator is verified by running data traces on it and matching the results with traces obtained from instrumenting the TensorFlow model for Inception v3. To model the time of data loading, we write a C micro-benchmark, which sequentially reads out the exact sets within a way that need loading with data. The set decoding was reverse engineered based on Intel's last level cache architecture [14], [15]. We conduct this measurement for all the layers according to their different requirement of sets to

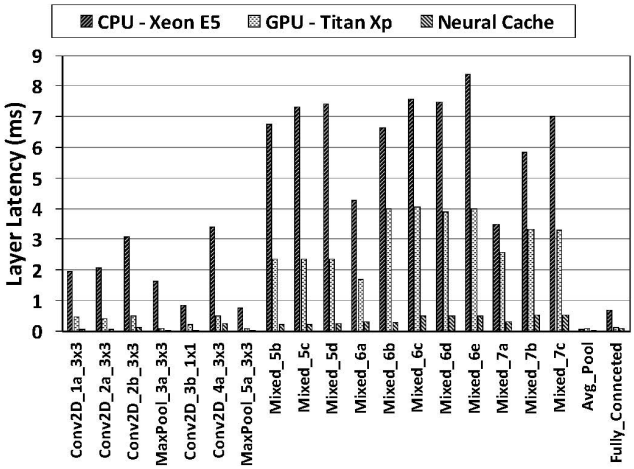

Figure 13: Inference latency by Layer of Inception v3.

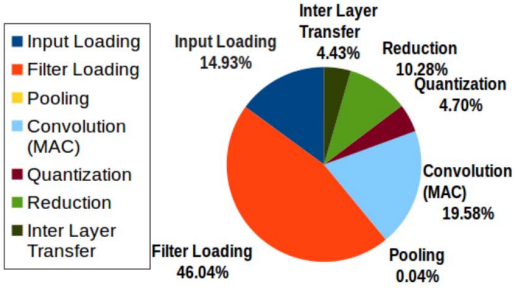

Figure 14: Inference Latency Breakdown.

be loaded. Then, the measured time is multiplied with a factor that accounts for the sequential data transfer across different ways, as well as the sequential computation of different output pixels within a layer. Note that the measured time includes the data loading from DRAM to on-chip cache, but for input data loading, the data is already in the cache (except the first layer). For more accurate modeling, the micro-benchmark is profiled by the VTune Amplifier [29] for estimating the percentage of DRAM bound cycles, and such DRAM-loading time is excluded from the input data loading time.

VI. Results

A. Latency

Figure 13 reports the latency of all layers in the Inception v3 model. A majority of time is spent on the mixed layers for both CPU and GPU, while *Neural Cache* achieves significantly better latency than baseline for all layers. This is primarily because *Neural Cache* exploits the available data parallelism across convolutions to provide low-latency inference. *Neural Cache*'s data mapping not only makes the computation entirely data independent, but the operations performed are identical, allowing for SIMD instructions.

Consider an example layer, Conv2D_Layer_2b_3×3. This layer computes $\approx 1.4\ million$ convolutions, out of which *Neural Cache* executes $\approx 32\ thousand$ convolutions in parallel and 43 in series. The compute cache arrays show 99.7% utilization for this layer during convolutions(after data loading). Each convolution takes 2784 cycles (236 cycles/MAC× 9 + 660 reduction cycles). Then reduction takes a further 660 cycles. The whole layer takes 117912 cycles (43 convolution in series × 2784 cycles), taking 0.0479 ms to finish the convolutions for *Neural Cache* running at 2.5 GHz. Remaining time for the layer is attributed to data loading. CPU and GPU cannot take advantage of data parallelism on this scale due to lack of sufficient compute resources and on-chip data-movement bottlenecks.

Figure 14 shows the breakdown of *Neural Cache* latency. Loading filter weights into cache and distributing them into arrays takes up 46% of total execution time, and 15% of the time is spent on streaming in the input data to the appropriate position within the cache. Transferring output data to the proper reserved space takes 4% of the time. Filter loading is the most time consuming part since data is loaded from DRAM. The remaining parts are for computation, including multiplication in convolution (MACs) (20%), reduction (10%), quantization (5%), and pooling (0.04%).

Figure 15 shows the total latency for *Neural Cache*, as well as the baseline CPU and GPU, running inference on the Inception v3 model. *Neural Cache* achieves a 7.7× speedup in latency compared to baseline GPU, and 18.3× speedup on baseline CPU. The significant speedup can be attributed to the elimination of high overheads of on-chip data movement from cache to the core registers, and data parallel convolutions.

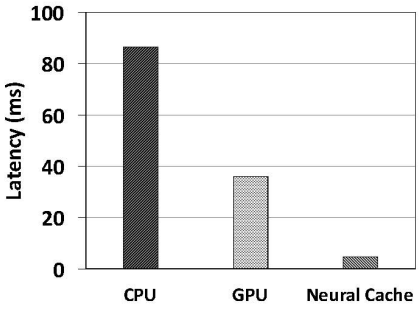

Figure 15: Total Latency on Inception v3 Inference.

B. Batching

Figure 16 shows the system throughput in number of inferences per second as the batch size varies. *Neural Cache* outperforms the maximum throughput of baseline CPU and GPU even without batching. This is mainly due to the low latency of *Neural Cache*. Another reason is that *Neural Cache* scales linearly with the number of host CPUs, and thus the throughput of *Neural Cache* doubles with a dual-socket node.

As the batch size N increases from 1 to 16, the throughput of *Neural Cache* increases steadily. This can be attributed to the amortization of filter loading time. For even higher batch sizes, the effect of such amortization diminishes and therefore the throughput plateaus. As shown in the figure, the GPU throughput also plateaus after batch size exceeds 64. At the highest batch size, *Neural Cache* achieves a

throughput of 604 inferences/sec, which is equivalent to 2.2× GPU throughput, or 12.4× CPU throughput.

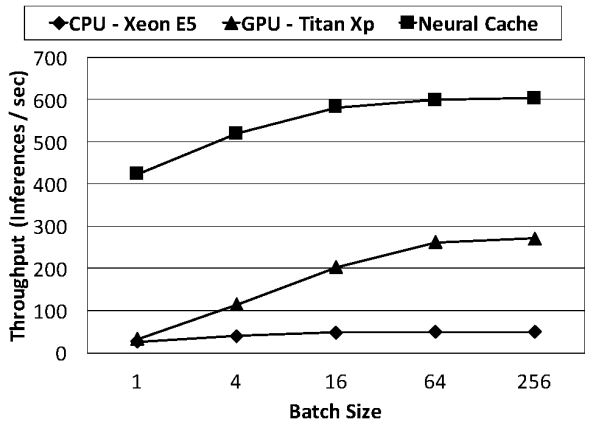

Figure 16: Throughput with Varying Batching Sizes.

C. Power and Energy

This section discusses the energy and power consumption of *Neural Cache*. Table III summarizes the energy/power comparison with baseline. *Neural Cache* achieves an energy efficiency that is 16.6× better than the baseline GPU and 37.1× better than CPU. The energy efficiency improvement can be explained by the reduction in data movement, the increased instruction efficiency of SIMD-like architecture, and the optimized in-cache arithmetic units.

The average power of *Neural Cache* is 53.11% and 49.87% lower than the GPU and CPU baselines respectively. Thus, *Neural Cache* does not form a thermal challenge for the servers. *Neural Cache* also outperforms both CPU and GPU baselines in terms of energy-delay product (EDP), since it consumes less energy and has a shorter latency.

	CPU	GPU	Neural Cache
Total Energy / J	9.137	4.087	0.246
Average Power / W	105.56	112.87	52.92

Table III: Energy Consumption and Average Power.

Cache Capacity	35MB	45MB	60MB
Inference Latency	4.72 ms	4.12 ms	3.79 ms

Table IV: Scaling with Cache Capacity (Batch Size=1).

D. Scaling with Cache Capacity

In this section we explore how increasing the cache capacity affects the performance of *Neural Cache*. We increase the cache size from 35MB (14 slices) to 45MB (18 slices), and 60MB (24 slices). Increasing the number of slices speedups most aspects of the inference. The total number of arrays which compute increases, thereby increasing convolutions that can be done in parallel. This reduces the convolution compute time. Filter loading will not be affected as unique filters are not done across slices, rather filters are replicated across the slices. Input loading, however, will decrease since we are using the additional intra-slice bandwidth of new slices to decrease the time it takes to broadcast inputs. Inter-layer data transfer will also be reduced due to less outputs being transfered in each slice.

VII. Related Work

To the best of our knowledge, this is the first work that exploits in-situ SRAM computation for accelerating inferences on DNNs. Below we discuss closely related prior works.

In-Memory Computing: *Neural Cache* has its roots in the processing-in-memory (PIM) line of work [3], [4]. PIMs move logic *near* main memory (DRAM), and thereby reduce the gap between memory and compute units. *Neural Cache*, in contrast, morphs cache (SRAM) structures into compute units, keeping data *in-place*.

It is unclear if it would be possible to perform true in-place DRAM operations. There are four challenges. First, DRAM writes are destructive, thus in-place computation will corrupt input operand data. Solutions which copy data and compute on them are possible [30], but slow. Second, the margin for sensing data in DRAM bit-cells is small, making sensing of computed values error prone. Solutions which change DRAM bit-cell are possible but come with 2-3× higher area overheads [30], [31]. Third, bit-line peripheral logic needed for arithmetic computation has to be integrated with DRAM technology which is not optimized for logic computation. Finally, data scrambling and address scrambling in commodity DRAMs [32]–[34] make re-purposing commodity DRAM for computation and data layout challenging.

While in-place operations in Memristors is promising [35], [36], Memristors remain a speculative technology and are also significantly slower than SRAM.

As caches can be found in almost all modern processors, we envision *Neural Cache* to be a disruptive technology that can enhance commodity processors with large data-parallel accelerators almost free of cost. CPU vendors (Intel, IBM, etc.) can thus continue to provide high-performance general-purpose processors, while enhancing them with a co-processor like capability to exploit massive data-parallelism. Such a processor design is particularly attractive for difficult-to-accelerate applications that frequently switch between sequential and data-parallel computation.

ASICs and FPGAs: Previously, a variety of neural network ASIC accelerators have been proposed. We discuss a few below. DaDianNao [37] is an architecture for DNNs which integrates filters into on-chip eDRAM. It relieves the limited memory bandwidth problem by minimizing external communication and doing near-memory computation. Eyeriss [11] is an energy-efficient DNN accelerator which reduces data movement by maximizing local data reuse. It leverages row-stationary data flow to adapt to different shapes of DNNs. It adopts a spatial architecture consisting of 168 PE arrays connected by a NoC. In Eyeriss terminology, *Neural Cache* has a hybrid of weight stationary and output stationary reuse pattern. Neurocube [7] is a 3D DRAM accelerator solution which requires additional specialized logic integrated memory chips while *Neural Cache* utilizes

widely-used SRAM. The Tensor Processing Unit (TPU) [22] is another ASIC for accelerating DNN inferences. The TPU chip features a high-throughput systolic matrix multiply unit for 8-bit MAC, as well as 28 MB on-chip memory.

In general, custom ASIC accelerator solutions achieve high efficiency while requiring extra hardware and incurring design costs. ASICs lack flexibility in that they cannot be re-purposed for other domains. In contrast, our work is based on the cache, which improves performance of many other workloads when not functioning as a DNN accelerator. *Neural Cache* aims to achieve high performance, while allowing flexibility of general purpose processing. Further, *Neural Cache* is limited by commercial SRAM technology and general purpose processor's interconnect architecture. A custom SRAM accelerator ASIC can potentially achieve significantly higher performance than *Neural Cache*. Being a SRAM technology, we also expect the compute efficiency of *Neural Cache* to improve with newer technology nodes.

The BrainWave project [38] builds an architecture consisting of FPGAs connected with a custom network, for providing accelerated DNN service at a datacenter scale. The FPGA boards are placed between network switches and host servers to increase utilization and reduce communication latency between FPGA boards. The FPGA board features a central matrix vector unit, and can be programmed with a C model with ISA extensions. BrainWave with a network of Stratix 10 280 FPGAs at 14 nm is expected to have 90 TOPs/s, while *Neural Cache* (Xeon E5 2-socket processor) achieves 28 TOPs/s at 22 nm technology without requiring any additional hardware. BrainWave with current generation FPGAs achieves 4.7 TOPs/s.

Terasys presents a bit-serial arithmetic PIM architecture [39]. Terasys reads the data out and performs the compute in bit-serial ALU's outside the array. *Neural Cache* differs by performing partial compute along the bitlines and augments it with a small periphery to perform arithmetic in an area efficient architecture. Further Terasys performs software transposes while *Neural Cache* has a dedicated hardware transpose unit, the TMU.

Bit-serial computation exploits parallelism at the level of numerical representation. Stripes [40] leverages bit-serial computation for inner product calculation to accelerate DNNs. Its execution time scales proportionally with the bit length, and thus enables a direct trade-off between precision and speed. Our work differs from Stripe in that *Neural Cache* performs in-situ computation on SRAM cells, while Stripe requires arithmetic functional units and dedicated eDRAM.

Sparsity in DNN models can be exploited by accelerators [41], [42]. Utilizing sparsity in DNN models for *Neural Cache* is a promising direction for future work.

VIII. Conclusion

Caches have traditionally served only as intermediate low-latency storage units. Our work directly challenges this conventional design paradigm, and proposes to impose a dual responsibility on caches: store *and* compute data. By doing so, we turn them into massively parallel vector units, and drastically reduce on-chip data movement overhead. In this paper we propose the *Neural Cache* architecture to allow massively parallel compute for Deep Neural Networks. Our advancements in compute cache arithmetic and neural network data layout solutions allow us to provide competitive performance comparably to modern GPUs with negligible area overheads. Nearly three-fourth of a server class processor die area today is devoted for caches. Even accelerators use large caches. Why would one *not* want to turn them into compute units?

IX. Acknowledgements

We thank members of M-Bits research group for their feedback. This work was supported in part by the NSF CAREER-1652294 award, and Intel gift award.

References

[1] W. A. Wulf and S. A. McKee, "Hitting the memory wall: Implications of the obvious," *SIGARCH Comput. Archit. News*, vol. 23, no. 1, pp. 20–24, Mar. 1995.

[2] M. Gokhale, B. Holmes, and K. Iobst, "Processing in memory: The terasys massively parallel pim array," *Computer*, vol. 28, no. 4, pp. 23–31, 1995.

[3] P. M. Kogge, "Execube-a new architecture for scaleable mpps," in *Parallel Processing, 1994. Vol. 1. ICPP 1994. International Conference on*, vol. 1. IEEE, 1994, pp. 77–84.

[4] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick, "A case for intelligent ram," *Micro, IEEE*, vol. 17, no. 2, pp. 34–44, 1997.

[5] "Hybrid memory cube specification 2.0," 2014.

[6] J. Ahn, S. Yoo, O. Mutlu, and K. Choi, "Pim-enabled instructions: a low-overhead, locality-aware processing-in-memory architecture," in *Computer Architecture (ISCA), 2015 ACM/IEEE 42nd Annual International Symposium on*. IEEE, 2015, pp. 336–348.

[7] D. Kim, J. Kung, S. Chai, S. Yalamanchili, and S. Mukhopadhyay, "Neurocube: A programmable digital neuromorphic architecture with high-density 3d memory," in *Proceedings of ISCA*, vol. 43. IEEE, 2016, pp. 380–392.

[8] S. Jeloka, N. B. Akesh, D. Sylvester, and D. Blaauw, "A 28 nm configurable memory (tcam/bcam/sram) using push-rule 6t bit cell enabling logic-in-memory," *IEEE Journal of Solid-State Circuits*, vol. 51, no. 4, pp. 1009–1021, 2016.

[9] S. Aga, S. Jeloka, A. Subramaniyan, S. Narayanasamy, D. Blaauw, and R. Das, "Compute caches," in *Proceedings of the 23rd International Symposium on High Performance Computer Architecture (HPCA-23)*. IEEE, 2017, pp. 481–492.

[10] J. Cong and B. Xiao, "Minimizing computation in convolutional neural networks," in *International conference on artificial neural networks*. Springer, 2014, pp. 281–290.

[11] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*. IEEE, 2016, pp. 367–379.

[12] S. Jeloka, N. Akesh, D. Sylvester, and D. Blaauw, "A configurable tcam / bcam / sram using 28nm push-rule 6t bit cell," ser. IEEE Symposium on VLSI Circuits. IEEE, 2015, pp. C272–C273.

[13] M. Kang, E. P. Kim, M. s. Keel, and N. R. Shanbhag, "Energy-efficient and high throughput sparse distributed memory architecture," in *2015 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2015, pp. 2505–2508.

[14] M. Huang, M. Mehalel, R. Arvapalli, and S. He, "An energy efficient 32-nm 20-mb shared on-die L3 cache for intel® xeon® processor E5 family," *J. Solid-State Circuits*, vol. 48, no. 8, pp. 1954–1962, 2013.

[15] W. Chen, S.-L. Chen, S. Chiu, R. Ganesan, V. Lukka, W. W. Mar, and S. Rusu, "A 22nm 2.5 mb slice on-die l3 cache for the next generation xeon® processor," in *VLSI Technology (VLSIT), 2013 Symposium on*. IEEE, 2013, pp. C132–C133.

[16] K. E. Batcher, "Bit-serial parallel processing systems," *IEEE Transactions on Computers*, vol. 31, no. 5, pp. 377–384, 1982.

[17] P. B. Denyer and D. Renshaw, *VLSI signal processing: a bit-serial approach*, vol. 1.

[18] A. Subramaniyan, J. Wang, E. Balasubramanian, D. Blaauw, D. Sylvester, and R. Das, "Cache automaton," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2017, pp. 259–272.

[19] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision," in *Proceedings of the 32nd International Conference on Machine Learning (ICML-15)*, 2015, pp. 1737–1746.

[20] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *arXiv preprint arXiv:1510.00149*, 2015.

[21] S. Zhou, Y. Wu, Z. Ni, X. Zhou, H. Wen, and Y. Zou, "Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients," *arXiv preprint arXiv:1606.06160*, 2016.

[22] N. P. Jouppi, C. Young, N. Patil, D. Patterson *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ACM, 2017, pp. 1–12.

[23] "Parabix Transform." http://parabix.costar.sfu.ca/wiki/ParabixTransform, accessed: 2017-11-20.

[24] D. Lin, N. Medforth, K. S. Herdy, A. Shriraman, and R. Cameron, "Parabix: Boosting the efficiency of text processing on commodity processors," in *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on*. IEEE, 2012, pp. 1–12.

[25] Intel Corporation, "Cache Allocation Technology," https://software.intel.com/en-us/articles/introduction-to-cache-allocation-technology, accessed: 2017-11-20.

[26] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 2818–2826.

[27] H. David, E. Gorbatov, U. R. Hanebutte, R. Khanna, and C. Le, "Rapl: memory power estimation and capping," in *Low-Power Electronics and Design (ISLPED), 2010 ACM/IEEE International Symposium on*. IEEE, 2010, pp. 189–194.

[28] Nvidia Corporation, "Nvidia system management interface," https://developer.nvidia.com/nvidia-system-management-interface, accessed: 2017-11-18.

[29] Intel Corporation, "Intel vtune amplifier performance profiler," https://software.intel.com/en-us/intel-vtune-amplifier-xe, accessed: 2017-11-18.

[30] V. Seshadri, D. Lee, T. Mullins, H. Hassan, A. Boroumand, J. Kim, M. A. Kozuch, O. Mutlu, P. B. Gibbons, and T. C. Mowry, "Ambit: In-memory accelerator for bulk bitwise operations using commodity dram technology," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2017, pp. 273–287.

[31] S. Li, D. Niu, K. T. Malladi, H. Zheng, B. Brennan, and Y. Xie, "Drisa: A dram-based reconfigurable in-situ accelerator," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2017, pp. 288–301.

[32] Intel Corporation, *6th Generation Intel® Processor Datasheet for S-Platforms*, 2015.

[33] I. Skochinsky, "Secrets of intel management engine," http://www.slideshare.net/codeblue_jp/igor-skochinsky-enpub, accessed: 2016-02-17.

[34] S. Gueron, "A memory encryption engine suitable for general purpose processors." *IACR Cryptology ePrint Archive*, vol. 2016, p. 204, 2016.

[35] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar, "Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars," in *Proceedings of the 43rd International Symposium on Computer Architecture*. IEEE Press, 2016, pp. 14–26.

[36] P. Chi, S. Li, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie, "PRIME: A novel processing-in-memory architecture for neural network computation in reram-based main memory," in *43rd ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2016, Seoul, South Korea, June 18-22, 2016*, 2016, pp. 27–39.

[37] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam, "Dadiannao: A machine-learning supercomputer," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2014, pp. 609–622.

[38] E. Chung, J. Fowers, K. Ovtcharov, M. Papamichael, A. Caulfield, T. Massengil, M. Liu, D. Lo, S. Alkalay, M. Haselman, C. Boehn, A. Forin, K. S. Gatlin, M. Ghandi, S. Heil, K. Holohan, T. Juhasz, R. K. Kovvuri, S. Lanka, F. v. Megen, D. Mukhortov, P. Patel, S. Reinhardt, A. Sapek, R. Seera, B. Sridharan, L. Woods, P. Yi-Xiao, R. Zhao, and D. Burger, "Accelerating persistent neural networks at datacenter scale," 2017, hot Chips: A Symposium on High Performance Chips.

[39] M. Gokhale, B. Holmes, and K. Iobst, "Processing in memory: The terasys massively parallel pim array," *Computer*, vol. 28, no. 4, pp. 23–31, 1995.

[40] P. Judd, J. Albericio, T. Hetherington, T. M. Aamodt, and A. Moshovos, "Stripes: Bit-serial deep neural network computing," in *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*. IEEE, 2016, pp. 1–12.

[41] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "Eie: efficient inference engine on compressed deep neural network," in *Proceedings of the 43rd International Symposium on Computer Architecture*. IEEE Press, 2016, pp. 243–254.

[42] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, "Scnn: An accelerator for compressed-sparse convolutional neural networks," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ACM, 2017, pp. 27–40.