CASH: Compiler Assisted Hardware Design for Improving DRAM Energy Efficiency in CNN Inference

Anup Sarma Huaipan Jiang Ashutosh Pattnaik Jagadish Kotra
Mahmut Taylan Kandemir Chita R Das
The Pennsylvania State University

ABSTRACT

The advent of machine learning (ML) and deep learning applications has led to the development of a multitude of hardware accelerators and architectural optimization techniques for parallel architectures. This is due in part to the regularity and parallelism exhibited by the ML workloads, especially convolutional neural networks (CNNs). However, CPUs continue to be one of the dominant compute fabric in data-centers today, thereby also being widely deployed for inference tasks. As CNNs grow larger, the inherent limitations of a CPU-based system become apparent, specifically in terms of main memory data movement. In this paper, we present CASH, a compiler-assisted hardware solution that eliminates redundant data-movement to and from the main memory and, therefore, reduces main memory bandwidth and energy consumption. Our experimental evaluations on a set of four different state-of-the-art CNN workloads indicate that CASH provides, on average, ~40% and ~18% reductions in main memory bandwidth and energy consumption, respectively.

CCS CONCEPTS

• **Computer systems organization → Neural networks**; **Multicore architectures**; • **Software and its engineering → Runtime environments**.

1 INTRODUCTION

In recent years, there has been tremendous growth in the adoption of machine learning (ML) algorithms in applications used on wearables to warehouse scale-computing for diverse areas such as medical diagnosis, gaming, etc. One of the most widely used ML models in practice today are based on convolution neural networks (CNNs). CNNs are frequently employed to find solutions to grid-based problems and, in general, provide higher accuracy compared to traditional rule-based algorithms. Over the years, as more problems are being solved using CNNs, these ML models, also known as deep neural networks (DNNs), have increased in complexity (higher number of layers) and contain many different types of layers [21, 23, 28, 40, 41].

Such large-scale adoption of DNNs has generated a plethora of research work, resulting in improved performance and energy efficiency. Notably, a series of hardware accelerators have been proposed to directly meet the reuse and data flow characteristics of such networks [4, 7, 13, 25]. Also, due to the high compute and data movement demands,processing-in-memory solutions [8, 26, 39] have been proposed as well. However, deep learning models have been rapidly evolving over the years, and apart from CONV/FC layers, also consist of non-GEMM amenable layers, such as data-compaction, dilated convolution or deformable convolution. State

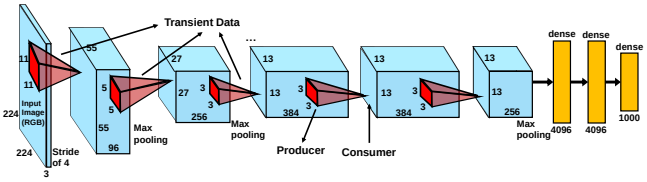


Figure 1: Layer wise execution model of a CNN [28].

of the art accelerators were designed mostly keeping compute intensive CONV/FC layers in mind, however, evolving nature of these workloads requires frequent synchronization of accelerators with host CPU for non-GEMM tasks, thereby incurring significant data movement cost. On the other hand, modern processors are also being equipped with wide vector extension modules and scatter-gather operations that make them compelling for data parallel tasks, given the high NRE cost & deployment time associated with standalone accelerator.

Prior works, focusing on general-purpose processors, have dealt with exploring the compute and communication characteristics of DNNs on large-scale compute nodes during the training phase [10, 11, 24, 27, 44]. While GPUs are preferred for training and batched execution of inference, however, under real-time latency constraints, CPUs are currently the choice of compute fabric at large scale data-centers [20, 34].As the number of cloud and datacenter services that run inferences on CPUs have increased tremendously [9, 15, 19], *it is imperative that the energy efficiency of inference execution on CPUs be improved further*. Therefore, in this paper, we focus on improving the energy efficiency of CNN inference execution on CPU-based systems.

Figure 1 shows a generic layout of a CNN [28]. These networks are modeled in a layer-wise fashion and follow a producer-consumer paradigm, where the first layer receives the input, and the subsequent layers *consume* the data *produced* from their preceding layer(s). In this work, we refer to the data generated from the intermediate layers as *"transient data"*.[1] The transient data is only needed during its consume phase and is *no longer required* once it is consumed. However, under current execution paradigm, this transient data is read and written-back to the main memory *unnecessarily*, thereby increasing both memory bandwidth and energy consumption. Therefore, it is imperative that any unnecessary data movement due to transient data be eliminated to further improve the energy efficiency of CPU-based inference execution. Our goal in this paper is three-fold: (1) to find the *live range of the transient data* that lead to unnecessary memory accesses; (2) to propose a simple

[1]We use the terms "transient" and "intermediate" interchangeably.

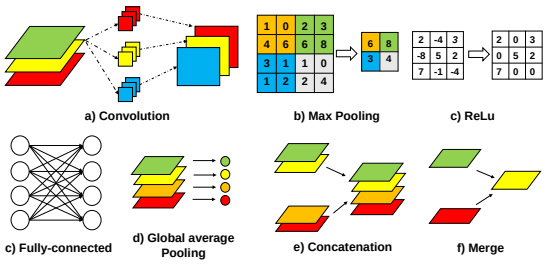


Figure 2: Different types of layer operations in a CNN

hardware mechanism to *discard unnecessary memory transactions* generated by the transient data;

and (3) to propose an optimized compiler algorithm to *identify the required buffer size that can be effectively reused across all layers during execution*. To this end, we propose **CASH**, a Compiler-ASsisted Hardware design that performs a liveness analysis on the application code to identify the live ranges of the transient data structures and instruments them in the application binary. A simple hardware filtering mechanism is implemented at the memory controller that keeps track of the transient data liveness information provided by the updated application binary. Whenever a memory transaction belonging to the transient data arrives at the memory controller, the filtering mechanism detects it and conditionally discards it, eliminating any unnecessary data movement to/from the main memory. Furthermore, it also provides a compiler optimization that finds the smallest buffer size that is needed by the CNN and allocates only a single buffer that is reused across all the layers of the network.

To our knowledge, this is the first work that considers reducing the memory traffic generated by CNN inferences in a CPU-based system by discarding any memory requests belonging to transient data at the memory controller. This paper makes the following major **contributions**:

• It proposes CASH, a Compiler ASsisted Hardware scheme that provides a hardware mechanism and compiler optimization method to reduce unnecessary data movement and reduce memory bandwidth and energy consumption.

• The first method proposes a memory controller based data region tracking mechanism and uses the liveness information passed from the application binary to filter out unnecessary memory transactions. The second method proposes a compiler optimization that allows for the creation & management of a single buffer space that is re-used throughout the course of a CNN execution.

• It comprehensively evaluates CASH using 4 state-of-the-art CNNs. The proposed optimizations provide, on an average, bandwidth reduction of ~40% and main memory energy savings of ~18%.

2 BACKGROUND AND MOTIVATION

2.1 Background on CNN computation

Convolutional neural networks usually comprise of multiple repeating instances of the following layers: convolution (CONV), pooling (POOL), activation (ACT — for example, ReLU), fully connected (FC), and global average pooling layer, as shown in Figure 2 (a)-(d). In a CONV layer, multiple filter kernels are applied to the input feature maps to generate output feature maps. The filter weights are typically learned offline during the training phase. POOL layer aggregates the features across a window of neighboring pixels of the output feature map. FC layers are usually present at the end of CNNs and acts as the classifier. In addition to the above well-known layers, recent state-of-the-art networks have introduced additional types of layers in CNNs for higher accuracy. The most important is the layer-wise concatenation (CONCAT), as shown in Figure 2 (e). It is used to propagate features from an earlier stage of the network to a later stage [23]. The concatenation is performed by placing together multiple feature maps from the previous layers into one contiguous memory region to be used as an input feature map for the subsequent layer. Another similar approach to propagate feature information is point-wise merger (MERGE), as shown in Figure 2 (f). It is used in identity mapping or shortcut connections in residual networks[21]. Unlike the CONV layer, these layers have less computational requirements, but leads to data re-use patterns which are not sequential in nature.

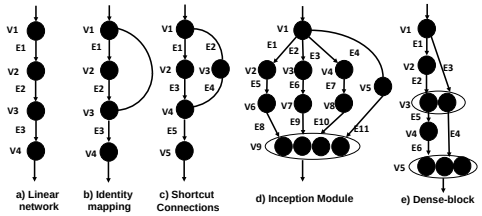
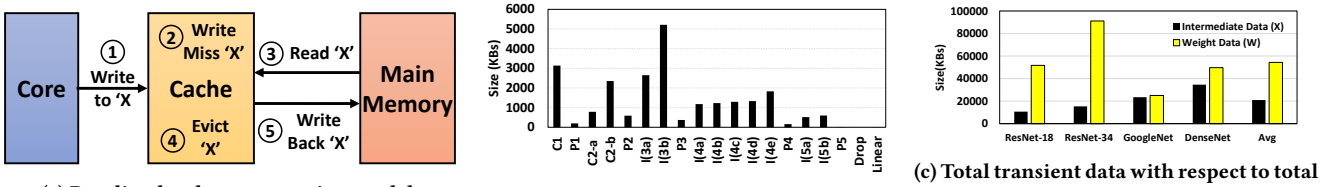


Figure 3: Different network topologies for CNN.

As shown in Figure 3, CNNs can be interpreted as a *directed acyclic graph* G ≡ (V,E), where V = $\{V_1, V_2, V_3, ...V_n\}$ is the set of vertices representing each layer, and E ⊆ {V × V} is the set of edges representing different operations (CONV, POOL, etc.) involved in a CNN application. Figure 3 depicts various CNN topologies that are in use today such as a linear, identity, and shortcut connection mappings (used in ResNet), inception modules (used in GoogLeNet) and denseblocks (used in DenseNet).

2.2 Motivation

As discussed in the previous section, CNN models are stacked layers of operations whose inputs and output(s) are represented as continuous blocks of data in main memory. As a result, when the core starts processing an input data layer to generate the subsequent output layer (①), it incurs "cold-write" misses (②) that propagate all the way down to the main memory (③) (referred to as write-allocate reads). Similarly, when an output layer is generated, the previous layer's dirty data is "written back" (⑤) to the main memory upon a memory de-allocation, or as a part of regular dirty cache line replacement (④) [22]. This particular behaviour of the hardware is shown in Figure 4a. We observe that this issue of write allocation reads and write-back is significant in CNNs and can be exploited to reduce off-chip bandwidth and main memory energy savings. We motivate the transient data movement issue via an example by analyzing the data generation characteristics of 4 state-of-the-art CNNs. Figure 4b shows the amount of intermediate data generated by different layers during the course of execution of `GoogLeNet`. This data can be as high as 3000KBs for a given layer. Recall that



(a) Baseline hardware execution model.

(b) Layerwise transient data in GoogLeNet.

(c) Total transient data with respect to total weight data in four ML network.

Figure 4: Hardware execution model and transient data involved in networks

the data generated by C1 is consumed by P1 (Section 2.1). However, considering the baseline CPU architecture and memory hierarchy, all the intermediate data for C1 (3000KB) need to be fetched from main memory due to cold misses. This is unnecessary as the *malloc*ed data is garbage in the beginning and are computed at runtime. Similarly, after the data is computed, it is eventually written back to the main memory during cache eviction. This is also unnecessary, as once the computed data is consumed by the next layer, it is *no* longer needed and does *not* need to be written back to main memory. This behavior is repeated from a layer to any future layer (not just the successive ones) and across most CNNs. Figure 4c shows the total transient data and total weight data for 4 different CNNs. We observe that the intermediate data can be a significant fraction of the total weights, which are to be read from main memory, ranging from ~10% in `ResNet-34` to ~100% in `GoogLeNet`. In general, a layer with transient data size X and weight size W would incur $(W+2X)$ bytes of data to be fetched from main memory. Our scheme would require fetching only W bytes, resulting in date movement savings of $[100/(1+(W/2X))]$%. Therefore, as seen in Figure 4c, on an average, X is ~21MB and W is ~54MB, leading to a memory bandwidth savings of 43.6%.

3 CASH DESIGN

The development of CASH involves design and integration of two distinct aspects: First is the addition of a hardware mechanism to track memory data regions with compiler identified liveness information. Second is a compiler optimization approach, which can reduce the cost of hardware unit by identifying and managing a minimum data region which is required for network execution.

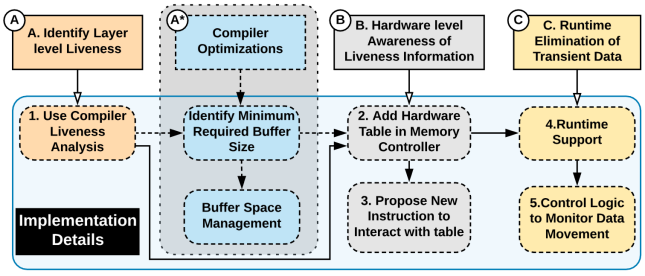


Figure 5: Overview of our proposed scheme.

In this section, we outline the high level view of our proposed mechanism. Figure 5 shows the various steps involved. In order to efficiently eliminate the unnecessary data movements involving the intermediate data structures, we first need to identify such intermediate data structures (Ⓐ). To this end, our scheme adopts a compiler-assisted "liveness analysis" on the application code. While liveness analysis is used for traditional register allocation and dead code elimination [33], we use it in the context of liveness of transient data, by adopting an approach similar to the work of Guyer et al. [16]. The liveness information collected from step Ⓐ needs to be passed on to the hardware to make it aware of the liveness scope (Ⓑ). To keep track of the liveness information in the hardware, we augment the memory controllers with a register file, referred to as Transient Data Detect Table (TDDT). Furthermore, we propose a new ISA instruction that is capable of passing the liveness information along with the transient data structure sizes to the memory controller and populate the TDDT entry. This instruction is inserted into the application binary by the compiler for each transient data structure after the liveness analysis. For efficient elimination of unnecessary memory transactions (Ⓒ), we incorporate a simple hardware filtering mechanism at the memory controller. At runtime, when the memory controller receives a memory transaction, the hardware unit checks if it belongs to any of the intermediate data structures. It discards any requests belonging to the transient data from being sent to the main memory and only sends necessary memory transactions to be issued to the main memory. Due to scalability limitations of per layer hardware tracking, in the second part of compiler optimization (Ⓐ*), we refine our approach to identify and reuse only a single buffer throughout network execution. We discuss the various intricacies involved in the buffer management of various types of CNNs and provide an efficient algorithm to handle such scenarios. Next, we describe in detail the individual steps 1 through 5 of our proposed scheme.

3.1 Hardware mechanism to track data regions with compiler liveness analysis

3.1.1 ***Compiler Liveness Analysis:*** Liveness analysis of data variables is a well-known technique [16]. We perform a static liveness analysis of a program's data structures to determine their live ranges in the application code. Let us consider the example shown in Figure 6. It shows the liveness of the various data structures[2] of a sample NN graph and its corresponding representative high-level code during different instances of its execution. At time t2, the computation of Layer1 has just finished, which in turn is going to be used to generate Layer2. Therefore, the live set consists of L1. At time t3, the computation to generate L2 has finished, and

[2]For simplicity, we club all the transient data structures present in a layer together and find the liveness of a layer as a whole.

therefore, the live set consists only of L2 as L1 has been moved to the dead set. Similarly, We can use the same principle to annotate any arbitrarily complex NN graph to find the liveness and use specifically designed instructions to pass the liveness information to the memory controllers, as discussed below.

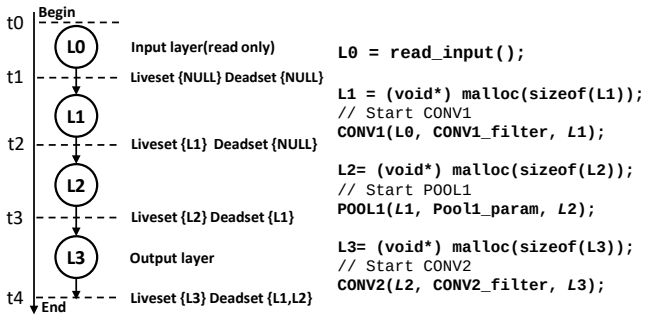


Figure 6: Layer wise liveness of a sample NN graph.

3.1.2 ***TDDT Register File at the Memory Controller:*** Once the intermediate data structures have been identified by the compiler-assisted liveness analysis, our scheme requires the associated hardware support to *track* the liveness information. For this, we propose adding an SRAM-based register file at the memory controller (referred to as Transient Data Detection Table or TDDT for short), which is in a sense a programmable register file that is exposed to the core via ISA. We also add the corresponding ISA extension needed to populate this register file with necessary data values (discussed next). These register files are used to store the physical address range and status of the intermediate data structure. TDDT consists of a 32-bit physical address field, a 30-bit intermediate data size field, and the read and write-back status bits. Based on the values populated in these registers and address of the memory transactions reaching the memory controller, it is decided whether to forward the request to the main memory, or simply return it (in the case of reads due to write allocate), or discard it (in the event of write-back while layer/data-structure lifetime has come to an end). Note that, by allowing the register file at the memory controller to be directly programmed by an ISA instruction, it becomes a part of the process execution context. Therefore, the register file needs to be replicated and made exclusive for each CPU core.

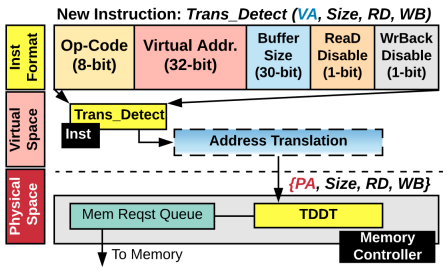


Figure 7: ISA extension and system interaction.

3.1.3 ***ISA Extension:*** In order to populate the TDDT entry fields at the memory controller, we propose an ISA extension. The ISA instruction is used to annotate the intermediate data structure with its base address and size based on the liveness analysis and compiler-passed information, which gets allocated before its actual usage. The instruction format is shown in Figure 7. It has an opcode field, address field, size and two additional bit fields. The bit fields are used to indicate read_disable or write_disable status of the data structure, respectively. When the instruction is issued from the core with its base virtual address, it performs the TLB lookup and the base physical address of the data structure is passed on to the memory controller along with its size.

We use the mnemonic *Trans_Detect(arg1, arg2, arg3, arg4)* to refer to this instruction call. Note that the ISA format shown here is for illustrative purposes only and will require specific adaptation for the target platform.

3.1.4 ***Operating System Support.*** We adopt a simple range comparator logic to determine the status of an incoming address at the memory controller. However, if the data-structure spans multiple pages, we cannot guarantee contiguity of the data structure based on base address only, as the physical page allocation might be discontinuous in the main memory. Also, the overhead of maintaining metadata to handle such cases would be very high. Therefore, we require support from the operating system to provide a physically contiguous address space [18] for the malloc() operation for the intermediate data structure. With OS support, given the base address and the size of a data structure, we can easily determine whether the memory request belongs to one of the tagged data structures.

3.1.5 ***Control Logic for Filtering Transient Data :*** As discussed earlier, we add TDDT in the memory controller to tag the starting address and the size of each transient layer in the ML inference application. In addition to the address and the size field, we also maintain status bits to indicate the status of the data structure, viz., read_disable or write_disable, which is enabled based on the arrival of special ISA instruction as discussed earlier. TDDT at the memory controller is organized as a FIFO queue, and the incoming ISA request (used to tag the memory regions) is used to populate TDDT entry fields. Figure 8 shows the operation of the memory controller using a representative example. As shown in Figure 8(a), the compiler identifies and inserts Trans_Detect() instruction calls in the program code. At instant t0, prior to computation of CONV1, the base address and size of Layer-1 (L1) data-structure are passed to the memory controller using the Trans_Detect call with the read_disable flag bit set and the TDDT entries are filled out (❸) in a FIFO order as maintained by the TDD controller (❺). During the computation phase of CONV1, the read request packets (❶) arrive at the memory controller, and are added to the Transactions Buffer (❷). If the difference between the requesting address and the base address from the TDDT entry is found to be less than the corresponding size field of the TDDT entry (referred to as range-matching) along with read_disable flag, the read request is considered as "pseudo" by ❹ after performing a table lookup (❻) and returns a zero valued response packet (❼). During the CONV1 computation, the read requests corresponding to the input and weights pass through the memory controller but read requests for

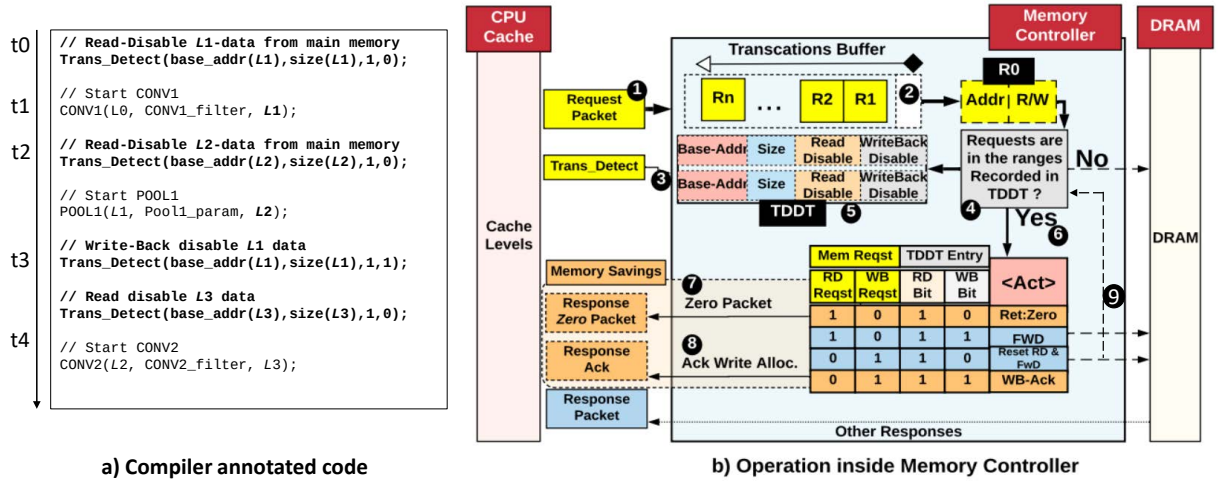


Figure 8: Compiler annotated code and operation of modified memory controller

Layer-1 intermediate data (due to write misses) get filtered out. Until a range-match is found, ❹ issues request to the TDD-controller (❺) to provide the next TDD entry (scenario-b in Figure 8(b)). If no range-match is found in all the entries of TDDT, the memory request is sent to main memory.

Similarly, at instant t2, Layer-2's (L2) metadata is passed to the memory controller and it fills up the second entry in TDDT with the read_disable flag bit set. Thus, during computation of POOL1, pseudo reads for L2-data are filtered out. At instant t3, the L1-data becomes a part of the dead set and is identified through the liveness analysis. Therefore, any write-backs of this data to the main memory is *inconsequential*. The compiler, therefore, inserts Trans_detect instruction for L1 data structure with write_disable (WB) flag bit set; this updates the WB flag bit of the corresponding TDDT entry. Thus, any instant t ≥t3, the WB requests corresponding to L1-data are discarded by the memory controller logic (❽).

What happens if write-back occurs before the WB disable bit is set at the TDDT entry? We refer to this condition as "overflow of transient data". When an overflow situation is detected, we simply reset the read_disable flag of the corresponding TDDT entry (❾) and forward the writeback to main memory. This enables us to guarantee consistency, as any future read to this data structure will not be filtered out by our controller logic, and will be read from main memory instead. In the next section, we discuss potential drawback of a hardware only approach and propose necessary optimization steps at the compiler-side to overcome the drawback.

Algorithm 1 Find the global and local minimum buffer size.

Input: A Neural Network G ≡ $\{L_i | i \in 1,\ 2 \ldots n\}$

Initialization:

$BufferSize \leftarrow L_1.inputsize + L_1.outputsize$;
$Stack.push(L_1)$
$Local\_BufferSize \leftarrow \{BufferSize\}$
$Min\_BufferSize \leftarrow L_0.outputsize$
Repeat
$L \leftarrow Stack.pop()$
for *each successors layer* L_i *of layer* L **do**
  if *the successor of* L_i *is not a converging layer* **then**
    $BufferSize \leftarrow BufferSize + L_i.outputsize$
    $Stack.push(L_i)$
  else
    if L_i *is the last predecessor of its successor* **then**
      $Stack.push(L_i)$
    end
  end
  if L_i *is a converging layer* **then**
    $BufferSize \leftarrow BufferSize + L_i.outputsize$
  end
end
$Min\_BufferSize \leftarrow max(BufferSize,\ Min\_BufferSize)$
if *the successor of* L *is not a converging layer* **then**
  $BufferSize \leftarrow BufferSize - L.outputsize$
end
if *Layer* L *is re-sized by pooling layer* **then**
  Update $Local\_BufferSize$
end
Until $Stack$ is empty
Return $Min\_BufferSize, Local\_BufferSize$;

4 COMPILER OPTIMIZATION

4.1 Scalability Issues with Per-Layer Tracking

As discussed in the previous section, our proposed hardware mechanism requires tracking each layer's intermediate data at the memory controller. However, the state-of-the-art neural networks can well exceed over 100 layers, and therefore, would require a very large register file to be implemented at the memory controller. In addition, they would also add to the overhead of runtime execution, as each memory transaction will now need to be range-matched with all the TDDT entries.

Therefore, we propose a compiler approach to address this issue by noting that, it is not necessary to malloc() a new memory region for each new layer of neural network. Instead, we can *reuse* the memory region corresponding to a dead layer towards the

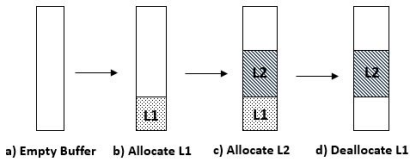


Figure 9: A naive buffer space management scheme.

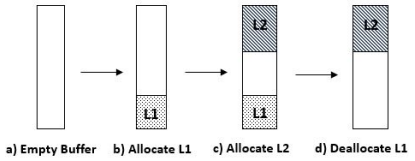


Figure 10: A ping-pong allocation strategy.

generation of a next layer's data. Thus, we need to identify the minimum buffer size that will be required during the course of the network execution. Algorithm 1 provides the pseudo code to determine the execution order of the independent layers and infer the minimum required buffer size for allocating the layers. The algorithm essentially obtains the $MinBufferSize$ by computing the $max\{L_{i-1}+L_i, L_i+L_{i+1}\}$ on the network graph. At the end of each layer's lookup, the minimum buffer size is updated. *This allows efficient memory usage and reduces the allocated buffers involved in the inference phase to only "one".* It also outputs Local_BufferSpace which is described in Section V-E.

4.2 Managing Buffer Space

In this section, we show that our proposed scheme requires a well orchestrated buffer management scheme to avoid any buffer overflow scenario. We refer back to the execution schedule we considered previously. As discussed earlier, with the compiler assisted approach, we need to define one contiguous region of buffer that needs to be managed throughout the execution of the network.

In a naive buffer management approach, the memory blocks can be allocated consecutively one after the other. In Figure 9, memory block for region L1 is allocated starting at the bottom of the buffer, and the next block corresponding to L2 is assigned immediately after it. After L2 is generated, block L1 is marked as free for further use. However, this approach can lead to fragmentation by creating holes on both sides of the managed buffer as shown in part Figure 9d). Therefore, despite having enough free space on the buffer, we may not be able to contiguously allocate space for L3, if $sizeof(L3) > sizeof(L1)$ and $sizeof(L3) > MinBufferSize - (sizeof(L1)+sizeof(L2))$. Note that contiguity of blocks is necessary from a performance angle.

4.3 Efficient Buffer Management

To eliminate buffer fragmentation, we need to adopt a simple yet effective way of managing the buffer space. The key observation which leads to fragmentation in earlier scenario was that holes were being created on both sides of an allocated memory region. Therefore, in principle, if we can coalesce the holes in one specific direction of the buffer, we can eliminate the issue of fragmentation. To this end, we resort to a "ping-pong strategy" to allocate and de-allocate buffer space, as illustrated in Figure 10.

We show the pseudo code of *Buffer Management* in Algorithm 2. In this approach, region for layer L1 is allocated starting at the bottom of the stack. For allocating buffer to L2, which is a child-node of L1, we allocate it at the opposite end of the designated buffer region. At the end of generation of L2 data, we set the region belonging to L1 as free, and therefore, the buffer remains *contiguous* and free

Algorithm 2 Buffer management for each layer.

Input: A Neural Network G $\equiv \{L_i | i \in 1,\ 2 \dots n\}$, $Min\_BufferSize$, $Auxiliary\_BufferSize$
Initialization:
$Stack.push(L_0)$
Allocate layer L_1 at bottom of the buffer
Repeat
$L \leftarrow Stack.pop()$
Remove layer L from the buffer
for *each non-converging successors layer* L_i *of layer* L **do**
  if *the successor of* L_i *is not a converging layer* **then**
    $Stack.push(L_i)$
    Allocate layer L_i in the buffer at the opposite end of L
  else
    if L_i *is the last predecessor of its successor* **then**
      $Stack.push(L_i)$
    end
    Allocate layer L_i in the auxiliary space of the buffer
  end
end
if *the successors layer* L_i *of layer* L *is a converging layer* **then**
  $Switch\ the\ logical\ view\ of\ the\ Buffer$
  $Stack.push(L_i)$
end
Until $Stack$ is empty
Return $Min\_BufferSize, Local\_BufferSize$;

of any fragmentation. We show that this approach is extendable to other non-linear network architectures with arbitrary branch structures, and still achieves contiguity for all the allocated blocks.

4.4 Handling Networks with Branching

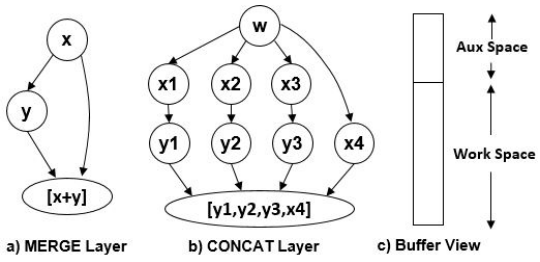


Figure 11: MERGE and CONCAT Layers in CNN.

State-of-the-art CNNs come with variety of branching topologies such as residual, dense or inception blocks. The two key underlying operations which are utilized to converge such branched structures are via MERGE (element-wise reduction of feature map) or

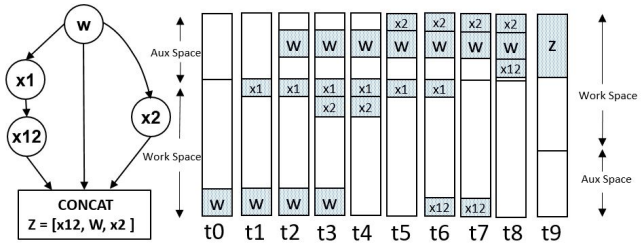


Figure 12: Memory allocation flow for CONCAT

CONCAT (stacking of feature maps), as shown in Figure 11. Due to arbitrary depth and branching factor involved in such structures, we may witness scenarios that need specific adaptation in the buffer management scheme. Therefore, to keep the approach generic and straightforward, we define an *auxiliary* "work-space buffer", which equals to the maximum of the observed CONCAT layer or the MERGE layer size, in addition to the *MinBufferSize* derived from a linear notion of the execution graph.

Figure 12 and 13 shows the flow of the memory block assignment given an execution schedule. The execution schedule shown here follows a BFS ordering, so that the parent node can always be released after all the associated edges have been computed. However, in order to avoid the memory fragmentation issue, in the next run of BFS, it needs to follow a LIFO ordering with respect to the output of the previously-executed edges. At instant t4, all the outgoing edges associated with W (Figure 12) have been executed, and therefore W is being released. Also, it is to be observed that, at t3, while executing the CONCAT operation with respect to W, it is copied to the auxiliary space while preserving the order (as defined by the CONCAT operation). For the MERGE layer scenario Figure 13, the first execution of the edge is an assignment operation, following which the edges can be executed in an additive fashion (because we are re-using allocated spaces, it is going to consist of non-zero values). Therefore, running the MERGE operation (repeated addition) on such a block would lead to inaccurate output value being computed. At end of MERGE or CONCAT layers, we switch the logical view of the buffer in such a way that current auxiliary space becomes a part of work space for the reminder of the execution graph, while carving out equivalent amount of auxiliary space from previous work-space region.

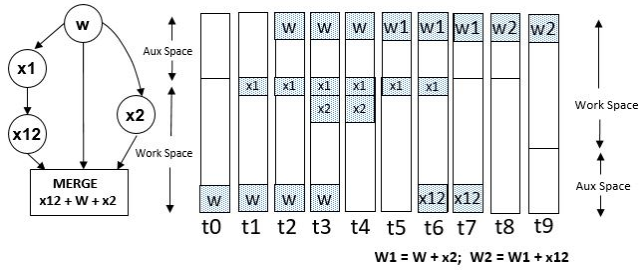


Figure 13: Memory allocation flow for MERGE

Table 1: Configuration parameters for baseline architecture.

CPU	4×Out-of-Order, 2.5GHz
L1-I,D Cache	32kB, 4-way associative with 32B cacheline
L2-Cache	256kB, 8-way associative, Private
L3-Cache	16MB, 16-way associative, Shared
Cache Policy	Mostly Inclusive with LRU
Main Memory	DDR3-1600, 64-bit bus width, 12.8GB/s 1KB row buffer, 8 banks/rank, 8 devices/rank, 2 ranks/channel, t_{RCD} = 13.75ns, t_{CL} = 13.75ns, t_{RP} = 13.75ns, t_{RAS} = 35ns, t_{RFC} = 300ns, t_{REFI} = 7.8us

4.5 Dynamic Buffer Resizing

With the compiler managing the layerwise memory allocation, we need to maintain only a single entry in TDDT per inference, corresponding to a single buffer, and this meets the minimum space requirement throughout the network. However, an important attribute of CNN is that generated featured map periodically undergoes the reduction of resolution. As such, the generated intermediate feature map size decreases towards the end of the network, despite the increase in the number of channels. This in turn means that the required min-buffer size gets smaller in successive phases of the network. Note that this can potentially lead to pseudo overflow scenario, where an un-utilized portion of the buffer can get cache-evicted at runtime, triggering the WB bit set for the buffer. To avoid this scenario, we utilize this characteristics of the network to periodically update the *Local_BufferSize* in Algorithm 1 and insert another entry in TDDT, marking the unused buffer space. By doing so, the pseudo overflow scenario is avoided, and at the same time, the WB corresponding to unused space can be successfully avoided.

5 EVALUATION METHODOLOGY

Simulation Platform: We simulate the baseline architecture given in Table 1 using the GEM5 simulator [3]. We extensively modified GEM5 to implement our proposed scheme. Specifically, we added a pseudo instruction to the Gem5 model which obtains the starting physical address from the virtual address (after the translation lookup) and propagates this information to program the fields of the Transient Data Detection Table (TDDT) fields at the memory controller. The starting address here corresponds to the buffer(s) defined to hold networks layer data. We also add queues to simulate TDDT and implemented the necessary logic for intermediate data handling as described in Section 3. We used the default main memory energy model in GEM5. Note that, all our experiments are performed for a batch size of 1 and simulated until completion of the inference phase. Although batch sizes of greater than 1 are preferred for higher throughput, in a CPU-based execution, increasing the batch size may lead to missing QoS deadlines [34, 42, 43].

CNN Benchmarks: We evaluated 4 state-of-the-art CNN architectures for our study: `ResNet-18`, `ResNet-34` [21], `GoogLeNet` [41], and `DenseNet` [23]. `ResNet-18` and `ResNet-34` have similar layer structures except that the latter has more layers/depth and also introduces the element-wise addition to establish the residual connectivity at every alternate CONV layer. `GoogLeNet` introduces inception module, which involves parallel convolution blocks followed by the concatenation of output feature maps. `DenseNet` introduces a dense-block architecture which consists of a densely connected architecture with recurring concatenation layers.

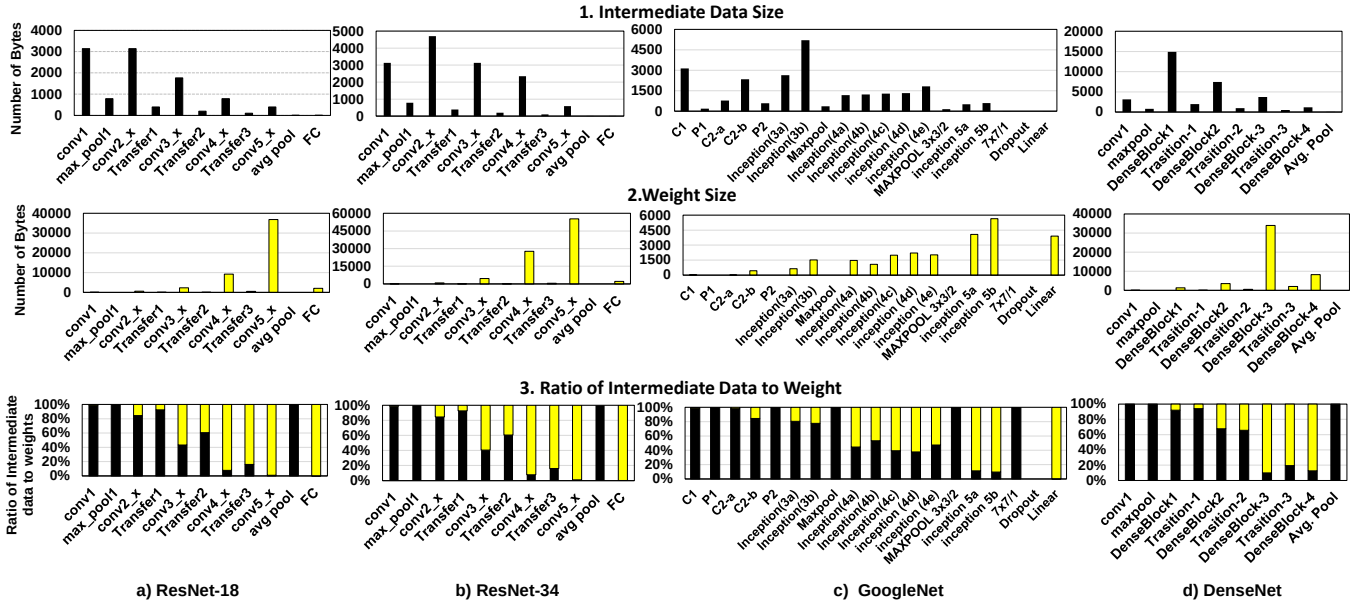


Figure 14: CNN benchmark characteristics.

Note that the baseline version of these benchmarks is the one in which every layer data is individually allocated without any compiler optimization or hardware tracking. However, known techniques of layer fusion and in-place optimization for CONV layer followed by non-linearity and/or batch norm to reduce intermediate data size is already part of the baseline. Similarly, our baseline for `DenseNet` is already an optimized version, which is otherwise known to have quadratic memory growth issue due to recursive malloc operations [35]. The parallelization involves processing different parts of output feature map by different cores simultaneously and independently. In other words, computation of the network graph proceeds by executing one edge at a time, but parallelized across the available cores. For CONV operations, we use direct method for computation without additional memory overheads [31].

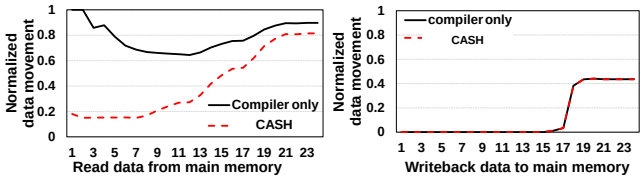
Figure 14 shows the important characteristics of our benchmarks, namely, intermediate data size, weight size, and their relative ratios. In general, the layers with the largest intermediate data are located in the first half of the network, while the weight data grows towards the end of the network. We also observe that, the total weight of a network is usually larger than the total intermediate data size.

6 EXPERIMENTAL RESULTS

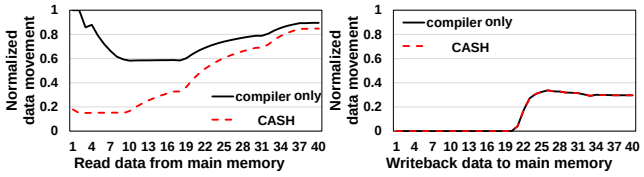
The evaluation metrics considered here are the volume of read and write memory transactions, memory bandwidth savings and the overall savings in DRAM energy consumption. We analyze these metrics under following two scenarios: First, a *Compiler Only* approach in which our proposed compiler specific refinements (Section 3.1) are applied, but without any support for the proposed tracking hardware logic on the memory controller. Second, our proposed *CASH* approach, which builds on top of the compiler only optimizations by adding hardware support at the memory controller for tracking the compiler identified data region (Section 4). Note that the performance improvement are marginal (within 1% of baseline performance) because the data movement latencies are already well hidden by the compute operations in the baseline execution.

Read and Writeback Data: Figure 15 shows the normalized memory reads and writes for all the CNN benchmarks with respect to the baseline. The X-axis shows the discrete layer boundaries. As can be seen, the compiler-only scheme incurs as much reads as the baseline scheme in the initial few layers, while the CASH scheme, with the additional hardware tracking has fewer reads. This is because of the following reason: when the execution begins, the LLC is initially empty. So, even when the intermediate data from the execution are written in the LLC (that usually are captured at the cache level and not percolated to the DRAM), it incurs cold misses. These cold misses for writing intermediate data at the LLC in turn causes a read from the DRAM to the corresponding cache block. This causes the read burst in the DRAM as observed in the beginning of workload execution.

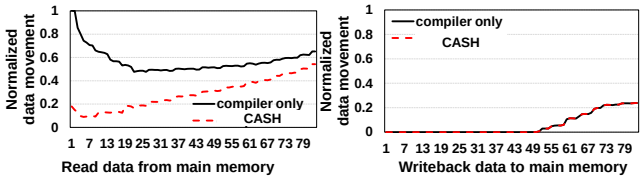
However, towards the end of execution, the two curves exhibit similar patterns. This is due to the fact that reading also incurs mandatory reads for the weight data (which is unavoidable). And, the difference between the two curves determines the ratio of weight data vs intermediate transient data for the particular CNN benchmark. We note that the difference in read data is less for ResNet-34 as compared to ResNet-18, although both have similar network structures. This can be attributed to the depth of the network architecture, as the network depth increases, weights start to dominate more and more over the intermediate data. Similarly, mandatory write-backs are incurred for both baseline and the employed schemes for the stack variables; however, in the baseline scenario, write-backs are also incurred for the intermediate transient data which is significantly greater than the total stack variables used for program management. Note that the write-backs start



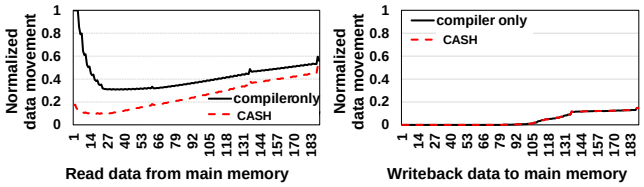
(a) ResNet-18.



(b) ResNet-34.



(c) GoogLeNet.



(d) DenseNet.

Figure 15: Read and Writeback main memory transactions Normalized to the Baseline execution

only after a certain point in program execution — once the LLC is completely filled and starts evicting cache lines.

Impact on DRAM Bandwidth: Figure 16a shows the savings in DRAM bandwidth after the application of proposed techniques to the set of benchmarks. Savings in DRAM bandwidth is governed by volume of memory transactions and computation flops, of which Densenet has the best performing ratio. We also find that the DRAM bandwidth follows the same order as DRAM energy savings, as explained in following subsection.

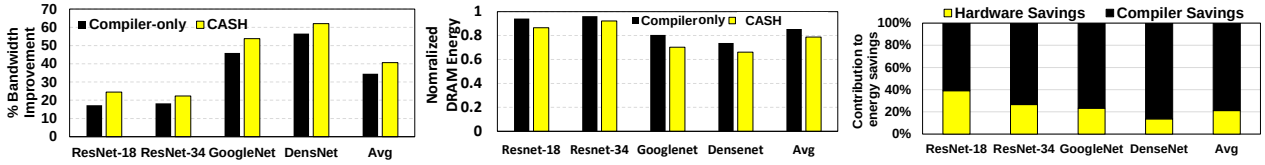
Impact on DRAM Energy: Figure 17 shows, for all our CNN benchmarks, the breakdown of the main memory energy usage for each DRAM function. We find that the self-refresh energy is nearly unchanged for all the benchmarks, but there is a significant difference between read, write, activation and pre-charge energy. As our proposed scheme does not affect the performance of the CNN inference, the DRAM self-refresh energy remains similar. The reduction in other DRAM functions leads to an energy savings of 13.5%, 7.8%, 29.7% and 33.8% for `ResNet-18`, `ResNet-34`, `GoogLeNet` and `DenseNet`, respectively. On an average, the total DRAM energy usage is reduced by 18% for all the evaluated CNN benchmarks. We also observe that both `DenseNet` and `GoogLeNet` achieve higher savings compared to `Resnet-18` and `ResNet-34`. This is because DRAM energy is mainly dominated by self-refresh energy. Therefore, CNNs which have lesser compute requirement and have higher transient data to weight ratio will benefit the most from our proposed mechanism in terms of main memory energy consumption. As can be seen in Figure 14, both `GoogLeNet` and `DenseNet` have high transient data to weight ratio. Furthermore, in `GoogLeNet`, due to the presence of 1x1 helper layers within the inception module, it allows for dimensionality reduction in the 3x3 and 5x5 CONV layers, thereby reducing the required number of computations. Similarly, `DenseNet` has a 1x1 reduction layer prior to every 3x3 CONV layers to help keep its computations low. On the other hand, `ResNet-18` and `Resnet-34` are mostly made up of 3x3 CONV layers without dimensionality-reducing layers and have low transient data to weight ratio. These reasons them to have lower energy-efficiency gains when employing our proposed schemes.

Figure 16b shows the normalized main memory energy consumption for the benchmarks. Figure 16c shows the breakdown of the energy savings obtained by CASH arising from compiler technique and hardware mechanism, separately. We observe that within CASH, savings due to compiler increases as the network depth increases. This is because, with increased depth, reuse opportunities of an allocated buffer increases, accounting for higher efficacy of compiler technique.

Impact on Non-Volatile Memory: In recent years, non-volatile memories (NVM), specifically PCM have been proposed to replace DRAMs [29], due to their higher density and lack of refresh energy overheads. However, they suffer from the issue of cell wear-out due to writes and high write energy. Our proposed scheme reduces average number of read and write transactions by 34% and 72.5%, respectively (averaged from Figure 15). Therefore, we believe that NVM-based systems will greatly benefit from our proposed mechanism in terms of their lifetime as a direct result of the reduced number of writes [14].

Discussion on the CNN overflow issue: Our experimental observation reveals that the transient data in the evaluated set of workloads *did not* overflow out of the cache hierarchy. Note that, our workload consists of state-of-the-art CNNs which consist of a variety of network topologies. We argue that, the transient data are frequently reused before its lifetime gets over, and with the LRU replacement policy in place, the transient data gets retained in the cache subsystem. In general, an intermediate layer with tensor dimension $\{c, h, w\}$ and weight dimension of $\{m, c, k, k, s\}$ causes $m * k/s * k/s \equiv O(m * k^2)$ reuses of each input tensor point, leading to high data reuse and cache retention.

Area, energy and performance overheads: Recall that our proposed scheme consists of two additional hardware components: (1) Transient Data Detection Table (TDDT) and (2) Lookup logic. TDDT is a set of register files that contains the physical address of the intermediate data along with the size of the data structure and the read_disable and writeback_disable bits. The lookup table is a 4 input MUX logic, which decides the required action on incoming memory request. Considering 32 bit physical addresses, a maximum memory allocation size of 30 bits, and 1 bit for each flag bits, the required overhead is 64 bits per TDD entry. Using *hardware only per-layer tracking* would lead to significant overheads given the large number of network layers. However, our refined compiler approach identifies the minimum buffer size, and re-uses it across



(a) Main memory bandwidth reduction. (b) Normalized DRAM energy usage. (c) Breakdown of energy savings.

Figure 16: End-to-end normalized memory bandwidth reduction and energy savings of proposed mechanisms

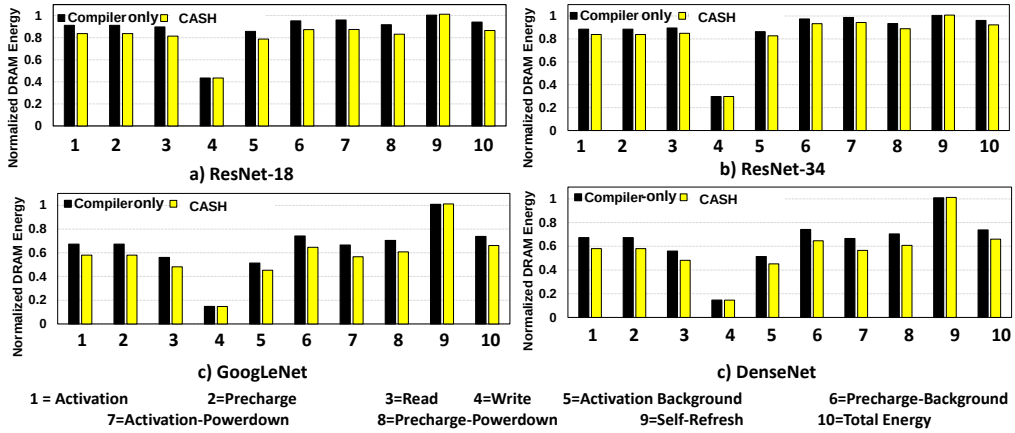


Figure 17: Different components of main memory energy usage normalized to the baseline

layers, which limits the number of TDDT entries and comparisons at the memory controller to only one per inference (two considering dynamic re-sizing of buffer size). Therefore, the maximum size of TDDT is 16 bytes, while the lookup table requires only 2 bytes of storage. Therefore, the overall overhead for area, energy and performance are negligible.

7 RELATED WORK

CPU optimizations. The issue of unwanted reads and write-backs of transient data can be resolved using prior techniques such as Scratch pads [2] are software managed on-chip memory that are used when the access pattern is known and the programmer explicitly mentions the data transfer between scratch pad and main memory. However, scratch pads have several drawbacks that have limited their widespread adoption in a high-performance CPU-based computing systems. First, scratch pad management is completely handled by the programmer, which limits programming flexibility. Second, scratch pads are usually in private address space and cannot take part in coherence. Third, context switching overheads increase significantly as the scratch pad data has to be a part of the context as well. Co-operative cache scrubbing [38] mechanism relies on garbage collector threads to perform variable lifetime analysis. This works only for managed languages that require a virtual-runtime system for applications to run. Core-initiated invalidation can also resolve the issues. Both the above techniques are not suitable for the context of CNN inference as they take up CPU cycles to perform their optimizations, which can potentially hamper performance as the CNN inferences are compute-intensive and QoS bound. Our proposed mechanisms leverages the layer wise structure of CNNs and uses liveness analysis and a simple hardware unit to identify and eliminate any unnecessary reads and write-backs to the main memory with negligible overheads.

Accelerators, PIMs and FPGAs. There has been a huge body of work that has been dedicated on developing domain specific accelerators for CNN based ML tasks such as [4–7, 12, 30, 37]. An important characteristic of any such accelerator is that address space used to handle intermediate data is entirely local, thereby eliminating constraints towards maintaining consistency with an external memory. In other words, the issue of read allocate due to write misses and write-back doesn't exist in the accelerator context, as long as it is designed with sufficient local memory to handle the computations. Due to the large volume of data movement involved in CNN operations, multiple PIM-based architectures have been proposed as well [8, 26, 39]. While these PIM architectures implicitly handle the issues arising out of memory consistency (no cached copies of data), we believe our work can still be useful under general purpose processors context until such platforms are deployed in practice. Furthermore, our compiler-only optimizations can be readily deployed without any hardware changes.

FPGAs have been used for custom CNN engine deployment as well. In Fused Layer CNN [1], computations are staged such that intermediate data is consumed as soon as it is generated in order to avoid costly overflows to the FPGA block RAMs. This essentially tries to limit the amount of intermediate data to as minimal as possible, by immediately staging consumption of the generated data. However, there are serious limitations in terms of performing cross

layer computations in a general purpose platform like CPU, most notably the high synchronization overheads due to fine-grained data dependencies. vDNN [36] is a data prefetching mechanism between CPU and GPU memories in the context of training. Note that, all the intermediate data structures need to remain persistent during the training phase of execution; hence, our proposed techniques are not applicable as they rely on transience of data layer values. For inferences, GPUs are equipped with large register files and local scratchpad memories, which can hide the intermediate data from the memory hierarchy. Applicability and extension of our proposed approach in GPUs is left for future exploration.

Compression and pruning techniques. There is a substantial body of work focused on the compression and pruning of ML models to reduce the compute and memory footprints. Prior techniques such as Deep Compression [17] and WRPN [32] enable weights to be stored on an on-chip SRAM and also allow for faster reduced precision computations. Note that our proposed mechanisms are effective on compressed and pruned ML models as well. As the footprint of weights reduces, it reduces computations. Therefore, the impact of reduced memory transactions become more apparent.

8 CONCLUSION

Deep learning applications have grown in significance over the years. While there have been several proposals on different type of accelerator designs for efficient execution of CNNs, CPUs remain as the most widely used compute units for executing inferences. Therefore, it is imperative to optimize inference execution on CPUs to allow further energy efficiency scaling. In this paper, we propose CASH, a compiler assisted hardware technique to eliminate unnecessary data movement during the course of CNN inference execution on CPUs. Our experimental evaluations on four state-of-the-art CNN benchmarks show that by avoiding unwanted memory transactions, it can reduce average memory bandwidth consumption and total main memory energy usage by ∼40% and ∼18%, respectively. Due to the widespread popularity and deployment of CNNs, our optimizations can have significant impact on the energy consumption of CPU-based inference devices.

ACKNOWLEDGMENTS

We would like to thank Asit K Mishra and other anonymous reviewers for their feedback and inputs. This research is supported in part by NSF grants #1317560, #1763681, #1629129 and #1629915.

REFERENCES

[1] Manoj Alwani, Han Chen, Michael Ferdman, and Peter Milder. 2016. Fused-layer CNN accelerators. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Press, 22.

[2] Rajeshwari Banakar, Stefan Steinke, Bo-Sik Lee, Mahesh Balakrishnan, and Peter Marwedel. 2002. Scratchpad memory: A design alternative for cache on-chip memory in embedded systems. In *Hardware/Software Codesign, 2002. CODES 2002. Proceedings of the Tenth International Symposium on*. IEEE, 73–78.

[3] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. 2011. The gem5 simulator. *ACM SIGARCH Computer Architecture News* 39, 2 (2011), 1–7.

[4] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. 2014. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. *ACM Sigplan Notices* 49, 4 (2014), 269–284.

[5] Yunji Chen, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. 2016. DianNao Family: Energy-efficient Hardware Accelerators for Machine Learning. *Commun. ACM* 59, 11 (Oct. 2016), 105–112. https://doi.org/10.1145/2996864

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

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

[8] Ping Chi, Shuangchen Li, Cong Xu, Tao Zhang, Jishen Zhao, Yongpan Liu, Yu Wang, and Yuan Xie. 2016. Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory. In *ACM SIGARCH Computer Architecture News*, Vol. 44. IEEE Press, 27–39.

[9] Inc. CISCO. [n.d.]. Cisco's Global Cloud Index. https://www.cisco.com/c/en/us/solutions/collateral/service-provider/global-cloud-index-gci/white-paper-c11-738085.html.

[10] Dipankar Das, Sasikanth Avancha, Dheevatsa Mudigere, Karthikeyan Vaidynathan, Srinivas Sridharan, Dhiraj Kalamkar, Bharat Kaul, and Pradeep Dubey. 2016. Distributed deep learning using synchronous stochastic gradient descent. *arXiv preprint arXiv:1602.06709* (2016).

[11] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, et al. 2012. Large scale distributed deep networks. In *Advances in neural information processing systems*. 1223–1231.

[12] Zidong Du, Robert Fasthuber, Tianshi Chen, Paolo Ienne, Ling Li, Tao Luo, Xiaobing Feng, Yunji Chen, and Olivier Temam. 2015. ShiDianNao: Shifting vision processing closer to the sensor. In *ACM SIGARCH Computer Architecture News*, Vol. 43. ACM, 92–104.

[13] Clément Farabet, Berin Martini, Benoit Corda, Polina Akselrod, Eugenio Culurciello, and Yann LeCun. 2011. Neuflow: A runtime reconfigurable dataflow processor for vision. In *Computer Vision and Pattern Recognition Workshops (CVPRW), 2011 IEEE Computer Society Conference on*. IEEE, 109–116.

[14] Alexandre P Ferreira, Miao Zhou, Santiago Bock, Bruce Childers, Rami Melhem, and Daniel Mossé. 2010. Increasing PCM main memory lifetime. In *Proceedings of the conference on design, automation and test in Europe*. European Design and Automation Association, 914–919.

[15] Arpan Gujarati, Sameh Elnikety, Yuxiong He, Kathryn S McKinley, and Björn B Brandenburg. 2017. Swayam: distributed autoscaling to meet SLAs of machine learning inference services with resource efficiency. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*. ACM, 109–120.

[16] Samuel Z Guyer, Kathryn S McKinley, and Daniel Frampton. 2006. Free-me: a static analysis for automatic individual object reclamation. In *ACM SIGPLAN Notices*, Vol. 41. ACM, 364–375.

[17] Song Han, Huizi Mao, and William J Dally. 2015. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149* (2015).

[18] Swapnil Haria, Mark D Hill, and Michael M Swift. 2018. Devirtualizing memory in heterogeneous systems. In *ACM SIGPLAN Notices*, Vol. 53. ACM, 637–650.

[19] Johann Hauswald, Yiping Kang, Michael A Laurenzano, Quan Chen, Cheng Li, Trevor Mudge, Ronald G Dreslinski, Jason Mars, and Lingjia Tang. 2015. DjiNN and Tonic: DNN as a service and its implications for future warehouse scale computers. In *ACM SIGARCH Computer Architecture News*, Vol. 43. ACM, 27–40.

[20] Kim Hazelwood, Sarah Bird, David Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, Aditya Kalro, et al. 2018. Applied Machine Learning at Facebook: A Datacenter Infrastructure Perspective. In *High Performance Computer Architecture (HPCA), 2018 IEEE International Symposium on*. IEEE, 620–629.

[21] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Identity mappings in deep residual networks. In *European Conference on Computer Vision*. Springer, 630–645.

[22] John L Hennessy and David A Patterson. 2011. *Computer architecture: a quantitative approach*. Elsevier.

[23] Gao Huang, Zhuang Liu, Kilian Q Weinberger, and Laurens van der Maaten. 2017. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, Vol. 1. 3.

[24] Forrest N Iandola, Matthew W Moskewicz, Khalid Ashraf, and Kurt Keutzer. 2016. Firecaffe: near-linear acceleration of deep neural network training on compute clusters. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2592–2600.

[25] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. 2017. In-datacenter performance analysis of a tensor processing unit. In *Computer Architecture (ISCA), 2017 ACM/IEEE 44th Annual International Symposium on*. IEEE, 1–12.

[26] Duckhwan Kim, Jaeha Kung, Sek Chai, Sudhakar Yalamanchili, and Saibal Mukhopadhyay. 2016. Neurocube: A programmable digital neuromorphic architecture with high-density 3D memory. In *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*. IEEE, 380–392.

[27] Alex Krizhevsky. 2014. One weird trick for parallelizing convolutional neural networks. *arXiv preprint arXiv:1404.5997* (2014).

[28] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*. 1097–1105.

[29] Benjamin C Lee, Ping Zhou, Jun Yang, Youtao Zhang, Bo Zhao, Engin Ipek, Onur Mutlu, and Doug Burger. 2010. Phase-change technology and the future of main memory. *IEEE micro* 30, 1 (2010), 143–143.

[30] Daofu Liu, Tianshi Chen, Shaoli Liu, Jinhong Zhou, Shengyuan Zhou, Olivier Teman, Xiaobing Feng, Xuehai Zhou, and Yunji Chen. 2015. Pudiannao: A polyvalent machine learning accelerator. In *ACM SIGARCH Computer Architecture News*, Vol. 43. ACM, 369–381.

[31] Ankush Mandal, Rajkishore Barik, and Vivek Sarkar. 2018. Using Dynamic Compilation to Achieve Ninja Performance for CNN Training on Many-Core Processors. In *European Conference on Parallel Processing*. Springer, 265–278.

[32] Asit Mishra, Eriko Nurvitadhi, Jeffrey J Cook, and Debbie Marr. 2017. WRPN: wide reduced-precision networks. *arXiv preprint arXiv:1709.01134* (2017).

[33] Steven Muchnick. 1997. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers.

[34] Jongsoo Park, Maxim Naumov, Protonu Basu, Summer Deng, Aravind Kalaiah, Daya Khudia, James Law, Parth Malani, Andrey Malevich, Satish Nadathur, et al. 2018. Deep Learning Inference in Facebook Data Centers: Characterization, Performance Optimizations and Hardware Implications. *arXiv preprint arXiv:1811.09886* (2018).

[35] Geoff Pleiss, Danlu Chen, Gao Huang, Tongcheng Li, Laurens van der Maaten, and Kilian Q Weinberger. 2017. Memory-efficient implementation of densenets. *arXiv preprint arXiv:1707.06990* (2017).

[36] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W Keckler. 2016. vDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Press, 18.

[37] Anup Sarma, Soubhagya Sutar, Vijay Kumar Sharma, and Kamala K Mahapatra. 2011. An ASIP for image enhancement applications in spatial domain using LISA. In *2011 International Conference on Recent Trends in Information Systems*. IEEE, 175–179.

[38] Jennifer B. Sartor, Wim Heirman, Stephen M. Blackburn, Lieven Eeckhout, and Kathryn S McKinley. 2014. Cooperative Cache Scrubbing. *ACM International Conference on Parallel Architecture and Compiler Techniques (PACT)*.

[39] Ali Shafiee, Anirban Nag, Naveen Muralimanohar, Rajeev Balasubramonian, John Paul Strachan, Miao Hu, R Stanley Williams, and Vivek Srikumar. 2016. ISAAC: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars. *ACM SIGARCH Computer Architecture News* 44, 3 (2016), 14–26.

[40] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[41] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, Andrew Rabinovich, et al. 2015. Going deeper with convolutions. CVPR.

[42] Jiecao Yu, Andrew Lukefahr, David Palframan, Ganesh Dasika, Reetuparna Das, and Scott Mahlke. 2017. Scalpel: Customizing dnn pruning to the underlying hardware parallelism. In *ACM SIGARCH Computer Architecture News*, Vol. 45. ACM, 548–560.

[43] Minjia Zhang, Samyam Rajbhandari, Wenhan Wang, and Yuxiong He. 2018. Deepcpu: Serving rnn-based deep learning models 10x faster. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*. 951–965.

[44] Aleksandar Zlateski, Kisuk Lee, and H Sebastian Seung. 2016. ZNN–A Fast and Scalable Algorithm for Training 3D Convolutional Networks on Multi-core and Many-Core Shared Memory Machines. In *Parallel and Distributed Processing Symposium, 2016 IEEE International*. IEEE, 801–811.