REGENT: A Heterogeneous ReRAM/GPU-based Architecture Enabled by NoC for Training CNNs

Biresh Kumar Joardar*, Bing Li†, Janardhan Rao Doppa*, Hai Li†, Partha Pratim Pande*, Krishnendu Chakrabarty†

*School of EECS, Washington State University
Pullman, WA 99164, U.S.A.
{biresh.joardar, jana.doppa, pande}@wsu.edu

†Department of ECE, Duke University
Durham, NC, USA.
{bing.li.ece, hai.li, krishnendu.chakrabarty}@duke.edu

***Abstract*— The growing popularity of Convolutional Neural Networks (CNNs) has led to the search for efficient computational platforms to enable these algorithms. Resistive random-access memory (ReRAM)-based architectures offer a promising alternative to commonly used GPU-based platforms for CNN training. However, backpropagation in CNNs is susceptible to the limited precision of ReRAMs. As a result, training CNNs on ReRAMs affects the final accuracy of learned model. In this work, we propose REGENT, a heterogeneous architecture that combines ReRAM arrays with GPU cores, and exploits the benefits provided by 3D integration along with a high-throughput yet energy efficient Network-on-Chip (NoC) for training CNNs. We also propose a bin-packing based framework that maps CNN layers and then optimize the placement of computing elements to meet the targeted design objectives. Experimental evaluations indicate that REGENT improves full-system EDP by 55.7% on average compared to conventional GPU-only platforms for training CNNs.**

I. Introduction

Deep learning technology is employed today in many real-world applications. Convolutional Neural Networks (CNNs) are one of the most popular techniques within the suite of deep learning tools, and they are often utilized for computer vision and image/text processing tasks. Therefore, the design of efficient hardware architectures to support CNN training is very important and is the focus of this work.

Emerging manycore processing platforms consisting of CPUs, GPUs, and accelerators are the most preferred choice of hardware for training CNNs [1][2]. CNN training involves many vector and matrix computations [1], which can be performed in parallel on GPUs, resulting in significant reduction in runtime compared to traditional CPU-based implementations. However, general-purpose GPU cores are not customized for running only CNN based applications and are often bottlenecked by higher area, insufficient memory access bandwidth, and higher power consumption [3]. Domain-specific customization is necessary to ensure efficient CNN training with large datasets.

Metal-oxide resistive random-access memory (ReRAM) is another popular choice for implementing high-performance architectures for training CNNs [4]. ReRAM crossbars can perform efficient matrix-vector multiplication, which forms the core computational element of CNN training [5]. ReRAMs are more energy and area efficient compared to their GPU counterparts [3][4]. However, ReRAM-only architectures for training CNNs have one major limitation: ReRAMs have low precision, which affects the accuracy of learned models. Specifically, backpropagation algorithm to train CNNs is sensitive to precision of weights and data [4][6]. Hence, despite its advantages, training CNNs on ReRAMs negatively affects accuracy [6] and addressing this shortcoming is necessary.

Moreover, CNN training involves frequent memory access to fetch inputs and weights along with data exchange between neurons in adjacent layers [2][7]. Without adequate Network-on-Chip (NoC) enabled architectural support, this can lead to significant performance bottlenecks due to likely network congestion [2][7]. Therefore, an appropriate computational platform for training CNNs must have a) efficient processing units to accelerate the large number of matrix multiplication-and-accumulation (MAC) operations in each layer; b) high-precision data representation to maintain the accuracy of backpropagation; and c) effective NoC architecture as the communication backbone to reduce the data-transfer and memory access overheads. In this paper, we propose REGENT: A heterogeneous ReRAM/GPU-based Processing-in-Memory (PIM) architecture enabled by 3D integration and an efficient NoC architecture as the communication backbone to reduce the overhead due to frequent data movement for efficient training of CNNs. The main contributions of this work include:

- We demonstrate the efficacy of a heterogeneous-ReRAM/GPU architecture enabled by an efficient NoC and 3D integration for training diverse CNNs.
- We undertake an in-depth study of the traffic patterns generated by different CNNs (and layers) when executed on GPU-based platforms to design an energy-efficient and high-performance NoC architecture.
- We introduce a bin-packing based algorithm that maps CNN layers to available computational units and then optimize their placements to accelerate training.

II. Related Work

As the focus of this paper is heterogeneous ReRAM/GPU architectures for accelerating CNN training, we mainly focus on related work covering these two broad themes:

A. ReRAM based architectures for CNNs

ReRAMs can be employed as memory as well as to perform in-situ MAC operations [5] making it an ideal candidate for PIM. ReRAM-based PIM architectures have been proposed for accelerating both inference [8][9] as well as training of CNNs [4]. ReRAM-based PIM designs offer limited data precision [4]. Multi-Layer Perceptron (MLP) and CNN inference can be executed with reasonable accuracy using reduced data precision [4][9]. However, CNN training involves execution of the backpropagation algorithm, which is sensitive to data precision [6]. Therefore, ReRAM-only architectures for training CNNs suffer from undesirable accuracy loss [4][6]. Moreover, accuracy preserving techniques (*e.g.*, stochastic rounding) for low precision CNN training require normalization [10], which cannot be implemented on ReRAMs [8].

B. GPU-based architectures for CNNs

Due to high data-parallelism in CNNs, use of GPU-based platforms have become widespread [1][2]. GPU-based architectures show some interesting characteristics: a) there is

This work was supported, in part by the US National Science Foundation (NSF) grants CNS-1564014, CCF 1514269, CSR-1717885 and USA Army Research Office grant W911NF-17-1-0485.

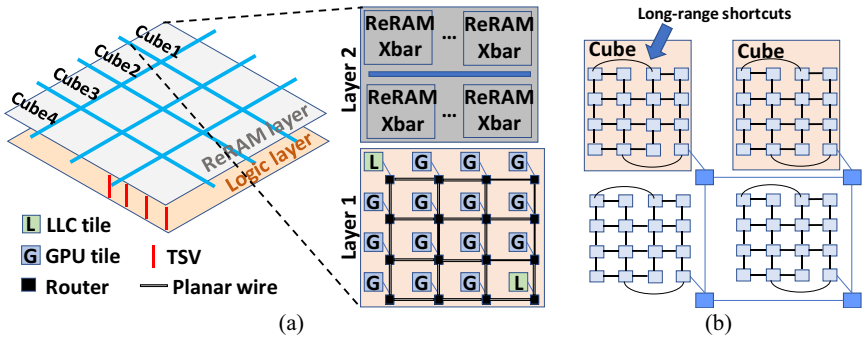


Fig 1: Overview of (a) REGENT architecture (b) two-level Hierarchical NoC (*HNoC*). This figure is for illustration only; The tile placements and NoC design have not been optimized here)

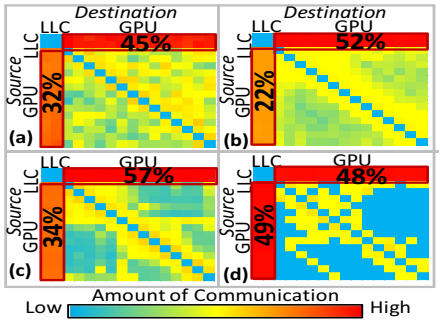


Fig. 2: Heat map of traffic patterns for (a) Conv2 (LeNet), (b) Conv1 (CDBNet), (c) Conv1 (MattNet), (d) FC2 (AlexNet)

low inter-GPU communication [2][11], and b) GPUs mostly communicate with the few shared Last-Level Caches (LLCs), causing a many-to-few traffic pattern [2][11]. This creates network congestion closer to LLCs [2]. Considering these observations, application-specific NoC design for CNNs has been explored [2]. However, the NoC design framework presented in [2] does not leverage the similarities in traffic generated by various CNN layers.

Prior work on PIM-based ReRAM architectures mostly target CNN inference. However, training CNNs is more complicated and resource-intensive as it involves repeated weight updates and data dependencies. In this work, we advance the state-of-the-art by proposing a ReRAM/GPU-based PIM architecture: REGENT, that incorporates a) the speed and energy-efficiency of ReRAM based MAC engines; b) high-precision data computing on GPUs; c) efficient memory access via vertical links (as compute and memory layers are stacked vertically); (d) an efficient NoC architecture connecting the compute elements for training CNNs.

III. The REGENT Architecture

In this section, we first discuss relevant properties of CNN training that are crucial in designing an efficient manycore architecture. Next, we provide an in-depth study of communication patterns between computing elements when different CNNs are executed on GPUs to design an efficient NoC. Finally, we describe a bin-packing-based task allocation algorithm to map CNN layers, and optimize the placement of processing units for high performance.

A. Training CNNs

A CNN consists of different types of layers, e.g., convolution (Conv), fully-connected (FC), etc. Training CNNs involve two major phases: feed forward (FF) and backpropagation (BP). In FF, the input passes through the layers to make a prediction. During BP, the prediction error relative to the ground truth is computed and back-propagated to update the weights. The BP phase is sensitive to precision and affects the accuracy of the learned model [6]. Motivated by these observations, we propose REGENT, a high-performance and energy-efficient heterogeneous ReRAM-GPU architecture specifically targeted for CNN training (shown in Fig. 1). REGENT consists of two-layers, with the lower layer (*logic layer*) consisting of GPUs and LLCs. The logic layer is connected to the upper ReRAM layer (*memory layer with additional compute capability*) via vertical links implemented using through-silicon vias (TSVs). In this architecture, the ReRAM layer acts as the Processing-in-Memory (PIM) module. The 3D interconnects enable energy efficient and high-bandwidth memory access required by the GPUs [2][12]. REGENT implements the pipelined-CNN described in [4] to reduce the buffering requirements and execution stalls due to frequent re-programming of ReRAM crossbars needed in conventional CNN implementations [4][13]. To prevent loss in accuracy of the learned model, the FF phase is executed completely on the ReRAMs, whereas the precision-sensitive BP is executed on GPUs. The PIM tier in REGENT (Fig. 1), is composed of two types of ReRAM arrays: morphable subarrays and memory subarrays [9]. The morphable subarrays can be configured for both storage (conventional memory) and computation (in-situ MAC), as necessary. The memory subarrays are only used for storage purposes (input, output, intermediate results). The bottom layer consists of LLCs and conventional GPUs connected via NoC.

Communication in CNN training is limited between the neurons connected in adjacent layers and the corresponding memory accesses to fetch data. It has been demonstrated that under such traffic, a large fraction of links in conventional NoCs e.g. mesh, remain under-utilized [2][7]. Instead, a hierarchical two-level NoC is more latency and energy-efficient [7]. To implement the hierarchical NoC, REGENT is divided into multiple cubes (logically) as shown in Fig. 1. Each cube is identical to the other in terms of computing and storage resources. Due to high data parallelism in CNNs, the compute units in each cube should be connected using an NoC that is optimized for throughput (first level). Communication between individual cubes is handled by the second level NoC (Fig. 1(b)). In what follows, we further elaborate some of the key design aspects associated with the REGENT architecture.

B. Traffic patterns for training CNNs

In this section, we study the behavior of different CNNs (and individual CNN layers) when they are executed on GPU platforms to design a suitable NoC architecture. Fig. 2 shows the traffic patterns for different CNN layers, belonging to four distinct CNNs: LeNet [14], CDBNet (for CIFAR-10) [15], AlexNet [16], and MattNet [17] when executed on a GPU-based system, for every source-destination pair. For the sake of brevity, we consider the traffic pattern of *Conv2* (LeNet), *Conv1* (CDBNet), *Conv1* (MattNet), and *FC2* (AlexNet). The numbers in Fig. 2 indicate the percentage of traffic contributed by the GPU-LLC (and vice-versa) communication. From Fig. 2, we note some interesting observations: (a) GPUs communicate heavily and (nearly) uniformly with few LLCs creating many-to-few traffic (nearly 85% of total traffic on average); and (b) communication between same types of cores (e.g., GPU-GPU) is almost negligible. These trends are observed for other layers

and CNNs as well. From Fig. 2, it is clear that inter-layer and inter-CNN variations are insignificant compared to the visibly dominant many-to-few LLC traffic (red/crimson patches in Fig. 2). Therefore, we can design a generic NoC architecture by primarily considering the many-to-few traffic rather than depending on any specific layer of CNN or specific CNN, without significant loss in performance.

C. NoC design framework

In order to efficiently handle the frequent data-transfers and memory accesses involved in CNN training, a high throughput NoC is necessary to deliver the large volumes of data. Here, the NoC throughput is improved by load-balancing the network to allow more messages to utilize the network resources better [2]. This reduces the amount of contention for heavily utilized links. As a result, links are more readily available and network is less congested, leading to improved throughput. Moreover, an NoC should remain within a given energy budget without sacrificing performance. Therefore, performance-energy joint optimization formulation is necessary for efficient training of CNNs. The total network energy (E) is the sum of router and link energy determined based on individual link/router utilization as follows.

$$E = \sum_{i=1}^{N}\sum_{j=1}^{N} f_{ij} \cdot \left(\sum_{k=1}^{L} p_{ijk} \cdot d_k \cdot \mathrm{E}_{link} + \sum_{k=1}^{R} r_{ijk} \cdot E_r \cdot P_k \right) \quad (1)$$

where N is the number of cores, R and L denote the total number of routers and links respectively, d_k represents the physical length of link k, E_{link} and E_r denote the average link energy per unit length and router logic energy per port respectively and P_k denotes the number of ports at router k. Both p_{ijk} and r_{ijk} are binary variables that indicate whether a link/router k is used to communicate between core i and core j respectively. The parameter f_{ij} denotes the frequency of communication between cores *i* and *j*. Overall, our aim is to design an NoC that increases throughput and reduces energy consumption for the REGENT architecture. Therefore, NoC design can be formulated as a multi-objective optimization (MOO) problem as follows:

$$D^* = MOO(D, OBJ = \{Throughput(d), Energy(d)\}) \quad (2)$$

Here, D^* is the set of pareto optimal NoC designs for REGENT, *MOO* stands for the multi-objective optimization solver, and *OBJ* is the set of all objectives to evaluate a candidate design $d \in D$. For a fair comparison, each candidate solutions $d \in D$ has the same number of links as in an equivalent mesh NoC. It should be noted that other objectives can also be included in (2) as per design requirements. We optimize the network by perturbing the placement of cores (GPUs and LLCs), and links to explore the trade-offs between throughput and energy. Here, we employ the archived multi-objective simulated annealing (AMOSA [19]) as the MOO solver to determine the optimal NoC architecture D^*. Without loss of generality, other MOO solvers can also be used for optimization.

D. Mapping CNN layers to cubes

In this section, we present a bin packing-based framework to allocate CNN layers to computing units (cubes with GPUs and ReRAMs) for efficient training. Unlike inference, where weights are only written once at the start and never changed, training is more challenging as it involves repeated weight updates and complex data dependencies [4]. Furthermore, to preserve accuracy, the layers in forward pass are executed on ReRAMs, whereas the precision sensitive layers in backpropagation are executed on GPUs (Section III.A). The same set of network weights and data, local to each layer, is used in both passes of the iterative training procedure (forward and backward). Therefore, the *mapping must also preserve spatial locality for layers between forward and backward passes*. For example, if forward convolution 1 (*Conv1*) of a CNN is mapped to ReRAMs on *cube1* (Fig. 1(a)), then weights and data for *Conv1* are stored in *cube1*. Therefore, mapping *Conv1* during backpropagation on the GPUs of *cube1* enables faster and high-bandwidth memory access via the vertical links. Mapping to any other cube, e.g., *cube3*, requires the use of inter-cube links for memory access. In that case, we would fail to fully exploit the benefits of 3D integration as the memory access latency is bottlenecked by the planar links connecting *cube1* to *cube3*. This happens as data from *cube1* requires extra cycles to reach *cube3* due to longer path lengths as opposed to local intra-cube communication. Therefore, there exists a strong dependency between mapping of CNN layers to cubes in the two passes.

We develop a bin-packing based solution aiming to not only reduce the network traffic but also address the dependencies between the two passes during the iterative training procedure. Bin packing is a widely studied problem with applications to the logistics and other domains, where *items* are placed in *bins* based on their *size* to meet given *objective(s)*. The *size* is used to characterize the *item*. Our proposed formulation allocates CNN layers (*items*) to the computing cubes (*bins*). The goal is to utilize the computing cubes (*bins*) efficiently to minimize the overall execution time. The overall runtime of a pipelined-CNN implementation is bottlenecked by the slower layers. To address this, we follow similar approach as in [4], which proposes to allocate more resources to increase the parallelism of these layers. The amount of resources allocated (*size*) to each layer is used to characterize the layer (*item*) in the proposed formulation.

Compared to the classical problem setting, the use of bin-packing to allocate CNN layers has some key differences that need to be considered: (a) resources allocated for a CNN layer can be greater than the maximum amount of computing resources in one cube (*size of item* > *size of bin*). Traditional bin-packing problems assume that *items* are smaller than *bins;* (b) a single CNN layer (*item*) can be fragmented and placed on multiple cubes (*bins*) as MAC operations within a layer are often independent of each other [7]; (c) mapping of CNN layers (Placing *items*) during the backward pass is influenced by the mapping in the forward pass. The idea of fragmenting *items* across multiple *bins* has been studied as a variant of the classical bin packing problem [20]. However, contrary to the problem setting of [20], our case has an additional constraint between the placements associated with the two passes of the CNN. Specifically, allocating CNN layers (*items*) to cubes (*bins*) in the *backward* pass presents a variant of bin packing that includes *item-fragmentation* with an additional *placement constraint*.

Considering the above differences, we develop the overall mapping methodology. In mapping of forward pass, it is important to map layers (*items*) to cubes (*bins*) with minimal fragmentation of layers to reduce the amount of multicast communication inherent in CNNs [7]. For this purpose, we adopt the First-Forward Decreasing (FFD) algorithm to map layers to the cubes with minimal fragmentation [20]. FFD is a greedy algorithm to place *items* in *bins* in the decreasing order of *size*. Note that any other bin-packing algorithm can also be seamlessly integrated in to our formulation. The FFD implementation for forward pass first attempts to map a layer

Algorithm 1: Bin-packing based mapping (Backward pass)

Input: num_GPU[.]: No. of GPUs required for each layer

Output: Layer mapping to cubes (GPUs)

Variable: priority (priority of mapping layers)

Initialize: priority ← preferences following (3)

Algorithm:

1 ***SORT*** (num_*GPU[.]*) in descending order
2 **For** every *GPU required* in num_GPU[.]:
3 **while** all *GPU required* not allocated:
4 ***FIND_BEST_CUBE*** (*priority*, *current layer*)
5 ***MAP*** (*GPU required*) on *best cube*
6 Update *GPU required*
7 Update *priority*
8 **return** mapping

(*item*) to cubes (*bins*) with sufficient available computational resources (ReRAMs) without having to fragment the layers. If no such cube (*bin*) exists, then the algorithm fragments the layer (*item*) and reattempts to allocate the fragments to the cubes (*bins*). The mapping is repeated until all layers (*items*) have been allocated in decreasing order of ReRAM requirements (*size*).

However, as noted earlier, to reduce traffic in the network, and fully utilize the benefits of 3D integration, the mapping in backward pass (Algorithm 1) is dependent on the solution of forward pass. Ideally, we would like to have minimum discrepancy between the mappings for forward and backward passes. We adopt a priority-based mapping to capture this dependency using the FFD algorithm. Here *priority* is defined as the preference of mapping a layer onto a cube. For a layer l mapped to a total of N_l cubes in the forward pass, the priority of mapping during backward computation to a cube k ($1 \leqslant k \leqslant p$, where p is the available number of cubes) is defined as:

$$Priority_{l,k} = \left\{ \begin{matrix} \frac{1}{N_l}, \ if\ l\ is\ mapped\ to\ k \\ 0, \ otherwise \end{matrix} \right\} \forall \text{ cube } k \qquad (3)$$

For each layer, the best cube is chosen based on the priority (and availability) (Line 4). ***MAP****(x)* (Line 5) maps the layer to the *best cube*. The main objective here is to map each layer to its preferred cubes while *minimizing* conflict with other layers. For mapping layer l on a cube k, we define conflict as:

$$Conflict_{l,k} = \max_i \{Priority(i,k)\} \ \forall \text{ layer } i \quad (i \neq l) \qquad (4)$$

A conflict occurs if two or more layers have non-zero priority for a given cube. In other words, if multiple layers are mapped (completely or partially) on a certain cube during forward pass, then each of these layers prefer to be mapped to the same cube in backward pass as well, leading to a conflict. The ***FIND_BEST_CUBE*** (*priority*, *layer*$_i$) (Line 4) attempts to pick the *best cube* for mapping *layer*$_i$ based on priority that has minimum conflict from other layers. After mapping to *best cube*, the priorities are dynamically updated (Line 7) based on the new availability of computing resources. This not only preserves spatial locality among layers in both passes but also ensures that smaller layers get the opportunity to be mapped to their preferred cubes (as much as possible). This eventually leads to reduction in network traffic, enables low latency and high-bandwidth access to 3D stacked memory via the vertical links. The performance is further improved by placing the mapped cubes and available links based on their amount of communication as discussed in Section III.B

TABLE 1: System Parameters

Component	Parameters
Logic Layer (GPU, LLC)	GPU: 700MHz, 64kB L1/core, Maxwell architecture, LLC: 512KB each
ReRAM (PIM)	Latency: read 29.31ns, write 50.88ns Energy/bit: read 1.08pJ, write 3.91pJ

IV. Experimental Results

In this section, we present the detailed performance evaluation considering diverse well-known CNNs.

A. Experimental Setup

Table 1 summarizes the relevant system parameters for the GPU and the ReRAM-based PIM used in this work. We employ Gem5-GPU [18], a full system simulator to obtain network and processor level information. We modified the Garnet network within Gem5-GPU to implement the different NoC topologies. We follow the MESI two-level protocol for cache coherence. Each GPU core has its own private L1 cache. The LLCs are shared among all the cores. The ReRAM-based PIM accelerator configuration is similar to [3]. We consider ReRAM cells with 4-bit accuracy and therefore four arrays are combined to realize 16bit fixed-precision computing. Overall, the system has 16 cubes (Fig. 1(a)) that are identical to each other (14 GPUs + 2 LLC tiles in logic layer and equal number of ReRAM crossbars in ReRAM layer). The number of ReRAMs in each cube is obtained based on the area estimates of the crossbar along with the peripheral circuitry reported in [8]. All ReRAM crossbars in *same* cube are connected to a shared bus while *inter-cube* ReRAM communication happen via the NoC. For experimental evaluation, we choose four well-known CNNs: LeNet [14], CDBNet for CIFAR-10 [15], AlexNet [16], and MattNet [17].

B. Performance Evaluation

Based on the discussion presented in Section III.B, we first show that an NoC optimized for any layer (capturing the many-to-few traffic in general) can show similar performance for other CNN layers (and the CNN overall) as well. For our experiments, we choose a 16-tile system (14-GPU, 2-LLC), representing a single REGENT cube, connected via the optimized NoC following the framework in Section III.C. We consider Energy-Delay Product (EDP) as the relevant performance metric since it captures the joint effect of both latency and energy.

NoC design for many-to-few traffic: Fig. 3 shows the results when different layers (from same/different CNNs) are run on the NoC optimized for just one layer. Fig. 3(a) shows the case where the NoC was optimized for one layer and then employed to execute other layers belonging to same CNN (intra-CNN).

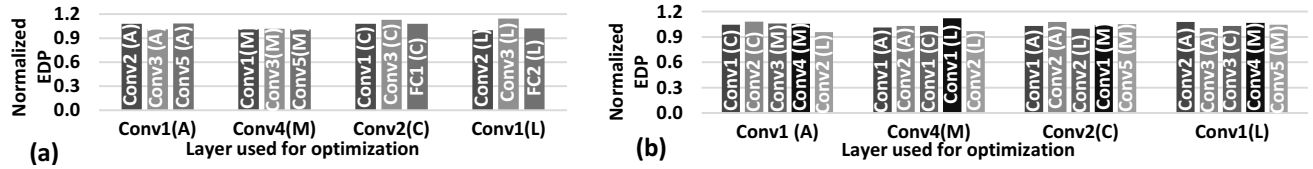
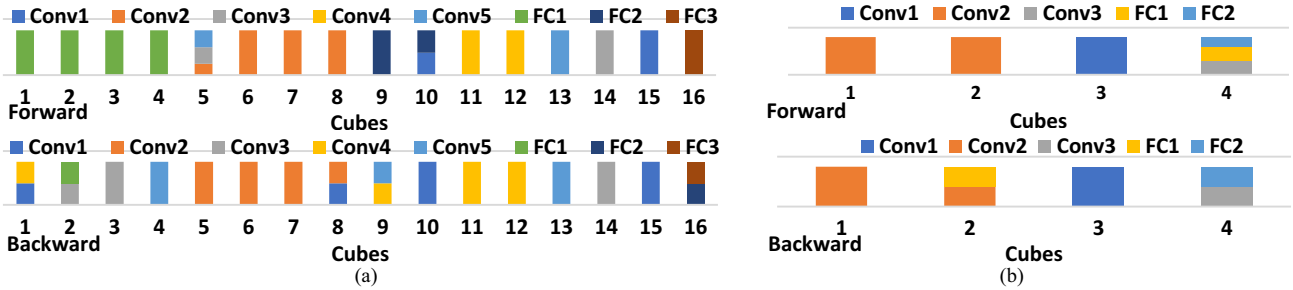


Fig. 3: Normalized EDP when an NoC optimized for one CNN layer is used to execute other layers from (a) same CNN (b) other CNNs (A: AlexNet, M: MattNet, C: CDBNet, L: LeNet)



(a) (b)

Fig. 4: Mapping layers in both passes during training on to cubes (forward on ReRAMs, backward on GPUs) for (a) AlexNet (b) LeNet. (Here, colors only indicate mapping of layers; they do not represent the exact amount of resources allocated in the cube)

Similarly, Fig. 3(b) highlights the case where the NoC is optimized for a layer but is used to execute layers belonging to other CNNs (inter-CNN). Both Fig. 3(a) and Fig. 3(b) indicate that NoCs optimized for a given $layer_1$ show minimal performance loss compared to another $layer_2$-specific architecture (where $layer1 \neq layer2$) when executing $layer_2$. For example, in Fig. 3(a), executing *Conv2* (AlexNet) on the NoC which was optimized specifically for *Conv1* (AlexNet) incurs only 8% EDP loss when compared to a *Conv2*-specific architecture. For the sake of brevity, we show only a handful of cases. However, a similar trend was observed for all the layers and CNNs considered in our study. On average, we observe only 6% performance degradation when different layers from various CNNs were executed on the NoC optimized considering just one of the layers. These observations indicate that it is possible to design an NoC optimized using one (or a subset) of layers/CNNs to capture the many-to-few traffic primarily, which can be used to train larger CNNs without prior knowledge about them.

Layer Mapping to REGENT cubes: The next crucial issue is the mapping of individual CNN layers to the cubes in REGENT (connected via the NoC) as it affects the overall achievable performance [7]. Here, layers are allocated and mapped to compute units (GPU, ReRAM) based on the amount of resources allocated to each layer to optimize the overall runtime. Also, the spatial locality in mapping forward and backward passes to the cubes is crucial in achieving high-performance and energy-efficiency as layers in both passes share weights and data [4] (Section III.D). Next, we show the effect of mapping the CNN layers to cubes using the bin-packing-based algorithm. Fig. 4 shows the results of mapping both passes to cubes (forward on ReRAMs and backward on GPUs) for two of the considered CNNs: AlexNet (Fig. 4(a)) and LeNet (Fig. 4(b)). For smaller CNNs, e.g., LeNet and CDBNet, we consider a reduced system size of 4 cubes (Fig. 4(b)). From Fig. 4, we note that in most cases, the backward pass is mapped on the same cube as its forward counterpart (best case) thereby preserving spatial locality. This allows high-throughput memory access via vertical links as data/weights can be fetched directly from the same cube without relying on the longer inter-cube links.

Interestingly, we also note that Full Connect (FC) layers require more cubes in the forward pass (ReRAMs) than the backward pass (GPUs) for larger CNNs. This happens as FC layers typically have a larger number of weights compared to convolution layers and all the weights need to be mapped individually on ReRAMs. This is necessary to avoid re-programming ReRAM crossbars during execution as it is time-consuming and can give rise to significant performance bottleneck [13]. However, GPUs do not have such a limitation and are therefore allocated based on the amount of parallelism required to balance all stages of the pipelined-CNN architecture adopted here [4]. It should be noted here that Fig. 4 only shows the mapping. The placement of the cubes integrated via NoC is further optimized following the MOO formulation of (2).

C. NoC and Full System Results

Based on the above layer mapping, we determine the traffic within each cube (intra-cube) and between them (inter-cube) to design the NoC (Section III.A, Fig. 1(b)) considering both forward and backward passes. REGENT employs a two-level Hierarchical NoC (*HNoC*), as shown in Fig. 1(b), the first level is optimized for the intra-cube many-to-few communication and the second level is optimized based on the inter-cube traffic. The first level NoC is same in all the cubes as we have already shown that an NoC optimized for one CNN layer (capturing many-to-few traffic in general) can perform at par with a layer-specific NoC (Fig. 3). The overall NoC is optimized following the methodology outlined in Section III.C, which brings the highly communicating tiles closer to each other (e.g., GPU-LLC). To achieve high throughput, link placement is optimized to increase path diversity between the highly communicating pairs. This allows traffic to be redistributed among multiple links, thus preventing congestion in a handful of links, as observed in conventional mesh NoCs [2]. Next, we compare the performance of *HNoC* with a similar state-of-the-art hierarchical NoC referred as *NeuNoC* [7], a heterogeneous two-level ring-mesh NoC that has been demonstrated to outperform a conventional single-level mesh NoC [7]. We also consider another NoC architecture, *Hmesh,* that follows a two-level mesh-mesh architecture for performance evaluation. The placement of tiles in both *NeuNoC* and *Hmesh* is optimized similar to *HNoC* following (2) (link placement remains unchanged; *NeuNoC*: ring-mesh [7], *Hmesh*: mesh-mesh).

Fig. 5 (a) and Fig. 5 (b) show the EDP and load distribution among the links respectively for the NoC architectures under consideration. To capture the performance difference due to the NoC only, we adopt the exact same layer mapping (Section III.D, Fig. 4) for all the three cases. It is clear that *HNoC* not only improves EDP significantly when compared to the remaining NoC architectures, but also distributes traffic more evenly among the links, thereby leading to higher throughput. On average, *HNoC* improves EDP when compared to the *NeuNoC* (ring-mesh) and *Hmesh* (mesh-mesh) by 56% and 18%, respectively. The improvement in *Hmesh* and *HNoC* over *NeuNoC* can be attributed to three major factors: (a) *NeuNoC* has fewer links compared to both *HNoC* and *Hmesh;* (b) *NeuNoC* does not have path diversity and forces data to follow a fixed path leading to congestion, and (c) Average hop count between cores in *NeuNoC* is higher than both *HNoC* and *Hmesh*. As a result, under many-to-few traffic pattern, the average load on

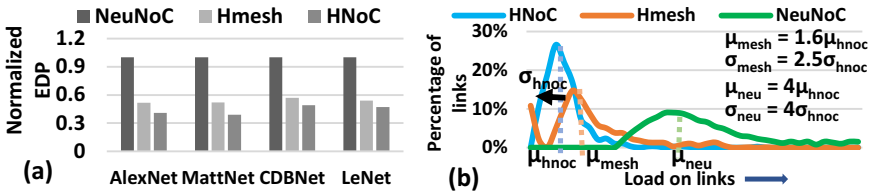


Fig 5: Comparison of (a) *NoC* EDP, (b) Load distribution in considered NoCs

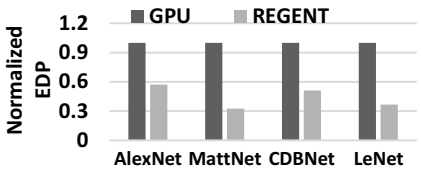


Fig 6: *Full-System* EDP comparison of REGENT and GPU-only architectures

links in *NeuNoC* is 4X and 2.5X higher when compared to *HNoC* and *Hmesh* respectively. The performance of *Hmesh* suffers due to the multi-hop nature of mesh and its inability to handle many-to-few traffic [2]. The average link load in *Hmesh* is 1.6X higher when compared to *HNoC* (Fig. 5(b)). This indicates that *HNoC* has fewer heavily congested links as traffic is distributed more uniformly. Hence, links are more readily available in *HNoC*, resulting in less network congestion and higher network throughput. By jointly optimizing throughput and energy, *HNoC* shows the best performance for training CNNs when compared to other NoCs.

Next, we compare the performance of REGENT with a conventional GPU-only platform in terms of full-system EDP (capturing both execution time and energy consumption) for training CNNs. Here, we consider *HNoC* in both cases to ensure similar NoC performance. The GPU-only platform has similar architecture as in Fig. 1(a): lower layer consisting of equal number of GPUs and LLCs in each cube, and ReRAMs in upper layer. However, ReRAMs are configured as *memory-only* in this case. The GPU and ReRAM parameters in both cases are same as described in Table 1. Fig. 6 shows the full-system EDP when different CNNs are executed on these two platforms. It is clear that the hybrid ReRAM/GPU architecture is more efficient and improves EDP by 55.7% on average compared to its GPU-only counterpart. The hybrid architecture performs better in terms of both execution time and energy. This is due to the fact that in GPU-only system, the layers (both forward and backward) are executed on GPUs, which are inherently slower and consumes more energy than ReRAMs [3][4]. Overall, the REGENT architecture together with *HNoC*, provides significant speed-up and energy savings for training CNNs when compared to conventional platforms. Here, we have omitted an accuracy comparison for the sake of brevity as similar experiments in prior work [6] (16-bit accuracy for forward and 32-bit accuracy for backward pass) have reported negligible accuracy loss when compared to a full 32-bit (GPU-only) implementation.

V. Conclusion

In this work, we introduced REGENT, a heterogeneous ReRAM/GPU-based architecture for training CNNs. REGENT incorporates the efficiency of ReRAMs with the high-precision computing of GPUs integrated using a high-throughput and energy-efficient NoC. We demonstrate that instead of optimizing the NoC for any specific CNN or a specific CNN layer, it is sufficient to consider the many-to-few traffic patterns inherent in any GPU-based architecture. Each layer of the CNN is mapped to the computing units (cubes) following a bin-packing formulation such that spatial locality of data between forward and backward passes is preserved. This reduces network traffic, leading to low-latency and energy-efficient communication. Experimental evaluation indicates that *HNoC* outperforms *NeuNoC*, a state-of-the-art NoC designed for neural networks, by 56% on an average while outperforming GPU-only platforms by 55.7% for training CNNs.

References

[1] Z. Lu, S. Rallapalli, K. Chan, T. L. Porta. "Modeling the Resource Requirements of Convolutional Neural Networks on Mobile Devices," in Proc. of ACM Multimedia Conf., Mountain View, CA, 2017, 1663-1671.

[2] W. Choi et. al., "On-Chip Communication Network for Efficient Training of Deep Convolutional Networks on Heterogeneous Manycore Systems," in IEEE TC, vol. 67, no. 5, 2018, pp. 672-686

[3] G. W. Burr et al., "Large-scale neural networks implemented with non-volatile memory as the synaptic weight element: Comparative performance analysis (accuracy, speed, and power)," in IEEE IEDM, Washington, DC, 2015, pp. 4.4.1-4.4.4.

[4] L. Song, X. Qian, H. Li and Y. Chen, "PipeLayer: A Pipelined ReRAM-Based Accelerator for Deep Learning," in IEEE HPCA, Austin, TX, 2017, pp. 541-552.

[5] M. Hu et. al., "Dot-product engine for neuromorphic computing: Programming 1T1M crossbar to accelerate matrix-vector multiplication," in DAC, Austin, TX, 2016, pp. 1-6.

[6] Y. Chen et. al., "DaDianNao: A Machine-Learning Supercomputer," in MICRO, Cambridge, 2014, pp. 609-622.

[7] X. Liu et. al., "Neu-NoC: a high-efficient interconnection network for accelerated neuromorphic systems," in ASPDAC, IEEE Press, Piscataway, NJ, 2018, 141-146

[8] A. Shafiee et. al., "ISAAC: a convolutional neural network accelerator with in-situ analog arithmetic in crossbars," in SIGARCH Comput. Archit. News 44, 2016, 14-26.

[9] P. Chi et al., "PRIME: A Novel Processing-in-Memory Architecture for Neural Network Computation in ReRAM-Based Main Memory," in ISCA, Seoul, 2016, pp. 27-39.

[10] T. Na, J. H. Ko, J. Kung and S. Mukhopadhyay, "On-chip training of recurrent neural networks with limited numerical precision," in IJCNN, Anchorage, AK, 2017, pp. 3716-3723

[11] A. Bakhoda, J. Kim and T. M. Aamodt, "Throughput-Effective On-Chip Networks for Manycore Accelerators," in MICRO, Atlanta, GA, 2010, pp. 421-432

[12] A. A. Maashri et. al., "3D GPU architecture using cache stacking: Performance, cost, power and thermal analysis," *in IEEE ICCD*, Lake Tahoe, CA, 2009, pp. 254-259.

[13] Y. Long, T. Na and S. Mukhopadhyay, "ReRAM-Based Processing-in-Memory Architecture for Recurrent Neural Network Acceleration," in IEEE TVLSI, 2018

[14] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," in Proc. of the IEEE, 1998

[15] Convnet: Deep Convolutional Network: http://libccv.org/ (CIFAR-10)

[16] A. Krizhevsky, I. Sutskever, G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in NIPS, 2012, pp. 1106–1114.

[17] M. D. Zeiler, R. Fergus, "Visualizing and understanding convolutional networks," in Proc. ECCV, 2014, CoRR, abs/1311.2901

[18] J. Power et.al., "gem5-gpu: A Heterogeneous CPU-GPU Simulator," in IEEE Computer Architecture Letters, vol. 14, no. 1, pp. 34-36, 2015

[19] S. Bandyopadhyay, S. Saha, U. Maulik, K. Deb, "A Simulated Annealing-Based Multiobjective Optimization Algorithm: AMOSA," in IEEE Trans. on Evolutionary Computation, vol. 12, no. 3, pp. 269-283, 2008.

[20] N. Menakerman, R. Rom, "Bin packing with item fragmentation," in Proc. of 7th Intl. Workshop on Algorithms and Data Structures, Lecture Notes in Computer Science, vol. 2125, 2001, pp. 313–324.