SCIENCE CHINA
Information Sciences

June 2021, Vol. 64 160401:1–160401:25
https://doi.org/10.1007/s11432-020-3219-6

• REVIEW •
Special Focus on Near-memory and In-memory Computing

Graph processing and machine learning architectures with emerging memory technologies: a survey

Xuehai QIAN

Ming Hsieh Department of Electrical and Computer Engineering, University of Southern California, Los Angeles 90089*, USA*

Received 31 December 2020/Revised 15 March 2021/Accepted 17 March 2021/Published online 10 May 2021

Abstract This paper surveys domain-specific architectures (DSAs) built from two emerging memory technologies. Hybrid memory cube (HMC) and high bandwidth memory (HBM) can reduce data movement between memory and computation by placing computing logic inside memory dies. On the other hand, the emerging non-volatile memory, metal-oxide resistive random access memory (ReRAM) has been considered as a promising candidate for future memory architecture due to its high density, fast read access and low leakage power. The key feature is ReRAM's capability to perform the inherently parallel in-situ matrix-vector multiplication in the analog domain. We focus on the DSAs for two important applications—graph processing and machine learning acceleration. Based on the understanding of the recent architectures and our research experience, we also discuss several potential research directions.

Keywords graph processing, machine learning acceleration, ReRAM, HMC/HBM

Citation Qian X H. Graph processing and machine learning architectures with emerging memory technologies: a survey. Sci China Inf Sci, 2021, 64(6): 160401, https://doi.org/10.1007/s11432-020-3219-6

1 Introduction

The improvement of complementary metal-oxide-semiconductor (CMOS) technology has been the driving force that enabled architects to invent various aggressive techniques to achieving high performance. In 1965, Gordon Moore predicted that the transistor density would double yearly, and in 1975, revised to a doubling every two years—eventually known as Moore's law. Another famous prediction is "Dennard scaling" made by Robert Dennard, stating that as transistor density increases, power consumption per transistor would drop, making the power per mm^2 of silicon nearly constant. Since the compute capability of a mm^2 of silicon increases with each technology generation, Dennard scaling implies that computers would become more energy efficient. Moore's law began to slow around 2000 and by 2018 showed a roughly 15-fold gap between the prediction and current capacity [1]. Similarly, Dennard scaling began to slow significantly in 2007 and completely faded by 2012 [1]. Before mid-2000s, the interaction between software and hardware was non-essential: the unmodified software can expect to run faster in a year or two by simply using newly announced microprocessors.

The end of Moore's law and Dennard scaling triggered major changes in the landscape of industry, entering the era of "dark silicon". The sharp increase in power densities prevents all the transistors from being powered simultaneously at the nominal voltage. This triggers the proliferation of domain-specific architectures (DSA) and domain-specific languages (DSL). Tailored for specific problem domains, the DSAs can potentially offer significant performance and power efficiency gains. The DSLs enable the expression of domain-specific structures, e.g., vector, sparse matrix, and graph operations, so that the problems can be efficiently mapped to DSAs. The emerging applications supported by DSAs and DSLs offer breathtaking opportunities. As the 2018 Turing Award winners John Hennessy and David Patterson pointed out in the Turing lecture [1], we have entered the "the new golden age for computer architecture", where achieving significant gains through DSA and DSL will require vertical integration that understands

Email: xuehai.qian@usc.edu

and connects applications, DSL and frameworks, computer architecture, and the underlying and emerging technology. Improving the software/hardware (SW/HW) interfaces and making design decisions across layers of abstractions are crucial to understanding complex trade-offs and achieving performance improvements.

The domain-specific architecture faces the important challenge of high cost of data movement. As the heart of deep learning applications, convolutional neural networks (CNNs) are not only compute intensive but also memory intensive. For example, to process just one image data, AlexNet [2] performs 10^9 operations. Since the conventional von Neumann architecture separates the computation and data storage, a large amount of data movements are incurred to process deep networks with large number of layers and millions of weights. Such data movements not only lead to a critical performance bottleneck due to the limited memory bandwidth, but also, and more importantly, an energy bottleneck. A recent study [3] showed data movements become the dominant source of energy consumption compared to computation operations—the data movements between CPUs and off-chip memory consumes two orders of magnitude more energy than floating point operations. Based on the application demands and high cost, it is crucial to reduce data movement and computing cost. In this study, we survey the recent DSA built from two emerging memory technologies.

The first technology we consider is hybrid memory cube (HMC) [4] and high bandwidth memory (HBM) [5]. This type of memory can reduce data movement between memory and computation by placing computing logic inside memory dies. The same idea has been proposed and investigated decades ago, but gradually fade out because it is not technically practical. Recently, thanks to the emerging 3D stacked memory technology, researchers intensively investigated the potential of the technology on various applications that did not exist before. In an abstract view, the architecture contains multiple memory cubes, and they are connected by external SerDes links with 120 GB/s per link. Within each cube, multiple DRAM dies are stacked through silicon via (TSV), providing higher internal memory bandwidth up to 320 GB/s. At the bottom of the dies, computation logic such as simple in-order cores can be embedded. Performing computation at in-memory compute logic can clearly reduce data movements in memory hierarchy. More importantly, HMC and HBM provide "memory-capacity-proportional" bandwidth and scalability. We consider the architectures developed based on HMC/HBM as near data processing (NDP) architectures because the computation units are moved closer to memory.

The second technology we consider is the emerging non-volatile memory, metal-oxide resistive random access memory (ReRAM) [6–9]. It is considered as one of the promising candidates for future memory architecture due to its high density, fast read access and low leakage power. Besides the advantage of being used as a type of non-volatile memory, ReRAM also enjoys the unique key feature of being able to perform the inherently parallel in-situ matrix-vector multiplication in the analog domain. For many important applications such as machine learning and graph processing, the key computation kernels all can be essentially expressed as matrix-vector multiplication, and thus ReRAM crossbars can naturally accelerate them with much less data movement and low-cost computation. We consider the architectures developed based on ReRAM as processing-in-memory (PIM) architectures because the computations are directly performed by ReRAM crossbars.

In this study, we focus on the domain-specific NDP or PIM architectures for two important applications —graph processing and machine learning acceleration. Graphs are natural data representation to capture the relationships between data items, such as interactions or dependencies. Graph analytics is an important way to understand the relationships between heterogeneous types of data. In various important applications, such as machine learning tasks [10], natural language processing [11–13], anomaly detection [14–16], clustering [17, 18], recommendation [19–22], social influence analysis [23–25], bioinformatics [26–28], the valuable insights from patterns in the graph data can be obtained. At the same time, the execution of the graph algorithms on conventional architecture also poses two key challenges. First, they have poor locality because of the random accesses in traversing the neighborhood vertices. Second, they demand high memory bandwidth requirement because the computations on data accesses from memory are typically simple. In another word, the time spent on computation after fetching a piece of data from memory hierarchy is short, making it hard to hide the latency. Moreover, a side effect of the poor locality is the memory bandwidth waste. When a cache line is fetched, only a small portion of it is used in the computation. As a result, graph processing incurs a significant amount of data movements and energy consumption.

The second application we consider is the acceleration of deep neural networks (DNNs), which have become the fundamental element and core enabler of the ubiquitous artificial intelligence [29]. As we

showed earlier, the execution of DNNs leads to substantial data movements and computation operations. To achieve high accuracy, the model size is also growing rapidly, exaggerating the bottlenecks. To reduce data movement, two techniques are intensively studied. First, the model compression techniques [30–33] reduces model sizes and thus eliminates many dynamic random-access memory (DRAM) accesses. The challenge is to still achieve the high accuracy with the smaller model sizes. The second trend is the design of hardware accelerators [34–42][1)]. While perhaps having quite different architectures, they all share the benefit of performing computations efficiently in manners that matches the structure of the DNNs. The computation cost can be reduced by lowering the precision of floating point operations, or avoiding redundant computation (e.g., multiplication with zeros) at an element or even bit level. Most of these ideas have been investigated using CMOS technology, but with the end of Moore's law [43], the potential of the acceleration architecture based on conventional technology might be limited. We believe that the drastic improvements can be likely achieved by (1) the next-generation emerging device/circuit technology beyond CMOS; and (2) the vertical integration [1] and optimization of algorithm, architecture and technology innovations to deliver better overall performance and energy efficiency for various applications.

The paper is organized as follows. Section 2 provides the background of different emerging memory technologies. Section 3 discusses the key problems and challenges of graph processing and machine learning acceleration. Section 4 surveys the recent PIM and NDP architectures for graph processing. Section 5 surveys the recent PIM and NDP architectures for machine learning acceleration. Section 6 discusses several potential future research directions. Section 7 concludes the paper.

2 Emerging memory technologies

This section provides the background on HMC/HBM and ReRAM memory technology. They are the essential building blocks for various architectures we survey in this paper.

2.1 Hybrid memory cube

Thanks to the technology advance, the 3D integration technology [44] enables the integration of computing logic with memory dies. In this paper, we mainly discuss on HMC [45], but the insights and principles can be also applied to other alternatives such as HBM [46]. Both of them can be abstractly considered as a number of computing nodes with local memory: the computation is performed by the logic die under the multiple levels of memory dies. The property is that accessing data in the local memory dies is shorter than accessing the remote dies. An HMC device, also called a cube, is a single chip stack that consists of several memory dies/layers and a single logic die/layer. Based on this structure, we should consider two kinds of bandwidth. The internal bandwidth determines the maximum data transfer speed between memory dies and the logic dies of a same cube. The external bandwidth determines the data transfer capability from the cube to the external devices such as other cubes and the host processor.

Figure 1 shows the organization of one HMC based on the specification 2.1 [45]. We see that each cube is divided into 32 vertical slices (also known as vaults), and has at most 4 multiple serial links serving as the interface for off-chip communication, and a crossbar network that connects the slices. Each vault has a logic layer and several memory layers, which provides up to 256 MB of memory space (8 GB space per cube). These layers are connected through low-power TSV. Each TSV can provide up to 10 GB/s of bandwidth, and thus the maximum internal bandwidth of a cube is $32 \times 10 = 320$ GB/s. Based on the default configuration, the each off-chip links contain 16 input lanes and 16 output lanes for full duplex operation. This leads to at most 480 GB/s external bandwidth (i.e., 120 GB/s per link).

Besides the capability of providing high density and bandwidth, HMC also makes it possible to integrate computation logics into its logical die/layer. For example, in Tesseract [47], a single-issue, in-order core and a prefetcher are placed in the logic die of each vault (i.e., 32 cores per cube). It is feasible, because the area of 32 ARM Cortex-A5 processors including an FPU (0.68 mm^2 for each core[2)]) corresponds to only 9.6% of the area of an 8 GB DRAM die area (e.g., 226 mm^2 [48]). The key benefit that HMC can provide is memory-capacity-proportional bandwidth. Thus, the benefits of multiple HMCs are always preserved. Typically, a system that contains N HMCs can provide $N \times 8$ GB memory space and $N \times 320$ GB/s

1) Google supercharges machine learning tasks with TPU custom chip. https://cloudplatform.googleblog.com/2016/05/Google-supercharges-machine-l earning-tasks-with-custom-chip.html.

2) ARM. ARM Cortex-A5 Processor. http://www.arm.com/products/processors/cortex-a/cortex-a5.php.

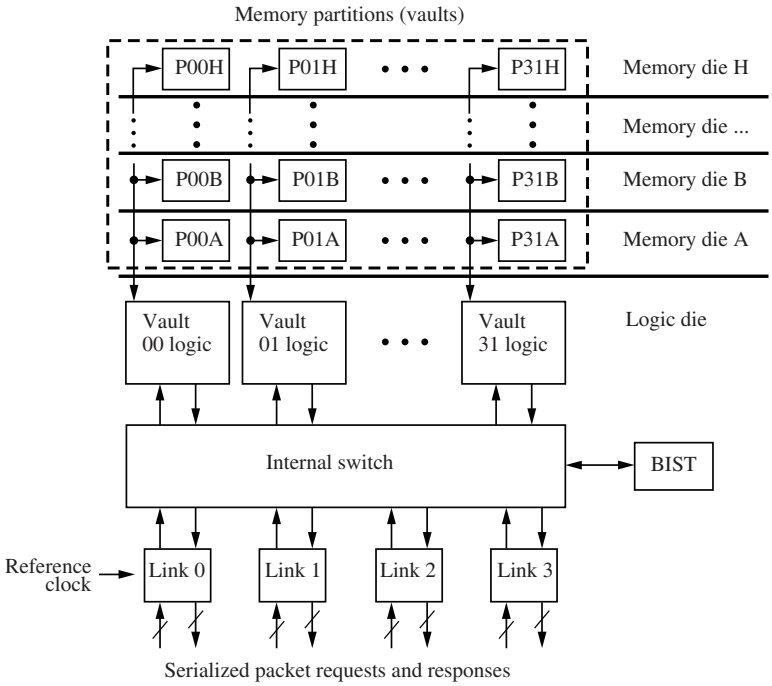

Figure 1 An implementation of HMC.

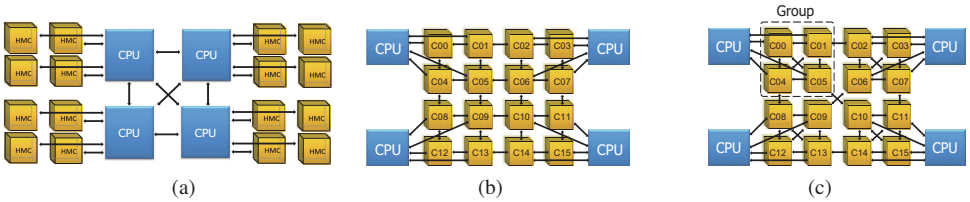

Figure 2 (Color online) Examples of HMCs' interconnections. (a) Processor-centric network; (b) memory-centric network: mesh; (c) memory-centric network: dragonfly

aggregation internal bandwidth. The actual aggregated bandwidth is determined by the interconnection network that connects the HMCs and host processors.

A straightforward interconnect design is the "processor-centric network", which simply reuses the current non-uniform memory access (NUMA) architecture and replaces traditional dual in-line memory modules (DIMMs) with HMCs. Figure 2(a) presents a typical system that contains four processor sockets. In this case, a fully-connected interconnection network among the processors is built based using Intel QuickPath Interconnect (QPI) technology. A processor is connected to a set of HMCs, while each HMC can be only exclusively connected to a particular processor. This also means that there is not a direct connection between HMCs. This network organization is simple and compatible with the current architecture. Unfortunately, Kim et al. [49] concluded that this processor-centric organization does not fully utilize the additional opportunities offered by multiple HMCs. Since the routing/switching capacity can be supported by HMC's logic die, it is possible to use more sophisticated topology and connection that were infeasible with traditional DIMM-based DRAM modules. To take this opportunity, Kim et al. [49] proposed the "memory-centric network", which allows HMCs to be directly connected to each other without providing direct connections among processors. This means that all processor communications need to go through HMCs indirectly. The evaluation results show that the throughput of a memory-centric network can exceed the throughput of a processor-centric network.

Kim et al. [49] also evaluated various types of topology to interconnect HMCs. Figures 2(b) and (c) show two commonly used examples. Dragonfly [50] is shown to be advantageous for three reasons. First, it has higher connectivity and shorter diameter than a simple topology like mesh. Second, it can achieve a similar performance as the best interconnection topology, named flattened butterfly [51], according to the evaluation of 16 HMCs. Third, it does not face the same scalability problem as a flattened butterfly.

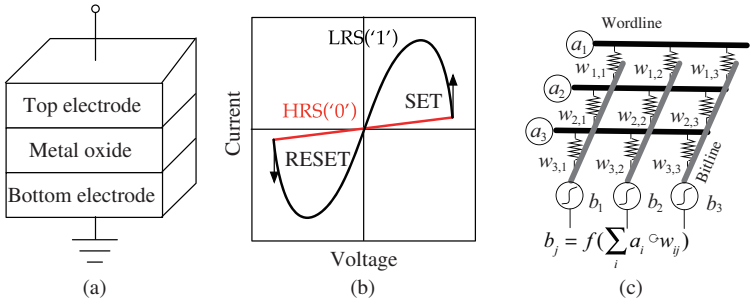

Figure 3 (Color online) Basics of ReRAMs. (a) Metal-insulator-metal structure of a ReRAM cell; (b) switch between high/low resistance states; (c) in-situ matrix-vector multiplication in ReRAM crossbars.

2.2 ReRAM basics

Thanks to the significant progress of fabricating non-volatile memories, the non-volatile memories have become commercially available. For example, the 3D Xpoint [52, 53] is an example of commercial non-volatile memories fabricated jointly by Micron and Intel. Resistive RAM is a type of non-volatile memory with nearly zero leakage power, high integration density, and high scalability. Research papers demonstrated the results of fabricated ReRAM memory cells, memory arrays [6, 54–56] and also neuromorphic accelerators using ReRAM technology [52, 53, 57–63].

Primarily, the ReRAM can be used as an alternate for main memory [64–66]. Figure 3(a) shows the ReRAM cell containing the metal-insulator-metal (MIM) structure. The structure is composed of a bottom electrode, a top electrode and a metal-oxide layer sandwiched between electrodes. An external voltage can be applied to a ReRAM cell and switch it between a high resistance state (HRS or OFF-state) and a low resistance state (LRS or ON-state). They can represent the logical "0" and "1", respectively, as shown in Figure 3(b). The concern of the lifetime is not as serious for ReRAM as the other non-volatile memory, such as PCM [67], since the endurance of ReRAM could be up to 10^{12} [68, 69].

As a secondary but important application, the ReRAM also features the capability to perform in-situ matrix-vector multiplication [70, 71] as shown in Figure 3(c). It utilizes the property of bitline current summation in ReRAM crossbars to enable computing with high performance and low energy cost. While conventional CMOS based architecture showed substantial success on neural network acceleration [72–74], recent studies [75–78] demonstrated that ReRAM-based architectures can provide significant performance and energy benefits for the neural network computing that is both compute and memory intensive.

3 Understanding the applications

3.1 Graph processing

3.1.1 *Graph computation and application programming interfaces (APIs)*

A graph G is defined as an ordered pair (V, E), where each edge in the set E connects a pair of vertices in V. During the execution, a graph algorithm visits all edges and vertices to extract the hidden structures and information based on the graph structure. A graph can be naturally considered as an adjacency matrix, where each matrix row and column correspond to a vertex, and the matrix elements represent the edges. An element is non-zero if there is an edge between the vertices corresponding to the row and column. Thus, matrix operations can express most graph algorithms. However, in practice, the graph is typically sparse—most elements in the adjacency matrix are zeros. If we store the graphs and perform the computations using matrix format, it will obviously incur both storage and compute resources waste. Therefore, all graph processing systems support the data representations for sparse graph data. In the conventional architecture with CPUs or GPUs, each of the simple operation, e.g., multiplication of two non-zeros, on sparse data is performed as the normal ALU instruction.

Due to the importance of developing graph algorithms, the programmers must be able to easily express the desired graph computation. For this purpose, several domain-specific programming models that emphasize the local view of computations are proposed. The key principle is "think like a vertex"—the user-defined functions just need to specify the local computation on a vertex using the data of

Table 1 Vertex programming APIs

UDF	Input parameters	Output	Operation
`processEdge`	Source vertex value	Partial update	User-defined computation for each edge
`reduce`	Reduced/partial update	Reduced update	Generate update value for dest vertex
`apply`	Reduced update/old value	New value	Update the value of dest vertex

```
for (v ← Graph.vertices) {
    for (e ← outEdges(v)) {
        res = processEdge(e, v.value, ...)
        u ← comp[e.dest]
        u.temp = reduce(u.temp, res)
    }
}
for (v ← Graph.vertices) {
    v.value, v.active = apply(comp[v].temp, v.value)
}
```

Figure 4 Vertex programming model.

neighbors and incoming/outgoing edges. The programming models designed based on this principle include amorphous data-parallel program [79], gather-apply-scatter program [80], vertex program [81], and some other frameworks [82]. Among them, the vertex program is a commonly supported interface, which is adopted by several software and hardware accelerated graph processing frameworks adopted this API, including Tesseract [47], GraphLab [83], and Graphicionado [84]. Table 1 lists the semantics of three user-defined functions (UDF) of vertex program. Figure 4 illustrates a general graph application expressed with these primitives.

Typically, the execution of graph processing visits all vertices in the vertex array in a certain order, when a vertex is visited, its incoming or outgoing edges are also visited. For each edge, an updated value for the destination vertex is computed and stored in a compute array, which is updated to the vertex array after an iteration. This process involves three steps that can be expressed in user-defined functions. (1) Process. The `processEdge` function computes the partial update of source vertex `v` through edge `e` to the destination vertex `u` for every out-going edge `e` of vertex `v`. (2) Reduce. From each vertex `u`, the `processEdge` function returns the new update, which is aggregated with the current value of `u` in compute array by the `reduce` function. This step incurs a random access. (3) Apply. It is executed after an iteration, and the updated value of each vertex in the compute array is applied to the vertex array by the `apply` function. The graph algorithms are normally iterative, the above steps are repeatedly executed for multiple iterations until certain convergence condition is achieved. For example, the difference of page rank of a vertex after two consecutive iterations becomes smaller than a pre-defined threshold.

3.1.2 *Architectural challenges*

The graph processing applications are generally believed to have poor locality and high memory bandwidth requirement. However, it is important to understand the reason. While randomly accessing the vertices will certainly lead to random accesses, the graph processing execution actually accesses them mostly sequentially. As we discussed above, all vertices are processed in the order according to how they are stored in a vertex array, so there is no random access. For each vertex, its edge list is accessed sequentially as well; the random access only happens when we move from one vertex to another. In Figure 5(b), it is indicated as the global random access. However, the ratio of such random access is quite low. The major source of random access actually happens when we update the compute array for each edge. In synchronous graph processing, the compute array is only updated to the vertex array at the end of an iteration. For asynchronous graph processing, the vertex array is directly updated. In either case, storing the new value for the destination vertex of an edge is a random access. Intuitively, we can consider the graph data access having three aspects: source vertices, edges, and destination vertices. Among the three, only two can be accessed sequentially.

The reason for the high memory bandwidth requirement is due to the nature of graph algorithms and the sparse representation. Typically, the computation involved in graph algorithms is quite simple, e.g., computing the partial update, and then it is followed with a random access to store the newly

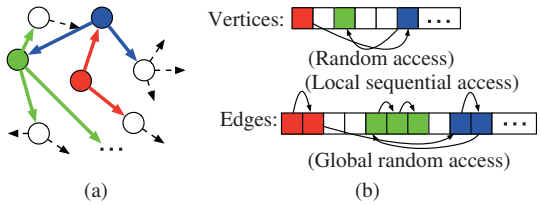

Figure 5 (Color online) Vertex and edge accesses. (a) Vertex and edge accesses in graph processing; (b) access pattern in vertexcentric program.

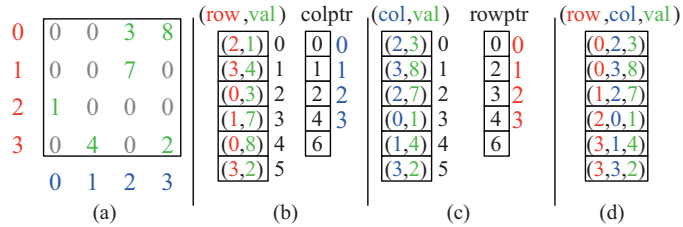

Figure 6 (Color online) (a) Sparse matrix representations in (b) CSC, (c) CSR, (d) COO.

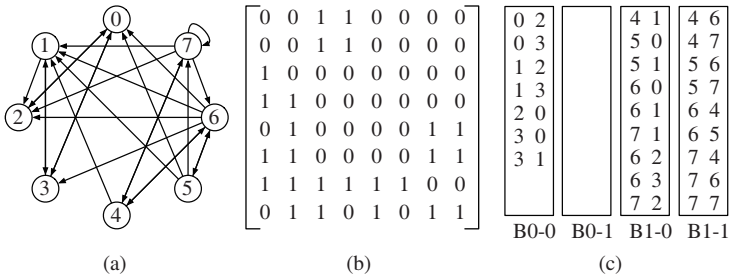

Figure 7 (a) A directed graph represented by (b) adjacency matrix and (c) coordinate list.

computed value. Thus, the whole execution timeline can be considered as an alternative sequence of short computation time and long random access latency. Since the computation is much faster than memory access in this scenario, the execution is memory bound and requires high memory bandwidth.

3.1.3 *Graph representation*

As discussed before, graphs are typically stored in the compressed sparse matrix representation. Here, Figure 6 illustrates three commonly used compressed sparse representations: coordinate list (COO), compressed sparse row (CSR), and compressed sparse column (CSC). CSC representation stores the non-zeros in column major order as (row index, value) pairs in a list, so the number of non-zeros is equal to the number of entries in the list. The starting index of a row in the (row, val) list is kept by another list, each entry indicates a column starting pointer. In Figure 6(a), 4 in the `colptr` indicates that the 4-th entry in (row, val) list, i.e., (0, 8) is the starting of column 3. The number of entries in `colptr` is equal to (the number of columns + 1). Similar to CSC, the CSR just swaps the row and column. The COO is more straightforward: each entry directly indicates the position and the non-zero value by a tuple—(row index, column index, value).

Given a graph in Figure 7(a), both the adjacency matrix representation and COO representation are partitioned into four 2×2 subgraphs shown in Figures 7(b) and (c). In this example, the coordinate list saves around 61% space to store the graph than the ordinary adjacency matrix. In the real-world, the graphs tend to have higher sparsity, making the saving even higher.

3.2 Machine learning acceleration

3.2.1 *The computation of deep neural network*

CNN is the core component of many deep learning applications including computer vision [85–89], data mining [90–93], and language processing [94–98]. Figure 8 illustrates the organization of an example CNN

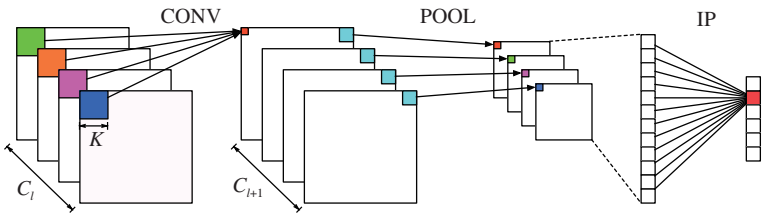

Figure 8 (Color online) Convolutional neural network.

with three types of layers: convolution layer, pooling layer and inner product layer. In a convolution layer, a set of kernels are convoluted with data of channels from the previous layer (layer l) to generate data for channels of next layer (layer $l+1$). d_l is a cube of data in a layer. $d_l[x, y, c]$ is the value at a point in the three dimensional data cube. Let $(X_l \times Y_l \times C_l)$ denote the size of d in layer l, so $0 \leqslant x \leqslant X_d - 1, 0 \leqslant y \leqslant Y_d - 1, 0 \leqslant c \leqslant C_d - 1$ and C_d is the number of channels. (x_l, y_l, c_l) indicates a point in layer l's data cube. K is the kernel composed of a set of weights. K_l is the kernel used in the computation to generate data in layer l. A kernel represents four dimensional data: the size of each dimension is K_x, K_y, C_l and C_{l+1}, where K_x and K_y are determined by algorithm (e.g. in LeNet [99], K_x and K_y are both 5).

d_{l+1} is computed as

$$d_{l+1}[x, y, c] = \sum_{c_l=0}^{C_l-1} \sum_{k_x=0}^{K_x-1} \sum_{k_y=0}^{K_y-1} K_l[k_x, k_y, c_l, c] \times d_l[x + k_x, y + k_y, c_l]. \tag{1}$$

To perform (1), in total $(X_{l+1} \times Y_{l+1} \times C_{l+1} \times C_l \times K_x \times K_y)$ multiplications and $(X_{l+1} \times Y_{l+1} \times C_{l+1} \times (C_l \times K_x \times K_y - 1)$ additions are performed.

A pooling layer performs the subsampling. Taking average pooling as an example, a window of data in l is averaged to get one data point in $l+1$ as follows:

$$d_{l+1}[x, y, c] = \frac{1}{K_x K_y} \sum_{k_x=0}^{K_x-1} \sum_{k_y=0}^{K_y-1} d_l[K_x x + k_x, K_y y + k_y, c]. \tag{2}$$

This average pooling operation performs $(X_{l+1} \times Y_{l+1} \times C_{l+1} \times (K_x \times K_y - 1))$ additions and $(X_{l+1} \times Y_{l+1} \times C_{l+1})$ multiplications. The multiplication could be implemented as shift operation if $(K_x \times K_y)$ is the power of 2. Max pooling is another variance, where the maximum value among values in a window in l is selected for $l+1$.

In the inner product layer, the values in data tube of l and $l+1$ are considered as a vector (denoted as d_l and d_{l+1}). If the previous layer is convolution or pooling, the size of d_l is $X_l \times Y_l \times C_l$. If the previous layer is also inner product, then the size of d_l is the size of the output vector from l. d_{l+1} is an $n \times 1$ vector, n is determined by the algorithm. W_{l+1-l} is a weight matrix of size $(n \times m)$, m is the size of d_l. b is a vector of bias.

The vector of $l+1$ is computed as

$$d_{l+1} = W_{l+1-l} d_l + b. \tag{3}$$

This inner product operation performs $(n \times (m-1))$ additions and $(n \times m)$ multiplications.

Activation function is another important component. It is an element-wise operation and usually a nonlinear function, such as sigmoid $\frac{1}{1+e^{-x}}$ or rectified linear unit (ReLU) $\max(0, x)$.

3.2.2 *Data forward and backward in a neural network*

A neural network should be trained with a large amount of data and then deployed into the real-world applications and perform inference tasks. Compared to training, the inference is simpler. Its goal is to give input samples, such as an image, to the neural network, which will generate the prediction, e.g., the type of object in the image. For the modern deep neural networks, the input data samples go through the layers in sequence in forward direction. This is shown in Figure 9 and is often called forward propagation. We can see that the input of layer l is the output of layer $l-1$. The concrete computations in forward

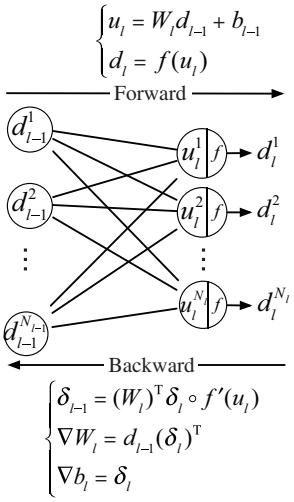

Figure 9 Two adjacent layers.

propagation are expressed as the two equations shown above. The goal for training is to generate the weights with a data set. The computation is much more complex and intensive in a sense that it does not only perform forward propagation but also backward propagation, which update the weights based on the errors and newly observed data in the training set. A cost function is needed in training a neural network since we need a way to measure how well can a neural network make the prediction by comparing the network's output with the standard labels. We use y and t to represent the output of a neural network and the standard label respectively. An L^2 norm loss function is defined as $J(W,b) = \frac{1}{2}||y-t||_2^2$ and $J(W,b) = -\sum_{i,j} 1(y^i = t^j)\log p(y^i = t^j)$ is the softmax loss function.

The error δ for each layer is defined as $\delta_l \triangleq \frac{\partial J}{\partial b_l}$. If we use an L^2 norm loss function, for the last (output) layer L, the error is $\delta_L = f'(u_L) \circ (y-t)$ where $\circ$ represents a Hadamard product, i.e., element-wise multiplications. For other layers excluding the output layer, the error is $\delta_l = (W_{l+1})R^{\mathrm{T}}\delta_{l+1} \circ f'(u_l)$. And with a ReLU activation function, the error can be rewritten as $\delta_l = (W_{l+1})^{\mathrm{T}}\delta_{l+1} \circ f'(d_l)$. So that the backward partial derivative to W^l is $\frac{\partial J}{\partial W_l} = d_{l-1}(\delta_l)^{\mathrm{T}}$. And the backward partial derivative to b_l is $\frac{\partial J}{\partial b_l} = \delta_l$. The gradient descent method can be applied to generate the weight updates of the neural network. Because the update of weights to a layer depends on the previous layer's error and earlier forward propagation, the training has more data dependencies and more time consuming. It is recently reported that the training can even take more than half a month [100].

4 PIM and NDP architectures for graph processing

4.1 ReRAM-based graph processing accelerator

4.1.1 *GraphR*

GraphR [101] is the first architecture that leverages ReRAM crossbar to perform graph processing. As discussed before, the graph computation can be naturally expressed as matrix operations, which can be performed efficiently in ReRAM crossbar, but at the same time, performing computation on matrix view also leads to storage and computation waste, e.g., multiplying by zero is always zero. The key insight to reconcile the conflicting goals is the following. The size of the ReRAM crossbar is typically small, e.g., 4×4. During graph processing, we can imagine that such a small square covers only a small portion of the large matrix for the complete graph. If any element in this small square is non-zero, we will perform the computation for all, but the waste is minimum due to the low computation cost. On the other side, if the graph is highly sparse, then it is also likely that all elements inside the square are zeros, in this case, the whole square can be skipped. Since the graph is stored in sparse format, we can apply graph reordering based on the architecture parameters so that each such square covered by the ReRAM crossbars is filled by sequentially accessing the edges (non-zeros) in the sparse graph data. With the high level insights, next we explain the detailed mapping of graph processing procedure to ReRAM crossbars.

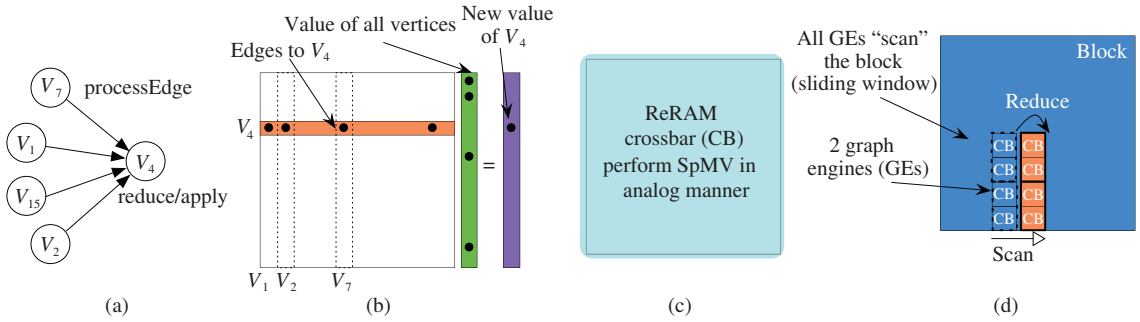

Figure 10 (Color online) GraphR key insight: supporting graph processing with ReRAM crossbars. (a) Vertex program in graph view. (b) Vertex program in matrix view. (c) Ideal case: CB of $|V| \times |V|$, $|V|$ is the number of vertices in a graph. (d) Realistic case: memory ReRAM stores a block of graph; ReRAM GEs process/scan subgraphs (sliding window).

The graph view of the vertex program is shown in Figure 10(a). Here, the `processEdge` function is executed for V_{15}, V_7, V_1 and V_2, and the outputs are stored in the edges from each of these vertexes to V_4. When all incoming edges of V_4 are processed, the `reduce` function is executed to generate V_4's new property, which is applied to the vertex. For graph processing, the computation performed in `processEdge` function is typically a multiplication that generates the updated property for each vertex. To generate the final update, partial results are reduced by a multiply-accumulate (MAC) operation. In this sense, executing the vertex program of each vertex is equivalent to performing a sparse matrix vector multiplication (SpMV). The connection is shown in Figure 10(b).

Let A be the sparse adjacency matrix, and `V.prop` of all vertices is stored in a vector x. The vertex program of all vertices covered by the crossbar can be computed in parallel in matrix view as $A^{\mathrm{T}}x$. This explains why the waste of computation on zeros is not significant with the ReRAM, because the extra computations do not lead to longer execution time. Therefore, a ReRAM crossbar is able to perform matrix-vector multiplication efficiently, as shown in Figure 3(c). Since a vertex program is equivalent to an SpMV, a ReRAM crossbar can accelerate the computation.

From Figure 10(b), we see that the sizes of the matrix and vector are $|V| \times |V|$ and $|V|$, respectively, where V is the number of vertices in the subgraph. With a ReRAM CB of size $|V| \times |V|$, the partial update of each vertex can be computed in parallel (i.e., `V.prop` in Figure 4). Using such crossbars as the building blocks, we can construct graph engines (GEs) composed of small CBs to process all subgraphs in a whole graph. There are still two remaining questions: (1) what is the size of subgraphs that should be processed together? (2) what is the order that all subgraphs should be processed?

The answer to the first question is intuitive, since it indicates a trade-off between the performance and efficiency. With larger subgraphs, all edges inside it can be processed in parallel, but if there are only a few edges, the wasted computation is high. In the extreme case, a subgraph is processed just because there is one edge inside it. With smaller subgraphs, the wasted computation is less but the performance is lower because all subgraphs should be processed sequentially following a certain order. In the experiments, we find that the subgraph sizes of 4 or 8×8 works well, which maps nicely to the realistic crossbar sizes. The second question is more subtle, and the GraphR design uses a column-major stream-apply execution model to reduce the needed register to store the intermediate results. The insight is that, we can divide the columns into several partitions, and we let the "square" in the matrix that is covered by all GEs in the architecture move vertically through each column partition one by one. This means that we first compute the final updates of a set of vertices before moving to another set. Compared to the column-major order, this method can effectively reduce the needed registers to store the partial accumulated updates.

Finally, we consider the mapping of algorithms to the ReRAM crossbars. GraphR can support two types of algorithms that are mapped in different manner with different parallelism. The key distinction is the type of computation performed by the `processEdge` function. For algorithms like Pagerank, `processEdge` performs a multiplication, which can be mapped to each cell and achieve $N \times N$ speedups. For algorithms like SSSP, `processEdge` performs an addition and all vertices in the same column need to be processed to generate the final reduced value. In this case, while we cannot achieve the parallelism among rows, multiple columns can be processed in parallel. Thus, with a $N \times N$ crossbar, the parallelism achieved is N.

Figure 11 shows the architecture of one GraphR node. GraphR is a ReRAM-based memory module

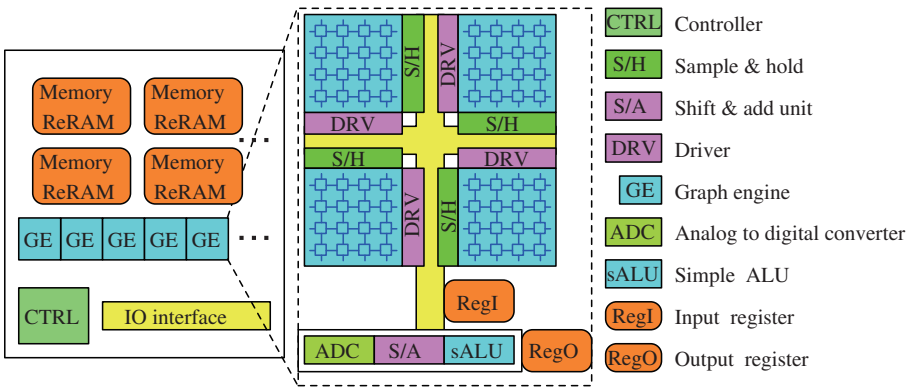

Figure 11 (Color online) GraphR architecture.

that performs efficient in-memory graph processing. It contains two key components: memory ReRAM and GE. Memory ReRAM stores graph data in original compressed sparse representation. GEs perform efficient matrix-vector multiplications on matrix representation. The execution first loads the edges from memory ReRAM with sequential read, thanks to the pre-processing step based on architectural parameters, e.g., the number of GEs contained in the node, and the size of the crossbar. Then the computation is performed in matrix form by the GEs. Intuitively, the sparse data for each subgraph must be "decompressed" into a small matrix for computation. A GE contains a number of ReRAM CBs, drivers (DRVs), sample and hold (S/H) components placed in mesh, and they are connected with analog to digital converter (ADC), shift and add units (S/A) and simple algorithmic and logic units (sALU). The input and output registers (RegI/RegO) are used to cache data flow.

4.1.2 *GraphSAR*

With high sparsity, even with small crossbars, GraphR still leads to considerable ineffectual computations. This is particularly a challenge for the real-world graphs that often have the skewed degree distribution. To address this problem, GraphSAR [102] is proposed based on the insights that the sparsity can be adjusted with graph reordering. Still considering the matrix view of a graph, if an update of a vertex is computed using all non-zeros in a column, it is possible to "move" the non-zeros close to each other so that the subgraphs covered by the GEs become denser, thereby reducing the computation waste.

GraphSAR uses a random-wordline-regular-bitline (RdWRgB) vertex mapping to guide the design of ReRAM based graph analytics accelerators. The Spara architecture is a new ReRAM-based accelerator for sparse graph analytics applications. It maximizes the workload density of ReRAM crossbars dynamically. The paper proposed an effective and fast preprocessing method to generate a CSR representation that fits the RdWRgB scheme. This representation enables quick access to the active data.

4.2 HMC/HBM-based graph processing accelerator

4.2.1 *Tesseract*

Tesseract [47] is a HMC-based graph processing accelerator with HMCs. The graph is partitioned among all the HMCs and the in-order cores embedded in each cube process the local subgraphs in parallel. During the execution, remote data may be accessed depending on the partition, which incurs the inter-cube communication. Tesseract is based on the vertex program model, providing low-level primitives to express the graph algorithms. For each vertex, the runtime system iterates over all its edges/neighbors and executes a `put` function for each of them. The signature of this `put` function is `put(id, void* func, void* arg, size_t arg_size, void* prefethch_addr)`. The semantics is the following. A function call `func` with argument `arg` is executed on the id-th cube. The execution can be one of the two cases: (1) a remote function call if the destination vertex is assigned to a different cube from source vertex; or otherwise (2) a local function call that only uses the data of the local subgraph. At the end of each iteration, a `barrier` ensures that all operations in the current iteration are performed before the start of the next iteration.

For clarity, Figure 12(a) shows the source of inter-cube communication in an adjacency matrix view. The rows and columns correspond to the vertices, if an edge between two vertices intersection of the row

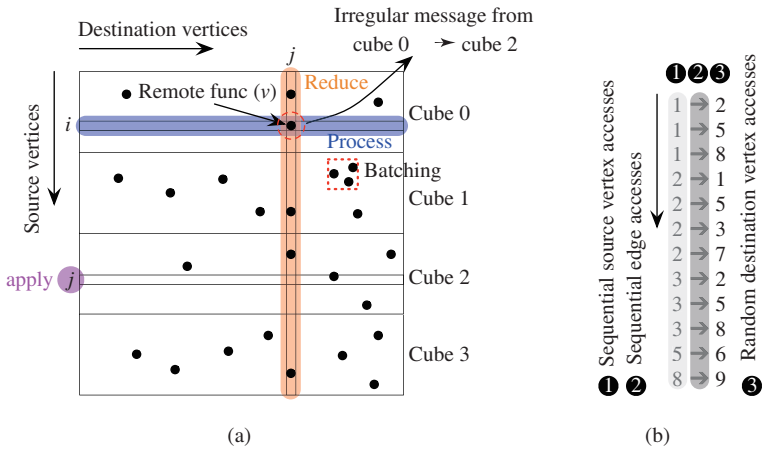

Figure 12 (Color online) Tesseract's access patterns. (a) Inter-cube communications; (b) intra-cube accesses.

and column contains a non-zero value, denoted by a dot. In this example, the vertices are partitioned among four cubes—each cube is assigned with a set of rows. The circled dot represents an edge from a vertex in cube 0 to a vertex in cube 2: $(v_i \rightarrow v_j)$. During execution, if the value of the vertex in cube 0 is updated, it will be propagated to the vertex in cube 2, resulting in an inter-cube message from cube 0 to cube 2. The communication pattern and amount among the cubes is determined by the graph structure. While intuitive, the execution will likely lead to the small and irregular inter-cube messages that affect the performance in two ways. First, since the time of message arrival is not predictable (a function of the graph data), the destination cube is interrupted to process the received messages. Even if such messages can be processed in batch (indicated as the square in Figure 12(a)), the benefit is not guaranteed. Second, Tesseract introduce one inter-cube message for each cross-cube edge after the partition, leading to excessive communication between cubes. We will explain that it can be reduced drastically.

A more subtle implication of the irregular communication between cubes is that it can affect the load balancing and hardware utilization. As discussed before, the communication depends on the graph data, and thus nothing prevents the pathological scenario when multiple messages are sent to the same cube from different senders. In this case, the receiver cube's message queue may become full, which will prevent receiving further messages. In essence, it generates the "feedbacks" through the interconnection to prevent senders from generating more messages. At this point, we can imagine that some cubes are busy with processing the received messages and cannot catch up with processing its own subgraph, while other cubes may be idle and waiting for the finish of the current iteration. Finally, from the hardware and energy consumption perspective, the dynamic communication pattern leads to excessive energy consumption of inter-cube links because it prevents the energy saving optimizations. To be specific, to save energy, each inter-cube link can be set to a low-power state (e.g., the Power-Down mode in HMC [45]). However, it is only feasible when the time of receiving message is known. In another word, such optimization is not possible when a message can go through the link at any time.

Finally, let us consider the data access inside a cube. Figure 12 shows an example of the intra-cube data movement. When an edge is processed, if its destination is assigned to the local cube, local applying is performed. In this case, a random write is incurred, which will interfere with the vertex and edge array accesses with good locality. Specifically, accesses to vertex array (①) and edge array (②) are sequential reads. However, the accesses to compute array for the destination vertices are random (③). This problem also exists in the conventional architecture when executing the vertex programs. The key question is that, if we had a chance to modify the hardware, is it possible to eliminate such interference? Similarly, the remote function call initiated by a remote cube will also introduce random accesses.

4.2.2 *Reducing data movement*

Earlier we mention that the inter-cube communication in Tesseract is excessive. Here we try to understand the problem better by considering the graph partition of Tesseract in a matrix view. Considering Figure 13(a), in Tesseract, each cube is assigned with a set of vertices based on the vertex-centric partition, and this corresponds to a set of rows. The edges, which are the non-zero elements in the matrix,

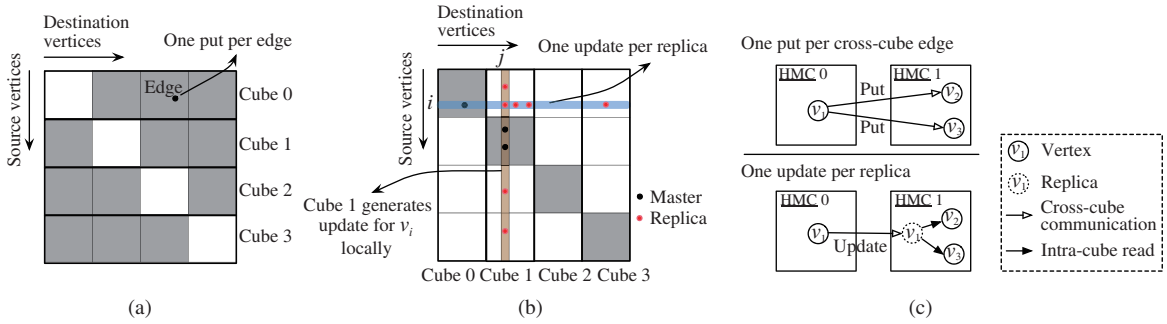

Figure 13 (Color online) (a) Graph partition for vertex program, (b) source-cut in matrix view and (c) source-cut in graph view.

are denoted as black dots. Such partition cuts the matrix into grids, each of which contains the set of edges from vertices assigned to cube i to vertices assigned to cube j. It is similar to the concept in GridGraph [103]. With N cubes, the whole matrix is divided into N^2 grids. Among them, the grids on the diagonal contain the local edges, whose source and destination vertex are assigned to the same cube. In Tesseract, each non-local edge incurs a cross-cube communication. If we consider the region in the matrix that can potentially generate such inter-cube communication, they are the gray grids except for the diagonal ones. Assuming that edges distribute in the graph uniformly, the amount of cross-cube communication in one iteration is $O(N(N-1)\frac{|E|}{N^2}) = O(\frac{(N-1)}{N}|E|)$.

To reduce communication, Zhang et al. [104] proposed a new graph partition strategy named source-cut and the GraphP architecture. According to this method, if a vertex (e.g., v_j) is assigned to a cube (e.g., cube 1), all the incoming edges of v_j are also assigned to the same cube, as shown in Figure 13(b). Different from Tesseract, the matrix is cut vertically: each cube is assigned with a set of columns, instead of rows. To propagate the value of the source vertex through an edge to the destination, a replica (denoted as red dot) is generated if a cube only contains the edge and its destination vertex. The masters (denoted as black dot) are the vertices in a cube that serve as the destination. With this data partition policy, the column of v_j corresponds to v_j's all incoming edges and neighbors, and therefore, v_j's update can be computed locally. The sources of edges in a column can be masters (black dot) or replicas (red dot).

Interestingly, the amount of inter-cube communication can be reduced with the source-cut graph partition method. The key reason is that, in source-cue, the communication is incurred when the replica synchronization is performed. Specifically, the values of all replicas in other cubes are updated with the new value of the master vertex. In the matrix view, it means that each master vertex in the diagonal grids updates its replicas in other cubes in the same row. In Figure 13(b), consider the master vertex v_i in cube 0. With source-cut, cube 0 sends v_i's value to both cube 1 and cube 3, but not cube 2, since cube 2 does not have any edge from v_i. In such replica synchronization, only one message is sent from cube 0 to cube 1, even if there are three edges from v_i to different vertices in cube 1. In essence, the reason for the reduced communication is: the source-cut requires one update per replica while the graph partition based on edge cut (used in Tesseract) would incur one put per cross-cube edge, shown in Figure 13(c).

GraphP is an important HMC-based accelerator design with new data partition and programming model. While the ideas seem to be intuitive, the design identifies data movement as performance bottleneck in HMC-based architectures. It is especially serious to graph processing and other irregular applications with random data access and high memory bandwidth requirement. GraphP is the first design that takes the data partition as first-order design consideration. The results show that the amount of communication can be reduced without sacrificing the programmability. There is another line of design principle in other graph accelerators [105] that aims for best performance and considers computer architecture first. The consequence is that changes in programming interface are nontrivial: the design contains architecture specific optimization and asks the programmers to modify the existing code heavily to be efficient.

4.2.3 *Enabling regular data movement*

Zhuo et al. [106] proposed GraphQ, the first multi-node HMC-based graph processing architecture built on Tesseract. The key insights are shown in Figure 14 with the adjacency matrix. First, GraphQ executes the `reduce` function in the source cube. Specifically, the source cube locally performs the reduction and generates the update for each destination vertex from the source vertices of edges in the same column.

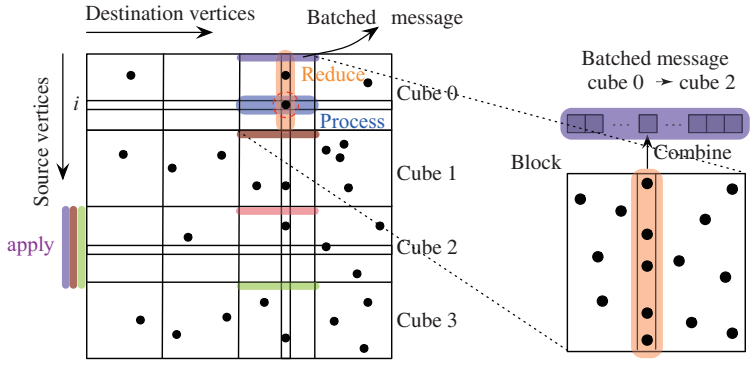

Figure 14 (Color online) Batched communications.

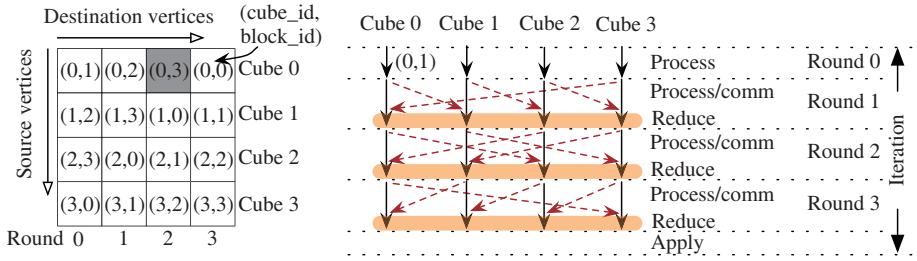

Figure 15 (Color online) Overlapped computation and communications. (a) GraphQ execution; (b) regular and overlapped communications.

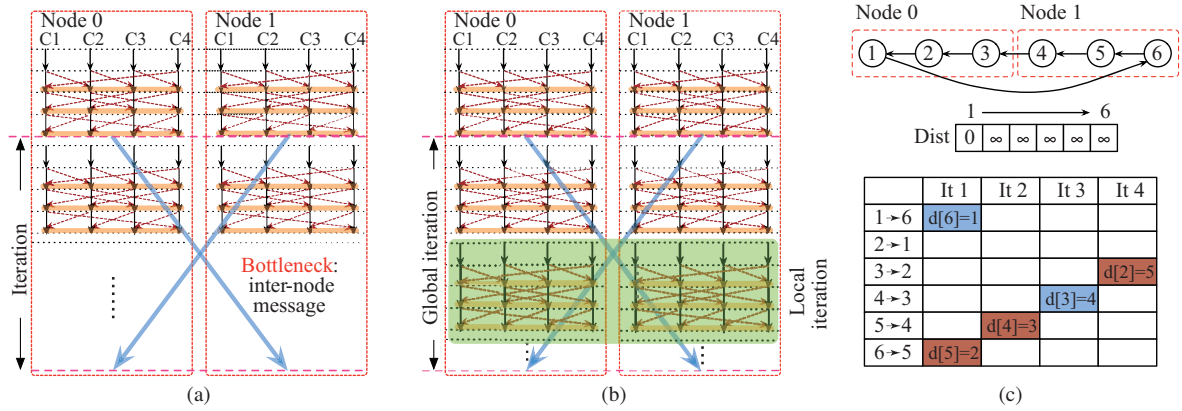

Figure 16 (Color online) Hybrid execution model. (a) GraphQ with batched and overlapped inter-node communication; (b) GraphQ with hybrid execution model; (c) insights of hybrid model by example.

It avoids sending the function and parameters to a remote cube for each edge. Second, we generate all messages for the same remote cube together, so that they can be batched in a batched message. To support this, the whole graph is partitioned into blocks, each of which contains the edges that will contribute to the batched message between a pair of cubes. For example, the third block in the first row will generate a batched message from cube 0 to cube 2.

The batched communication enables the new optimization to support the overlap of communication and computation. The insight is illustrated in Figure 15. We use the (cube_id, block_id) pair to indicate the source and destination of the batched messages. The order of batched messages is determined by the order of blocks in each cube, which is indicated from left to right. This execution model is called as rounded execution, because each iteration is separated into M rounds, and M is equal to the number of cubes. Thus an iteration contains four rounds with four cubes. The key advantage is that, each cube only generates one batched message for one remote cube at the end of a round.

To overcome inter-node communication bottleneck shown in Figure 16(a), GraphQ adopts a bandwidth-aware hybrid execution model, which performs potentially useful computation during idle time. Fig-

ure 16(b) demonstrates the idea with a concrete example. After each node finishes the execution of an iteration and while it is waiting for the batched message from a remote cube, it can simply run more iterations based on local subgraph. In this fashion, the idle time in each cube is not wasted because the computation can be performed opportunistically. We call a normal iteration a global iteration, which contains a number of local iterations. Within a global iteration, after receiving the most recent remote updates, the cube performs the first local iteration. During waiting, it performs the local iterations using locally available subgraph data. Thus, each node essentially executes multiple local iterations "asynchronously" within the cube before a global synchronization.

To implement the batched and overlapped inter-cube communication, GraphQ defines a set of low level primitives for synchronization (e.g., non-blocking inter-cube and inter-node communication) and buffer management. The paper also studies the memory overhead due to the send buffer for batched messages and understands its trade-off between performance overhead of processing received small messages.

5 PIM and NDP architectures for machine learning acceleration

5.1 ReRAM-based machine learning accelerator

5.1.1 *Accelerator for inference*

Given that the computation kernel in DNNs can be naturally expressed as matrix-vector multiplication, several ReRAM-based architectures have been developed for machine learning inference acceleration.

PRIME [76] is one of the first architectures for efficient neural network computation built upon ReRAM crossbar arrays. The design focuses on how to map the neural network computation from the software program down to hardware. The architecture leverages a portion of memory arrays besides normal memory to enable in-memory neural network inference acceleration. It is the first holistic solution with a software interface, and detailed architecture and circuit organization to allow the ReRAM arrays to be dynamically reconfigured between memory and accelerators. The architecture precisely leverages the advantage of ReRAM's compute capability: the same ReRAM memory can store the model weights and perform matrix-vector multiplication as the kernel for DNNs during inference. The execution does not move any data for the weights, significantly reduce the data movements. An inherit limitation of using ReRAM crossbar for computation is the precision, i.e., the number of bits supported. PRIME's results demonstrate that the inference task is quite resilient for the lower precision. For a number of large multilayer perceptrons (MLPs) and CNNs, the state-of-the-art performance can be achieved on varieties of deep learning applications. The energy saving of PRIME is significant, which attributes to both the efficient and low-cost computation using ReRAM and the reduction of data movement. The PRIME architecture can work in a stand-alone fashion without the involvement of another dedicated processor. Therefore, the hardware area overhead is also minimum. In terms of manufacturing in practice, the design also incurs very low cost. Compared to HMC/HBM-like structures, the architecture mostly deals with the memory system design and does not require sophisticated implementation considerations when integrating compute logic in 3D stacking.

In particular, the PRIME architecture is important because it provides a clear flow of how to map DNNs models described by programs to hardware and execute on the ReRAM crossbar. The whole procedure is composed of three steps: (1) expressing the models and controlling the hardware configuration with a programming interface, (2) compilation and optimization, and (3) hardware execution. PRIME provides the convenient application programming interface (APIs) to allow programmers to (1) specify how to map the DNNs to ReRAM subarrays, (2) write the model weights into mats, (3) set up the data paths of the ReRAM subarrays, (4) initiate execution, and (5) collect the inference results. It assumes that the models are pre-trained and can be used as the input for certain APIs. In the second compilation and optimization step, the DNNs are mapped to the ReRAM subarrays based on the user specification with certain optimizations for input data allocation. This step produces the specification for weights mapping, data path configuration, and execution instructions to manage data and control flow. The execution step performs the inference based on such a specification. The PRIME controller writes the weights to the dedicated and known addresses in the subarrays, then configures the peripheral circuits accordingly, and finally, orchestrates the data movements into or out of subarrays at execution time.

The compilation and optimization step is important for achieving high performance, which contains both DNN mapping and data placement. For DNN mapping, different strategies are applied based on the

scale of networks. The small-scale DNNs are replicated to the independent portions of the mat to maximize the performance gain. This strategy can be applied to both fully-connected and convolution layers. The medium-scale DNNs are split into smaller ones, similar to graph partition. After the computation, the results are simply merged. Note that there is extra computation to generate the merged results. The large-scale DNNs may use multiple banks, the optimization focuses on streamlining the data movements among each other so that the execution can be performed in a pipelined fashion with high throughput. To achieve bank-level parallelism and data placement, for small or medium networks, when they can fit into one bank, the subarrays in all the banks are configured in the same way and run in parallel.

ISAAC [75] is another ReRAM-based DNN inference accelerator that was proposed concurrently with PRIME. While PRIME mainly focuses on providing a complete software and hardware solution, ISACC adopts a more sophisticated pipelined design to improve the throughput. The main idea is to divide a layer into small tiles and establish data flow between two consecutive layers at the tile level. It means that the next layer can start the computation based on the partial results of the previous, without waiting for the whole outcome. In some senses, it fuses the execution of multiple layers. The results show that the deep pipeline can indeed achieve considerable performance gains. Specifically, a large amount of input data (either the input of the previous layer or input to the network) can be fed into the pipeline continuously. After the initial cycles to fill the pipeline, the outputs can be produced each cycle. Note that this design may not be suitable for the training phase, because the weights are updated at the end of a batch. Thus, the inputs of the next batch can be only processed based on the updated weights. In another word, it is unlikely to achieve a large amount of consecutive input data to establish the pipeline.

Another drawback of ISSAC is due to the potential pipeline bubble that causes execution stall. We elaborate on the issue using an example. Consider a hypothetical network in which all kernels have size $2 \times 2 \times 1$. One point p in layer l_5 will depend on 4, 16, 64, 256 points in layer l_4, l_3, l_2 and l_1. It means that the computation to produce p is stalled when any of the 340 $(= 4 + 16 + 64 + 256)$ points is delayed. Moreover, certain computations can depend on the value of p, and they will be also stalled. Obviously, such data dependencies are complex in the training phase due to the non-linear structure of the modern DNNs. For example, layer l may not only depend on layer $l - 1$, but also from earlier layers. The issue of the pipeline bubble is acknowledged in [75]. As discussed above, this also explains why ISAAC is not designed for training.

Later, an improved design of ISSAC, Newton [107], was proposed to address two hardware implementation issues. First, the ADCs consume high hardware power and area budget. Second, the original design partitions the hardware resources for the worst case. To solve the two problems, Newton introduces multiple new ideas for different levels of the tile hierarchy. A key factor determining the ADC hardware cost is precision. Thus, the design adopts the precision based on the computations' requirements. As long as reducing precision does not degrade accuracy, ADC precision can be adjusted to a lower level to save energy. On the other side, the hardware overhead is also reduced by a more friendly divide-and-conquer numeric algorithms.

A key consideration of the ReRAM-based DNN accelerator is that although the weights stored in ReRAM crossbar cells can be either positive or negative, the in-situ computation assumes all cells on each crossbar column hold the values with the same sign, i.e., all positive or all negative. The existing designs use two approaches to tackle the problem. PRIME uses two ReRAM crossbars to hold the positive magnitude and negative magnitude of weights separately, doubling the ReRAM portion of hardware cost. It is also used several other designs [108–111]. In contrast, ISAAC [75] adds an offset to weights so that all values become positive. While keeping the number of crossbars the same, the latter approach introduces additional hardware costs for the peripheral circuits by adding extra offset circuits.

In ReRAM, the computations are conducted on multi-level cell (MLC) that have limited precision. The implication is that, in general, the ReRAM-based accelerator is vulnerable to noises. While it can be mitigated by hardware supports, a natural solution is to adopt the binarized neural network (BNN). Inherently, it is a hardware-friendly model that can dramatically reduce the computation and storage overheads while at the same time being quite vulnerable to noises. However, directly applying this idea to ReRAM faces obstacles. It is because, XNOR, the key operation in BNNs, cannot be directly computed in ReRAM due to its nonlinear behavior. To enable efficient processing of BNNs in ReRAM, Song et al. [112] modified the BNN algorithm to enable direct computation of XNOR, POPCOUNT and POOL based on ReRAM cells. The complementary resistive cell (CRC) design is proposed to efficiently conduct XNOR operations and optimize the pipeline design with decoupled buffer and computation stages. XNORBIN [113] developed a BNN accelerator that exploits data reuse opportunities and the

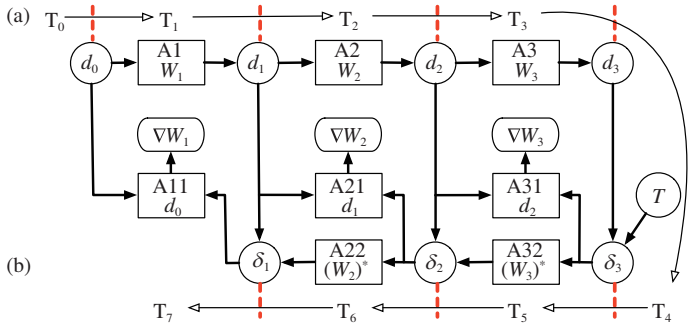

Figure 17 (Color online) PipeLayer for training. (a) Forward; (b) backward.

parallelism among primitive operations. Conti et al. [114] proposed an XNOR-based binarized neural network computing engine that optimizes the data path for XNOR-and-popcount operations. Jafari et al. [115] proposed an accelerator cluster architecture to support large models. Recently, Andri et al. [116] designed a highly energy-efficient BNN accelerator that achieves several hundreds of TOPS/W efficiency.

5.1.2 *Accelerator for training*

PipeLayer [77] is the first architecture that supports both DNN inference and training. The supporting training phase is more complex and challenging because it involves weight updates and complex data dependencies. For inference tasks, a ReRAM-based accelerator can directly compute the weights obtained after training, thereby only need to write ReRAM once before the execution. It is challenging to achieve high throughput for DNNs with a large number of layers, and PipeLayer adopts a novel and coarser grained pipelined architecture. Different from ISACC [75], the pipeline is established between layers, rather than the small tiles inside layers. We have analyzed that the ISACC design will actually negatively affect performance in training. With the pipeline in PipeLayer, the data can be continuously accepted by the accelerated and get processed in consecutive cycles. Based on the pipeline, the design explores different data input and kernel mapping schemes to achieve the balance between hardware cost and data processing parallelism. PipeLayer also makes an effort to reduce the overhead of ADCs and DACs by replacing the voltage-level based scheme for data input by a spike-based scheme. This technique can be also applied to accelerators for inference. For the spike-based scheme, more cycles are required to inject data, but the impact on performance is offset by the pipelined architecture among multiple layers. Note that ISSAC [75] also uses the spike-based inputs, which can also avoid DACs. The difference is that, the ADCs of integration and fire component are also eliminated in PipeLayer.

Similar to PRIME, the PipeLayer architecture has two regions of the ReRAM-based main memory: morphable subarrays (Morp) and memory subarrays (Mem). The PipeLayer configuration to execute the training task of a 3-layer CNN is shown in Figure 17. A layer of morphable subarrays is represented by the rectangles. The memory subarrays that store the intermediate results transferred between morphable subarrays for different layers are indicated as circles. We can observe the data dependencies between forward and backward propagation. Starting from forward computation, the initial cycle is T_0, in cycle T_1, the input (d_0) enters A1 (morphable subarrays), which perform the matrix-vector multiplication. The results are written to a memory subarray, d_1 at the end of T_1. The system state between two consecutive cycles is indicated by the red dashed lines. The data dependencies are marked by the solid lines between rectangles (morphable subarrays) and circles (memory subarrays).

Before the backward computation starting at T_4, the results of forward computation are stored in d_3. The errors (δ_l) (l is the layer) and partial derivatives to b and W (∇b_l and ∇W_l) are produced by the backward computations. The error for the last (third) layer (δ_3) is computed in T_4 at the first step. It is stored in the memory subarrays and will be used as the input of the next cycle's computation. In T_5, two computations can be performed in parallel: (1) the partial derivatives (∇W_3) are generated by previous results in d_2 and δ_3; (2) the errors (δ_2) of the second layer are computed from δ_3. Both of them depend on δ_3 computed in T_4. ∇W_3 is stored in memory subarrays. They are used to update weights in A3 and A32 later. Finally, T_7 computes the partial derivatives for the first layer (∇W_1), which are stored in memory subarrays. In training, batch size (B) indicates the number of data samples processed together before a weight update. If $B = 1$, ∇W_1, ∇W_2 and ∇W_3 are used to update the weights in A1, A2, A3 and A22, A32. If $B > 1$, ∇W_1, ∇W_2 and ∇W_3 are stored in the buffers, and later the average of

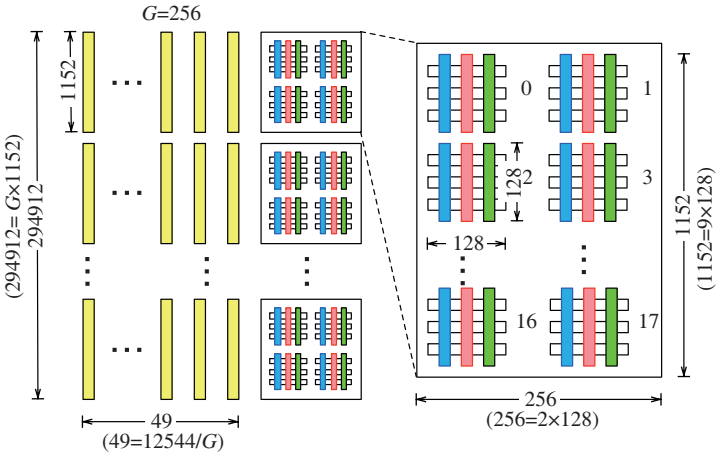

Figure 18 (Color online) Balanced scheme for data input and kernel mapping.

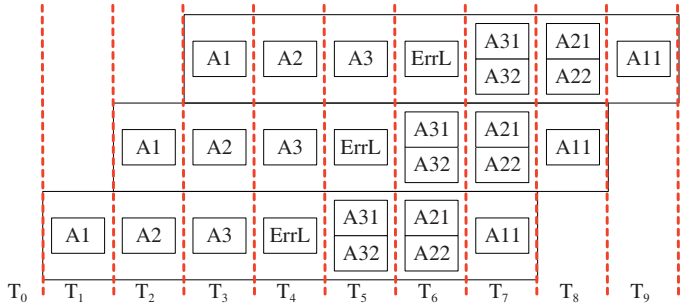

Figure 19 (Color online) Training pipeline in PipeLayer.

all partial derivatives is computed accordingly.

The large models can be partitioned to fit into the ReRAM array size. In PipeLayer, the 1152×256 matrix is decomposed to a group of 18 ($=9 \times 2$) matrices and each of them is mapped to a 128×128 ReRAM array. It is shown in the right part of Figure 18. The right results can be obtained by horizontally collecting array outputs and vertically summing them. To systematically capture the design space, the authors define a metric called parallelism granularity (G) to represent the number of duplicated copies of ReRAM arrays that store the same weights. If $G = 1$, the design is the same as the naive scheme. If $G = 12544$, the results of a layer can be produced in just one cycle but the hardware cost is prohibitive. Thus, the notion of parallelism granularity allows the designer to trade-off between the hardware resource of ReRAM array and performance. A good trade-off requires a carefully chosen G. Figure 18 shows an example with $G = 256$.

The training phase processes the input data in batchs. For the inputs of the same batch, they are all trained based on the weights at the start of the batch. In another word, the inputs in the same batch cannot observe the weight changes by each other. The weight updates are kept and only applied to at the end of a batch. It means that there is no dependency among data inputs of the same batch. The reason that PipeLayer can achieve performance gain is that the batch size is typically much larger than 1. Without such assumption, each input needs to be processed sequentially. Figure 19 illustrates an example of the pipelined execution.

5.2 HMC/HBM-based machine learning accelerator

Neurocube [117] is the first programmable and scalable machine learning accelerator based on 3D high-density memory such as HMC. The efficient neural computation supports are implemented in HMC's logic die. The architecture is composed of a cluster of processing engines, which are connected by a 2D mesh network. The major performance gain is achieved because the processing engines can access multiple vaults in parallel.

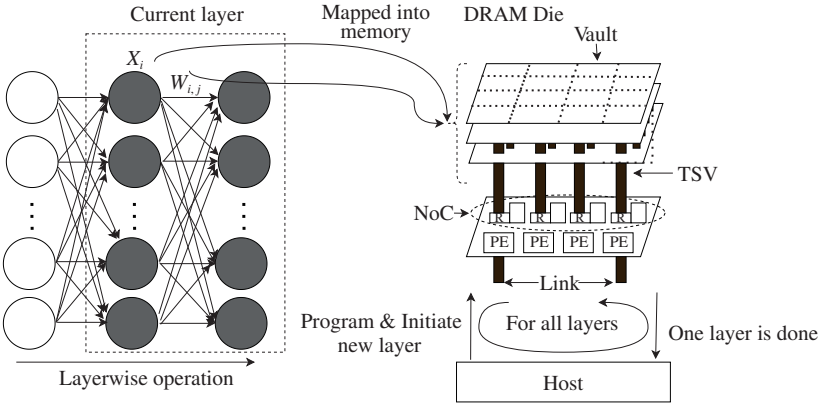

Figure 20 Neurocube SW/HW flow.

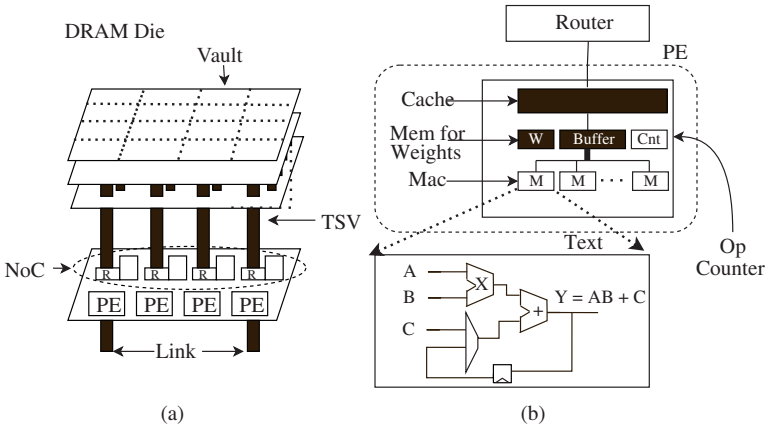

Figure 21 Neurocube architecture. (a) The neurocube architecture; (b) processing element (PE) architecture.

Figure 20 shows the flow of how a DNN layer is mapped to the different components of Neurocube architecture. The weights of the network layer are stored in the DRAM stack. The architecture sets up the address of the weights in different layers before execution. This address information, together with the network structures, determines the data movement patterns. With both known, the hardware can optimize the data movement paths between the DRAM layers and the logic layer to perform neural computation. Specifically, the data movement paths are compiled into the state machine descriptions that drive the programmable neurosequence generators (PNGs), which are integrated with the vault controllers. To start the execution, the host loads the state machine to stream data to the compute layer. Then the operations are performed in a data-driven manner.

Figure 21 shows the major components of the Neurocube architecture. They are implemented in the logic die of an HMC. With high-bandwidth TSVs, multiple processing elements can communicate concurrently with multiple DRAM vaults. The host communicates with the Neurocube through external links of the HMC to control the execution of specific neural network structures. The information provided includes the dimension of layers, the number of layers, and types of layers. Overall, the Neurocube architecture is composed of a global controller, PNG for DRAM, routers for a 2D-mesh network on chip, and processing elements.

The processing engines are interconnected by a 2D mesh network as shown in Figure 22. Figure 22(c) illustrates a block diagram of a router. Each processing engine is connected to a single router, which has six input and output channels (four for neighboring routers and two for PE and memory). The router is wormhole switched and uses the credit-based flow control. It maintains a packet buffer with 16 entries for both input and output channels. The deterministic X-Y routing is implemented with a routing table. To arbitrate among the inputs requesting the same outputs, a rotating daisy chain priority scheme is used, where the priority is updated every cycle. Based on the results, the network-on-chip (NoC) has an important impact on the performance. It is especially the case for the layer with dense connections. However, there is no significant throughput degradation from the locally connected layer to the fully

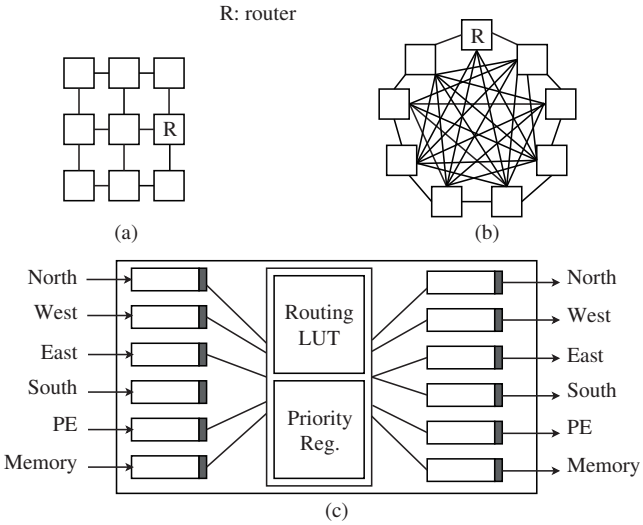

Figure 22 Neurocube interconnection. (a) 2D mesh NoC; (b) 2D fully connected NoC; (c) router design for 2D mesh NoC.

connected layer. It is because the lateral traffic is not injected into the NoC.

6 Research directions

In this section, we discuss several potential future research directions based on the understanding of the state-of-the-art architectures.

Applying the techniques in distributed graph processing. While at a different level, PIM architecture shares a common abstraction with distributed graph processing: graph processing can be performed with multiple nodes connected by certain communication links, while each node has its own local memory and computation capability. For distributed graph processing, each node corresponds to a machine with multi-cores and the memory hierarchy. In high performance clusters, the machines are connected by remote direct memory access (RDMA) network. The latency and bandwidth of accessing local memory are far cheaper than accessing remote memory in another machine. For PIM architectures such as HMC, each node is a memory cube, which contains stacks of HBM and computation logic. The memory cubes are connected by SerDes links, with 120 GB/s per link, and each cube can support up to 4 links. Although the total external bandwidth between memory cubes is higher than internal bandwidth, recent studies [47, 104, 106] have shown that the remote communication is still the bottleneck. We can apply ideas in the distributed graph processing such as [118] to further reduce the communication amount and improve performance.

Accelerator for graph neural network (GNN). GNNs reduce the dimensionality of the feature representations of graph data by aggregating the features of connected nodes and transforming them using shared machine learning layers. Graph convolutional neural network characteristics are significantly different from conventional DNNs such as CNNs and RNNs. GNNs explore random memory accesses and irregular or unstructured computations due to graph traversal during the aggregation phase, and sequential and structured computations during the transformation phase. As a result, the inference operation incurs hybrid compute and memory access patterns. Adopting GNNs into crossbar architectures is difficult. The existing ReRAM-based DNN and graph processing accelerator architectures can be potentially leveraged to accelerate the combination and aggregation kernels, but applying them simultaneously into GNNs as a unified architecture remains challenging. The ReRAM-based DNN accelerators map weight parameters of all layers into crossbars. However, features in a layer for GNNs are operated not only on weight parameters but also on sparse graph data. This makes conventional DNN accelerator architectures no longer suitable. Moreover, the graph sparsity leads to low efficiency of hardware. GNNs often have $100\times$ to $1000\times$ vertex dimensions than traditional graph algorithms. Aggregating such a multi-dimension vertex will further exaggerate the ineffective computation due to the crossbar sparsity.

From graph computation to graph mining. Different from the traditional iterative graph computation (e.g., PageRank, BFS, SSSP) with simple computations, graph mining applications are computation-

intensive [119–124]. The goal of graph mining is to find all embeddings that match specific patterns. The tasks are more challenging since the number of embeddings could be large. For example, in WikiVote, a small graph with merely 7k vertices, the number of vertex-induced 5-chain embeddings can reach 71 billion. In a state-of-the-art graph mining algorithm [121], the frequent intersection operations between two edge lists for constructing patterns pose a key challenge. We find that the execution time for the intersection operations is substantial in mining several representative graph patterns. Thus, to accelerate graph mining, efficient execution of intersection operation using PIM is an interesting and open problem.

Algorithm and hardware co-design. As discussed before, the ReRAM-based DNN accelerators need to store both positive and negative weights. The solutions in the current designs—using two crossbars (PRIME) or adding offsets (ISSAC)—are both not ideal. Alternatively, it may be possible to enforce exactly what is assumed in the in-situ computation—ensuring the pattern that all weights in the same column of a crossbar have the same sign. This approach takes the opportunity of algorithm and hardware co-design, and is motivated by the capability of the powerful alternating direction method of multipliers (ADMM) regularized optimization [125], which is exactly able to enforce patterns in DNN training while maintaining high accuracy. Based on this idea, we can train a DNN model using ADMM with our novel constraints such that the weights mapped to the same crossbar columns are all positive or negative. In general, the ADMM-based structured training has been shown as a powerful tool to enforce various structures with different benefits either in hardware [126] or compiler level [127].

Principled architecture design considering both communication and computation. To design efficient architecture for training, the fundamental problem is how to use multiple nodes (e.g., HMC) to achieve high throughput in processing a large amount of training data. Different from inference, each accelerator is normally abstracted as a "black box" with certain computing capability. In this scenario, the communication between accelerators becomes a key problem. Specifically, we need to consider the sophisticated interactions between model architecture (layer types and how layers are connected), parallelism configuration (data or model or hybrid parallelism), and architecture (computation, communication, and synchronization cost). With different parallelism configurations, the communication can happen at different phases (forward, backward, and gradient) with data transfer amount determined by layer structure. The crux of the problem is exhaustively enumerating all possible partitions of the tensors involved in the three phases. This is why a principled approach is required. The recent studies HyPar [128] and AccPar [129] advanced state-of-the-art using a systematic approach with a solid mathematical foundation. It is interesting to consider the problem in the context of HMC/HBM with their unique hardware and communication constraints.

Beyond graph processing and machine learning. While significant research efforts have been devoted to the ReRAM-based graph processing and machine learning acceleration, there are other important domains that have not been thoroughly investigated. For example, it is important to accelerate the scientific computing problems that model a complex system with partial differential equations (PDEs) to understand the natural phenomena in science [130, 131], or the design and decision-making of engineered systems [132, 133]. Most problems in continuous mathematics modeled by PDEs cannot be solved directly. In practice, the PDEs are converted to a linear system $A\boldsymbol{x} = \boldsymbol{b}$, and then solved through an iterative solver that ultimately converges to a numerical solution [134, 135]. To obtain an acceptable answer where the residual is less than a desired threshold, intensive computing power [136, 137] is required to perform the floating-point SpMV—the key computation kernel. Performing floating-point computation on ReRAM leads to much higher hardware cost and execution time than fixed-point due to the greater value range. To make it practical, the current solution truncates the higher bits in the exponent, e.g., using the low 6 bits or module 64 of the exponent to represent each original value. This ad-hoc solution does not ensure convergence of iterative computation due to the inaccurate exponent values, while unnecessarily paying the hardware and execution time cost for the full precision of fractions. Thus, a promising research direction is to investigate the efficient floating-point computation based on the nature of applications using ReRAM.

7 Conclusion

As we approach the end of Moore's law and Dennard scaling, it is important to explore the potential of new memory technologies to improve the performance of key applications. One of the main challenges for the domain-specific architecture is the high cost of data movement. To reduce the data movement

and computing cost, this paper surveys domain-specific architectures built from two emerging memory technologies. HMC and HBM can reduce data movement between memory and computation by placing computing logic inside memory dies. On the other hand, the emerging non-volatile memory, metal-oxide ReRAM has been considered as a promising candidate for future memory architecture due to its high density, fast read access and low leakage power. The key feature is ReRAM's capability to perform the inherently parallel in-situ matrix-vector multiplication in the analog domain. We focus on the domain-specific architectures for two important applications—graph processing and machine learning acceleration. Based on the understanding of the recent architectures and our research experience, we also discuss several potential research directions.

References

1 Hennessy J L, Patterson D A. A new golden age for computer architecture. Commun ACM, 2019, 62: 48–60
2 Krizhevsky A, Sutskever I, Hinton G E. Imagenet classification with deep convolutional neural networks. In: Proceedings of the 25th International Conference on Neural Information Processing Systems, 2012. 1097–1105
3 Farmahini-Farahani A, Ahn J H, Morrow K, et al. NDA: near-DRAM acceleration architecture leveraging commodity DRAM devices and standard memory modules. In: Proceedings of High-Performance Computer Architecture, 2015
4 Hybrid Memory Cube Consortium. Hybrid Memory Cube Specification Version 2.1. Technical Report. 2015
5 Lee D U, Kim K W, Kim K W, et al. A 1.2 V 8 Gb 8-channel 128 GB/s high-bandwidth memory (HBM) stacked DRAM with effective microbump I/O test methods using 29 nm process and TSV. In: Proceedings of IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2014. 432–433
6 Wong H S P, Lee H Y, Yu S, et al. Metal-oxide RRAM. Proc IEEE, 2012, 100: 1951–1970
7 Xia L, Li B, Tang T, et al. MNSIM: simulation platform for memristor-based neuromorphic computing system. IEEE Trans Comput-Aided Des Integr Circuits Syst, 2017, 37: 1009–1022
8 Prezioso M, Merrikh-Bayat F, Hoskins B D, et al. Training and operation of an integrated neuromorphic network based on metal-oxide memristors. Nature, 2015, 521: 61–64
9 Thomas A. Memristor-based neural networks. J Phys D-Appl Phys, 2013, 46: 093001
10 Xiao W, Xue J, Miao Y, et al. TUX^2: distributed graph computation for machine learning. In: Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation, 2017
11 Alexandrescu A, Kirchhoff K. Data-driven graph construction for semi-supervised graph-based learning in NLP. In: Proceedings of Human Language Technology Conference of the North American Chapter of the Association of Computational Linguistics, 2007. 204–211
12 Goyal A, Daumé III H, Guerra R. Fast large-scale approximate graph construction for NLP. In: Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning, 2012. 1069–1080
13 Zesch T, Gurevych I. Analysis of the wikipedia category graph for NLP applications. In: Proceedings of the TextGraphs-2 Workshop (NAACL-HLT 2007), 2007. 1–8
14 Qiu M, Zhang L, Ming Z, et al. Security-aware optimization for ubiquitous computing systems with SEAT graph approach. J Comput Syst Sci, 2013, 79: 518–529
15 Stankovic A M, Calovic M S. Graph oriented algorithm for the steady-state security enhancement in distribution networks. IEEE Trans Power Deliver, 1989, 4: 539–544
16 Wang Y J, Xian M, Liu J, et al. Study of network security evaluation based on attack graph model (in Chinese). J Commun, 2007, 28: 29–34
17 Shun J, Roosta-Khorasani F, Fountoulakis K, et al. Parallel local graph clustering. Proc VLDB Endow, 2016, 9: 1041–1052
18 Schaeffer S E. Survey: graph clustering. Comput Sci Rev, 2007, 1: 27–64
19 Fouss F, Pirotte A, Renders J M, et al. Random-walk computation of similarities between nodes of a graph with application to collaborative recommendation. IEEE Trans Knowl Data Eng, 2007, 19: 355–369
20 Guan Z, Bu J, Mei Q, et al. Personalized tag recommendation using graph-based ranking on multi-type interrelated objects. In: Proceedings of the 32nd International ACM SIGIR Conference on Research and Development in Information Retrieval, 2009. 540–547
21 Lo S, Lin C. WMR—a graph-based algorithm for friend recommendation. In: Proceedings of the 2006 IEEE/WIC/ACM International Conference on Web Intelligence, 2006. 121–128
22 Mirza B J, Keller B J, Ramakrishnan N. Studying recommendation algorithms by graph analysis. J Intell Inf Syst, 2003, 20: 131–160
23 Campbell W M, Dagli C K, Weinstein C J. Social network analysis with content and graphs. Lincoln Laboratory J, 2013, 20: 61–81
24 Tang L, Liu H. Graph mining applications to social network analysis. In: Managing and Mining Graph Data. Berlin: Springer, 2010. 487–513
25 Wang T, Chen Y, Zhang Z, et al. Understanding graph sampling algorithms for social network analysis. In: Proceedings of the 31st International Conference on Distributed Computing Systems Workshops, 2011. 123–128
26 Aittokallio T, Schwikowski B. Graph-based methods for analysing networks in cell biology. Briefings Bioinf, 2006, 7: 243–255
27 Enright A J, Ouzounis C A. BioLayout—an automatic graph layout algorithm for similarity visualization. Bioinformatics, 2001, 17: 853–854
28 Novére N L, Hucka M, Mi H, et al. The systems biology graphical notation. Nat Biotechnol, 2009, 27: 735–741
29 Goodfellow I, Bengio Y, Courville A. Deep Learning. Cambridge: MIT Press, 2016
30 Han S, Pool J, Tran J, et al. Learning both weights and connections for efficient neural network. In: Proceedings of the 28th International Conference on Neural Information Processing Systems, 2015. 1135–1143
31 Wen W, Wu C, Wang Y, et al. Learning structured sparsity in deep neural networks. In: Proceedings of the 30th International Conference on Neural Information Processing Systems, 2016. 2074–2082
32 Park E, Ahn J, Yoo S. Weighted-entropy-based quantization for deep neural networks. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2017. 5456–5464
33 Wu J, Leng C, Wang Y, et al. Quantized convolutional neural networks for mobile devices. In: Proceedings of IEEE

Conference on Computer Vision and Pattern Recognition (CVPR), 2016
34 Alwani M, Chen H, Ferdman M, et al. Fused-layer CNN accelerators. In: Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2016. 1–12
35 Shen Y, Ferdman M, Milder P. Maximizing CNN accelerator efficiency through resource partitioning. In: Proceedings of the 44th Annual International Symposium on Computer Architecture, 2017. 535–547
36 Chen T, Du Z D, Sun N H, et al. DianNao: a small-footprint high-throughput accelerator for ubiquitous machine-learning. In: Proceedings of ACM SIGARCH Computer Architecture News, 2014. 269–284
37 Merolla P A, Arthur J V, Alvarez-Icaza R, et al. A million spiking-neuron integrated circuit with a scalable communication network and interface. Science, 2014, 345: 668–673
38 Sharma H, Park J, Mahajan D, et al. From high-level deep neural models to FPGAs. In: Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2016. 1–12
39 Shen Y, Ferdman M, Milder P. Escher: a CNN accelerator with flexible buffering to minimize off-chip transfer. In: Proceedings of the 25th IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM17). Los Alamitos: IEEE Computer Society, 2017
40 Ovtcharov K, Ruwase O, Kim J Y, et al. Toward accelerating deep learning at scale using specialized hardware in the datacenter. In: Proceedings of IEEE Hot Chips 27 Symposium (HCS), 2015. 1–38
41 Ovtcharov K, Ruwase O, Kim J Y, et al. Accelerating deep convolutional neural networks using specialized hardware. Microsoft Research Whitepaper, 2015, 2: 1–4
42 Sharma H, Park J, Amaro E, et al. Dnnweaver: from high-level deep network models to FPGA acceleration. In: Proceedings of the Workshop on Cognitive Architectures, 2016
43 Waldrop M M. The chips are down for Moore's law. Nature, 2016, 530: 144–147
44 Black B, Annavaram M, Brekelbaum N, et al. Die stacking (3D) microarchitecture. In: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06), 2006. 469–479
45 Hybrid Memory Cube Consortium. Hybrid Memory Cube Specification 2.1, 2015
46 OĆonnor M. Highlights of the high-bandwidth memory (HBM) standard. In: Proceedings of Memory Forum Workshop, 2014
47 Ahn J, Hong S, Yoo S, et al. A scalable processing-in-memory accelerator for parallel graph processing. In: Proceedings of ACM SIGARCH Computer Architecture News, 2015. 105–117
48 Shevgoor M, Kim J S, Chatterjee N, et al. Quantifying the relationship between the power delivery network and architectural policies in a 3D-stacked memory device. In: Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture, 2013. 198–209
49 Kim G, Kim J, Ahn J H, et al. Memory-centric system interconnect design with hybrid memory cubes. In: Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques. Piscataway: IEEE Press, 2013. 145–156
50 Kim J, Dally W, Scott S, et al. Cost-efficient dragonfly topology for large-scale systems. IEEE Micro, 2009, 29: 33–40
51 Kim J, Dally W J, Abts D. Flattened butterfly: a cost-efficient topology for high-radix networks. In: Proceedings of ACM SIGARCH Computer Architecture News, 2007. 126–137
52 Izraelevitz J, Yang J, Zhang L, et al. Basic performance measurements of the Intel Optane DC persistent memory module. 2019. ArXiv:1903.05714
53 Hady F T, Foong A, Veal B, et al. Platform storage performance with 3D XPoint technology. Proc IEEE, 2017, 105: 1822–1833
54 Akinaga H, Shima H. Resistive random access memory (ReRAM) based on metal oxides. Proc IEEE, 2010, 98: 2237–2251
55 Liu W, Pey K L, Raghavan N, et al. Fabrication of RRAM cell using CMOS compatible processes. US Patent App. 13/052,864, 2012
56 Trinh H D, Tsai C Y, Lin H L. Resistive RAM structure and method of fabrication thereof. US Patent 9,978,938, 2018
57 Adam G C, Chrakrabarti B, Nili H, et al. 3D ReRAM arrays and crossbars: fabrication, characterization and applications. In: Proceedings of IEEE 17th International Conference on Nanotechnology (IEEE-NANO), 2017. 844–849
58 Chen W H, Lin W J, Lai L Y, et al. A 16 Mb dual-mode ReRAM macro with sub-14 ns computing-in-memory and memory functions enabled by self-write termination scheme. In: Proceedings of IEEE International Electron Devices Meeting (IEDM), 2017
59 Chang M F, Lin C C, Lee A, et al. A 3T1R nonvolatile TCAM using MLC ReRAM with sub-1 ns search time. In: Proceedings of IEEE International Solid-State Circuits Conference-(ISSCC) Digest of Technical Papers, 2015. 1–3
60 Han R, Huang P, Zhao Y, et al. Demonstration of logic operations in high-performance RRAM crossbar array fabricated by atomic layer deposition technique. Nanoscale Res Lett, 2017, 12: 1–6
61 Kataeva I, Ohtsuka S, Nili H, et al. Towards the development of analog neuromorphic chip prototype with 2.4 m integrated memristors. In: Proceedings of 2019 IEEE International Symposium on Circuits and Systems (ISCAS), 2019. 1–5
62 Bayat F M, Prezioso M, Chakrabarti B, et al. Implementation of multilayer perceptron network with highly uniform passive memristive crossbar circuits. Nature Commun, 2018, 9: 1–7
63 Cai F, Correll J M, Lee S H, et al. A fully integrated reprogrammable memristor-CMOS system for efficient multiply-accumulate operations. Nat Electron, 2019, 2: 290–299
64 Xu C, Niu D, Muralimanohar N, et al. Overcoming the challenges of crossbar resistive memory architectures. In: Proceedings of IEEE 21st International Symposium on High Performance Computer Architecture (HPCA), 2015. 476–488
65 Liu T, Yan T H, Scheuerlein R, et al. A 130.7-mm^2 2-layer 32-Gb ReRAM memory device in 24-nm technology. IEEE J Solid-State Circ, 2014, 49: 140–153
66 Fackenthal R, Kitagawa M, Otsuka W, et al. 19.7 a 16 Gb ReRAM with 200 MB/s write and 1 GB/s read in 27 nm technology. In: Proceedings of IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2014. 338–339
67 Qureshi M K, Karidis J, Franceschini M, et al. Enhancing lifetime and security of PCM-based main memory with start-gap wear leveling. In: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture, 2009. 14–23
68 Lee M J, Lee C B, Lee D, et al. A fast, high-endurance and scalable non-volatile memory device made from asymmetric Ta_2O_{5-x}/TaO_{2-x} bilayer structures. Nat Mater, 2011, 10: 625–630
69 Hsu C, Wang I, Lo C, et al. Self-rectifying bipolar TaO_x/TiO_2 RRAM with superior endurance over 10^{12} cycles for 3D high-density storage-class memory VLSI tech. In: Proceedings of Symposium on VLSI Technology, 2013. 166–167
70 Hu M, Strachan J P, Li Z, et al. Dot-product engine for neuromorphic computing: programming 1T1M crossbar to accelerate matrix-vector multiplication. In: Proceedings of the 53rd ACM/EDAC/IEEE Design Automation Conference (DAC), 2016
71 Hu M, Li H, Wu Q, et al. Hardware realization of BSB recall function using memristor crossbar arrays. In: Proceedings of

the 49th Annual Design Automation Conference, 2012. 498–503
72 Chen Y, Luo T, Liu S, et al. DaDianNao: a machine-learning supercomputer. In: Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture, Cambridge, 2014. 609–622
73 Mahajan D, Park J, Amaro E, et al. TABLA: a unified template-based framework for accelerating statistical machine learning. In: Proceedings of IEEE International Symposium on High Performance Computer Architecture (HPCA), 2016. 14–26
74 Albericio J, Judd P, Hetherington T, et al. Cnvlutin: ineffectual-neuron-free deep neural network computing. In: Proceedings of ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA), 2016. 1–13
75 Shafiee A, Nag A, Muralimanohar N, et al. ISAAC: a convolutional neural network accelerator with in-situ analog arithmetic in crossbars. In: Proceedings of ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA), 2016
76 Chi P, Li S, Xu C, et al. PRIME: a novel processing-in-memory architecture for neural network computation in ReRAM-based main memory. In: Proceedings of ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA), 2016
77 Song L, Qian X, Li H, et al. PipeLayer: a pipelined ReRAM-based accelerator for deep learning. In: Proceedings of IEEE 23rd International Symposium on High Performance Computer Architecture (HPCA), 2017
78 Liu X, Mao M, Liu B, et al. Reno: a high-efficient reconfigurable neuromorphic computing accelerator design. In: Proceedings of the 52nd ACM/EDAC/IEEE Design Automation Conference (DAC), 2015. 1–6
79 Pingali K, Nguyen D, Kulkarni M, et al. The tao of parallelism in algorithms. In: Proceedings of ACM Sigplan Notices, 2011. 12–25
80 Gonzalez J E, Low Y, Gu H, et al. Powergraph: distributed graph-parallel computation on natural graphs. In: Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation, 2012. 17–30
81 Malewicz G, Austern M H, Bik A J, et al. Pregel: a system for large-scale graph processing. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data, 2010
82 Shun J, Blelloch G E. Ligra: a lightweight graph processing framework for shared memory. In: ACM Sigplan Notices, 2013. 135–146
83 Low Y, Bickson D, Gonzalez J, et al. Distributed GraphLab: a framework for machine learning and data mining in the cloud. Proc VLDB Endow, 2012, 5: 716–727
84 Ham T J, Wu L, Sundaram N, et al. Graphicionado: a high-performance and energy-efficient accelerator for graph analytics. In: Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2016. 1–13
85 Lee H, Grosse R, Ranganath R, et al. Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations. In: Proceedings of the 26th Annual International Conference on Machine Learning, 2009
86 Ciresan D, Meier U, Schmidhuber J, et al. Multi-column deep neural networks for image classification. In: Proceedings of IEEE Conference on Computer Vision and Pattern Recognition, 2012
87 Ciresan D C, Meier U, Masci J, et al. Flexible, high performance convolutional neural networks for image classification. In: Proceedings of the 22nd International Joint Conference on Artificial Intelligence, 2011
88 Sermanet P, Chintala S, LeCun Y, et al. Convolutional neural networks applied to house numbers digit classification. In: Proceedings of the 21st International Conference on Pattern Recognition (ICPR2012), 2012
89 Oquab M, Bottou L, Laptev I, et al. Learning and transferring mid-level image representations using convolutional neural networks. In: Proceedings of IEEE Conference on Computer Vision and Pattern Recognition, 2014
90 LeCun Y, Boser B, Denker J S, et al. Backpropagation applied to handwritten zip code recognition. Neural Comput, 1989, 1: 541–551
91 Kim Y. Convolutional neural networks for sentence classification. 2014. ArXiv:1408.5882
92 Howard A G. Some improvements on deep convolutional neural network based image classification. 2013. ArXiv:1312.5402
93 Gong Y, Jia Y Q, Leung T, et al. Deep convolutional ranking for multilabel image annotation. 2013. ArXiv:1312.4894
94 Collobert R, Weston J. A unified architecture for natural language processing: deep neural networks with multitask learning. In: Proceedings of the 25th International Conference on Machine Learning, 2008. 160–167
95 Abdel-Hamid O, Mohamed A, Jiang H, et al. Applying convolutional neural networks concepts to hybrid NN-HMM model for speech recognition. In: Proceedings of IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), 2012
96 Kalchbrenner N, Grefenstette E, Blunsom P, et al. A convolutional neural network for modelling sentences. 2014. ArXiv:1404.2188
97 Deng L, Hinton G, Kingsbury B. New types of deep neural network learning for speech recognition and related applications: an overview. In: Proceedings of IEEE International Conference on Acoustics, Speech and Signal Processing, 2013
98 Graves A, Mohamed A, Hinton G. Speech recognition with deep recurrent neural networks. In: Proceedings of IEEE International Conference on Acoustics, Speech and Signal Processing, 2013
99 LeCun Y, Bottou L, Bengio Y, et al. Gradient-based learning applied to document recognition. Proc IEEE, 1998, 86: 2278–2324
100 Simonyan K, Zisserman A. Very deep convolutional networks for large-scale image recognition. 2014. ArXiv:1409.1556
101 Song L, Zhuo Y, Qian X H, et al. GraphR: accelerating graph processing using ReRAM. In: Proceedings of the 24th International Symposium on High-Performance Computer Architecture, 2018
102 Zheng L, Zhao J, Huang Y, et al. Spara: an energy-efficient ReRAM-based accelerator for sparse graph analytics applications. In: Proceedings of 2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2020. 696–707
103 Zhu X, Han W, Chen W. Gridgraph: large-scale graph processing on a single machine using 2-level hierarchical partitioning. In: Proceedings of 2015 USENIX Annual Technical Conference (USENIX ATC 15), 2015. 375–386
104 Zhang M, Zhuo Y, Wang C, et al. Graphp: reducing communication for PIM-based graph processing with efficient data partition. In: Proceedings of IEEE International Symposium on High Performance Computer Architecture (HPCA), 2018. 544–557
105 Ozdal M M, Yesil S, Kim T, et al. Energy efficient architecture for graph analytics accelerators. In: Proceedings of ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA), 2016. 166–177
106 Zhuo Y, Wang C, Zhang M, et al. GraphQ: scalable PIM-based graph processing. In: Proceedings of the 52nd International Symposium on Microarchitecture, 2019
107 Nag A, Balasubramonian R, Srikumar V, et al. Newton: gravitating towards the physical limits of crossbar acceleration. IEEE Micro, 2018, 38: 41–49
108 Choi S, Jang S, Moon J H, et al. A self-rectifying TaO_y/nanoporous TaO_x memristor synaptic array for learning and

energy-efficient neuromorphic systems. NPG Asia Mater, 2018, 10: 1097–1106
109 Li C, Belkin D, Li Y, et al. Efficient and self-adaptive in-situ learning in multilayer memristor neural networks. Nat Commun, 2018, 9: 2385
110 Liu Z, Tang J, Gao B, et al. Neural signal analysis with memristor arrays towards high-efficiency brain–machine interfaces. Nature Commun, 2020, 11: 1–9
111 Krestinskaya O, Choubey B, James A. Memristive GAN in analog. Sci Report, 2020, 10: 1–14
112 Song L, Wu Y, Qian X, et al. ReBNN: in-situ acceleration of binarized neural networks in ReRAM using complementary resistive cell. CCF Trans HPC, 2019, 1: 196–208
113 Bahou A A, Karunaratne G, Andri R, et al. XNORBIN: a 95 TOp/s/W hardware accelerator for binary convolutional neural networks. In: Proceedings of IEEE Symposium in Low-Power and High Speed Chips (COOL CHIPS), 2018
114 Conti F, Schiavone P D, Benini L. XNOR neural engine: a hardware accelerator IP for 21.6-fJ/op binary neural network inference. IEEE Trans Comput-Aided Des Integr Circ Syst, 2018, 37: 2940–2951
115 Jafari A, Hosseini M, Kulkarni A, et al. BiNMAC: binarized neural network manycore accelerator. In: Proceedings of Great Lakes Symposium on VLSI, 2018. 443–446
116 Andri R, Karunaratne G, Cavigelli L, et al. ChewBaccaNN: a flexible 223 TOPS/W BNN accelerator. 2020. arXiv:2005.07137
117 Kim D, Kung J, Chai S, et al. Neurocube: a programmable digital neuromorphic architecture with high-density 3D memory. In: Proceedings of ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA), 2016. 380–392
118 Zhuo Y, Chen J, Luo Q, et al. SympleGraph: distributed graph processing with precise loop-carried dependency guarantee. In: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation, 2020
119 Teixeira C H, Fonseca A J, Serafini M, et al. Arabesque: a system for distributed graph mining. In: Proceedings of the 25th Symposium on Operating Systems Principles, 2015. 425–440
120 Wang K, Zuo Z, Thorpe J, et al. RStream: marrying relational algebra with streaming for efficient graph mining on a single machine. In: Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation, 2018. 763–782
121 Mawhirter D, Wu B. Automine: harmonizing high-level abstraction and high performance for graph mining. In: Proceedings of the 27th ACM Symposium on Operating Systems Principles, 2019. 509–523
122 Jamshidi K, Mahadasa R, Vora K. Peregrine: a pattern-aware graph mining system. In: Proceedings of the 15th European Conference on Computer Systems, 2020. 1–16
123 Chen X, Dathathri R, Gill G, et al. Pangolin: an efficient and flexible graph mining system on CPU and GPU. 2019. ArXiv:1911.06969
124 Iyer A P, Liu Z, Jin X, et al. ASAP: fast, approximate graph pattern mining at scale. In: Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation, 2018. 745–761
125 Boyd S. Distributed optimization and statistical learning via the alternating direction method of multipliers. FNT Mach Learn, 2010, 3: 1–122
126 Ren A, Zhang T, Ye S, et al. ADMM-NN: an algorithm-hardware co-design framework of DNNs using alternating direction methods of multipliers. In: Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems, 2019. 925–938
127 Niu W, Ma X, Lin S, et al. PatDNN: achieving real-time DNN execution on mobile devices with pattern-based weight pruning. In: Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems, 2020
128 Song L, Mao J, Zhuo Y, et al. HyPar: towards hybrid parallelism for deep learning accelerator array. In: Proceedings of the 25th IEEE International Symposium on High-Performance Computer Architecture, 2019
129 Song L, Chen F, Zhuo Y, et al. AccPar: tensor partitioning for heterogeneous deep learning accelerators. In: Proceedings of the 26th IEEE International Symposium on High-Performance Computer Architecture, 2020
130 Harrison P, Valavanis A. Quantum Wells, Wires and Dots: Theoretical and Computational Physics of Semiconductor Nanostructures. Hoboken: John Wiley & Sons, 2016
131 Jensen F. Introduction to Computational Chemistry. Hoboken: John Wiley & Sons, 2017
132 Chapman T, Avery P, Collins P, et al. Accelerated mesh sampling for the hyper reduction of nonlinear computational models. Int J Numer Meth Engng, 2017, 109: 1623–1654
133 Nobile M S, Cazzaniga P, Tangherloni A, et al. Graphics processing units in bioinformatics, computational biology and systems biology. Brief Bioinform, 201, 18: 870–885
134 Arioli M, Demmel J W, Duff I S. Solving sparse linear systems with sparse backward error. SIAM J Matrix Anal Appl, 1989, 10: 165–190
135 Saad Y. Iterative methods for sparse linear systems. SIAM, 2003, 82
136 Fan Z, Qiu F, Kaufman A, et al. GPU cluster for high performance computing. In: Proceedings of the 2004 ACM/IEEE Conference on Supercomputing, 2004. 47
137 Song F, Tomov S, Dongarra J. Enabling and scaling matrix computations on heterogeneous multi-core and multi-GPU systems. In: Proceedings of the 26th ACM International Conference on Supercomputing, 2012. 365–376