Accelerator Architectures for
Deep Learning and Graph Processing

by

Linghao Song

Department of Electrical and Computer Engineering
Duke University

Date: ______________________________________
Approved:

__________________________________________
Yiran Chen, Advisor

__________________________________________
Hai Li, Co-Advisor

__________________________________________
Kenneth Brown

__________________________________________
Benjamin Lee

__________________________________________
Jun Yang

Dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Electrical and Computer Engineering in the Graduate School of Duke University

2020
ABSTRACT

Accelerator Architectures for Deep Learning and Graph Processing

by

Linghao Song

Department of Electrical and Computer Engineering
Duke University

Date: ______________________
Approved:

___________________________
Yiran Chen, Advisor

___________________________
Hai Li, Co-Advisor

___________________________
Kenneth Brown

___________________________
Benjamin Lee

___________________________
Jun Yang

An abstract of a dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Electrical and Computer Engineering in the Graduate School of Duke University

2020
Abstract

Deep learning and graph processing are two big-data applications and they are widely applied in many domains. The training of deep learning is essential for inference and has not yet been fully studied. With data forward, error backward, and gradient calculation, deep learning training is a more complicated process with higher computation and communication intensity. Distributing computations on multiple heterogeneous accelerators to achieve high throughput and balanced execution, however, remaining challenging. In this dissertation, I present AccPar, a principled and systematic method of determining the tensor partition for multiple heterogeneous accelerators for efficient training acceleration. Emerging resistive random access memory (ReRAM) is promising for processing in memory (PIM). For high-throughput training acceleration in ReRAM-based PIM accelerator, I present PipeLayer, an architecture for layer-wise pipelined parallelism. Graph processing is well-known for poor locality and high memory bandwidth demand. In conventional architectures, graph processing incurs a significant amount of data movements and energy consumption. I present GraphR, the first ReRAM-based graph processing accelerator which follows the principle of near-data processing and explores the opportunity of performing massive parallel analog operations with low hardware and energy cost. Sparse matrix-vector multiplication (SpMV), a subset of graph processing, is the key computation in iterative solvers for scientific computing. The efficiently accelerating floating-point processing in ReRAM remains a challenge. In this dissertation, I present ReFloat, a data format and a supporting accelerator architecture, for low-cost floating-point processing in ReRAM for scientific computing.
Acknowledgements

It is never easy to get a Ph.D. degree, but with the contribution and help from many people I finally did it. Here, I mention them.

Professor Yiran Chen, my advisor, and Professor Hai Li, my co-advisor, have invaluable advice and help to my research, professional development and life during my whole Ph.D. program. In the first three years of Ph.D. program, I struggled because none of my work was accepted by a conference. It was a really hard time and I even planned to quit and find an SDE job. But my advisors never gave up on me. They always encouraged me and believed in me. They gave me every opportunity to talk to or have a lunch with a scholar who visited our lab or was in collaboration, let me present at a senior student’s presentation, and made every effort to let me feel that I was not alone or forgotten. Professor Chen also gave me many skills suggestions for social communication, career and team management.

I would like to thank my committee members – Professor Kenneth Brown, Professor Benjamin Lee and Professor Jun Yang, and Professor Daniel Sorin in my qualify exam committee. Their time and efforts out of their busy schedules help me improve in the preliminary or/and final exams.

I would like to thank Professor Xuehai Qian (USC) for his insightful and valuable feedbacks on my research.

In my two summer research interns, my mentors Catherine Schuman (ORNL), Steven Young (ORNL) and Ang Li (PNNL) broadened my knowledge in computing systems.

I thank the efforts from my co-authors – Paul Bogdan, Nagadastagiri Challapalle, Fan Chen, Xiang Chen, Enes Eken, Yu Hua, Houxiang Ji, Bing Li, Xin Liu, Jiachen Mao, Vijaykrishnan Narayanan, Kent Nixon, Gabriel Perdue, Thomas Potok, Ximing Qiao, Sahithi Rampalli, Tianqi Tang, Chengning Wang, Yandan Wang, Yitu Wang, Wei Wen, Wujie Wen, Chunpeng Wu, Yuan Xie, Cong Xu, Huanrui Yang, Jianlei Yang, Youwei Zhuo.
My parents, Xuequan Song and Xiaoqin Dai, who always stand behind me.
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>An overview of my research.</td>
<td>4</td>
</tr>
<tr>
<td>2.1</td>
<td>Three basic tensor partitioning types.</td>
<td>20</td>
</tr>
<tr>
<td>2.2</td>
<td>Inter-layer communication patterns between three basic tensor partitioning</td>
<td>27</td>
</tr>
<tr>
<td></td>
<td>types.</td>
<td></td>
</tr>
<tr>
<td>2.3</td>
<td>Layer-wise partitioning is determined by dynamic programming to minimize</td>
<td>31</td>
</tr>
<tr>
<td></td>
<td>the computation cost and the communication cost.</td>
<td></td>
</tr>
<tr>
<td>2.4</td>
<td>Dynamic programming on multi-paths.</td>
<td>31</td>
</tr>
<tr>
<td>2.5</td>
<td>The speedup of data parallelism (DP), “one weird trick” (OWT) [1], HYPAR</td>
<td>36</td>
</tr>
<tr>
<td></td>
<td>[2] and AccPar in a heterogeneous accelerator array.</td>
<td></td>
</tr>
<tr>
<td>2.6</td>
<td>The speedup of data parallelism (DP), “one weird trick” (OWT) [1], HYPAR</td>
<td>38</td>
</tr>
<tr>
<td></td>
<td>[2] and AccPar in a homogeneous accelerator array.</td>
<td></td>
</tr>
<tr>
<td>2.7</td>
<td>The selected partitioning types for the weighted layers in Alexnet. The</td>
<td>39</td>
</tr>
<tr>
<td></td>
<td>number of hierarchies is 7 and the batch size is 128.</td>
<td></td>
</tr>
<tr>
<td>2.8</td>
<td>The speedup of data parallelism (DP), “one weird trick” (OWT) [1], HYPAR</td>
<td>40</td>
</tr>
<tr>
<td></td>
<td>[2] and AccPar under various partitioning hierarchies on Vgg19.</td>
<td></td>
</tr>
<tr>
<td>3.1</td>
<td>Convolutional neural network (CNN).</td>
<td>41</td>
</tr>
<tr>
<td>3.2</td>
<td>Data flows between two adjacent layers†.</td>
<td>44</td>
</tr>
<tr>
<td>3.3</td>
<td>Basics of ReRAM.</td>
<td>45</td>
</tr>
<tr>
<td>3.4</td>
<td>Data flows and dependencies in layer-wise pipeline.</td>
<td>47</td>
</tr>
<tr>
<td>3.5</td>
<td>A basic scheme for data input and kernel mapping.</td>
<td>50</td>
</tr>
<tr>
<td>3.6</td>
<td>A balanced scheme for data input and kernel mapping.</td>
<td>51</td>
</tr>
<tr>
<td>3.7</td>
<td>Layer-wise pipeline.</td>
<td>52</td>
</tr>
</tbody>
</table>
3.8 Latency of PIPELAYER architecture. ............................................. 53
3.9 Memory subarray design. .............................................................. 53
3.10 Overall architecture of PIPELAYER. ........................................... 55
3.11 Error backward for (a) activation function, (b) max pooling layer and (c) convolution layer. .......................................................... 58
3.12 Error backward for convolution layer by convolution of error of layer \( l \) and reordered kernels. .......................................................... 59
3.13 The computation of partial derivatives for kernels. .............................. 60
3.14 Tradeoff between resolution and accuracy. ........................................ 61
3.15 Weight partition. ........................................................................ 62
3.16 Speedups of networks in both training and inference. ............................ 65
3.17 Energy savings for PIPELAYER. ..................................................... 66
3.18 Speedups vs. parallelism granularity. ................................................ 68
3.19 Area vs. parallelism granularity. ...................................................... 68
4.1 Graph processing in vertex-centric programs. ...................................... 70
4.2 (a) Edge-centric processing and (b) dual sliding windows. ...................... 70
4.3 (a) Sparse matrix and its compressed representations in: (b) CSC, (c) CSR, and (d) COO. ................................................................. 73
4.4 (a) A directed graph and its representations in (b) adjacency matrix and (c) coordinate list. ................................................................. 74
4.5 Vertex Programming Model .............................................................. 75
4.6 GRAPHR Key insight: supporting graph processing with ReRAM crossbars. 76
4.7 GRAPHR accelerator architecture. .................................................... 78
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.8</td>
<td>Workflow of GRAPHR in out-of-core setting.</td>
<td>80</td>
</tr>
<tr>
<td>4.9</td>
<td>Controller operation.</td>
<td>80</td>
</tr>
<tr>
<td>4.10</td>
<td>Streaming-apply execution model.</td>
<td>81</td>
</tr>
<tr>
<td>4.11</td>
<td>Preprocessing edge list.</td>
<td>82</td>
</tr>
<tr>
<td>4.12</td>
<td>PageRank in vertex programming.</td>
<td>85</td>
</tr>
<tr>
<td>4.13</td>
<td>SSSP in vertex programming.</td>
<td>85</td>
</tr>
<tr>
<td>4.14</td>
<td>The configuration of sALU to perform (a) add in PageRank and (b) min in SSSP.</td>
<td>88</td>
</tr>
<tr>
<td>4.15</td>
<td>The processing of (a) PageRank and (b) SSSP in GRAPHR</td>
<td>89</td>
</tr>
<tr>
<td>4.16</td>
<td>GRAPHR speedup normalized to CPU platform.</td>
<td>94</td>
</tr>
<tr>
<td>4.17</td>
<td>GRAPHR energy saving normalized to CPU platform.</td>
<td>95</td>
</tr>
<tr>
<td>4.18</td>
<td>GRAPHR (a) performance and (b) energy saving compared to GPU platform.</td>
<td>96</td>
</tr>
<tr>
<td>4.19</td>
<td>GRAPHR (a) performance and (b) energy saving compared to PIM platform.</td>
<td>97</td>
</tr>
<tr>
<td>4.20</td>
<td>GRAPHR (a) performance and (b) energy saving v.s. dataset density.</td>
<td>97</td>
</tr>
<tr>
<td>5.1</td>
<td>Fixed-point (integer) MVM in ReRAM.</td>
<td>99</td>
</tr>
<tr>
<td>5.2</td>
<td>The bit layout of an (a) 8-bit signed integer and (b) a 64-bit double-precision floating-point number.</td>
<td>100</td>
</tr>
<tr>
<td>5.3</td>
<td>The cycle number and crossbar number under various bit length.</td>
<td>104</td>
</tr>
<tr>
<td>5.4</td>
<td>The conversion of index and value in floating-point format to REFLOAT format.</td>
<td>107</td>
</tr>
<tr>
<td>5.5</td>
<td>Comparison of a matrix block (a) in original full precision format and (b) in REFLOAT format.</td>
<td>109</td>
</tr>
</tbody>
</table>
5.6 The overall accelerator architecture for scientific computing in REFLOAT format. .......................................................... 113

5.7 The architecture for (a) a processing engine for floating-point MVM on a matrix block and (b) a crossbar cluster for fixed-point MVM. .......... 113

5.8 The row-major layout and block-major layout of a sparse matrix. .......... 116

5.9 The performance of GPU (in double and single precision), ESCMA and REFLOAT for CG solver. .................................................. 122

5.10 The performance of GPU (in double and single precision), ESCMA and REFLOAT for BiCGSTAB solver. ........................... 123

5.11 The convergence traces of CG solver of GPU (red line) and REFLOAT (blue line). .......................................................... 123

5.12 The convergence traces of BiCGSTAB solver of GPU (red line) and REFLOAT (blue line). .................................................. 124

5.13 Memory overhead of matrices in ESCMA and REFLOAT. ............... 125
List of Tables

2.1 Notations and descriptions. ........................................... 14
2.2 Landscape of DNN accelerators (accelerators highlighted in cyan are designed for training). ........................................ 16
2.3 Rotational Symmetry of the Three Tensor Multiplications. ............. 23
2.4 Intra-layer communication cost of the three basic tensor partitioning types. 26
2.5 Inter-layer communication cost between the three basic tensor partitioning types. .................................................. 26
2.6 The amount of floating point operations (FLOP) in the three multiplications. 29
2.7 The specifications of the accelerators. ........................................ 34
2.8 The caparison of flexibility of DP, OWT, HYPAR and ACCPAR. ....... 40
3.1 Operations in a cycle. ...................................................... 49
3.2 Cycles and cost in PIPELAYER architecture. .......................... 54
3.3 Hyper parameters of networks on MNIST. .................................. 64
3.4 Configurations of the GPU platform. ....................................... 64
3.5 Optimized parallelism granularity G configurations of each convolution layer in VGGs. ............................................. 67
4.1 Comparison of different architectures for graph processing. .............. 86
4.2 Property and operations of applications in GRAPHR. ................. 90
4.3 Graph datasets. ............................................................ 92
4.4 Specifications of the CPU platform. ....................................... 93
4.5 Specifications of the GPU platform. ....................................... 93
5.1 The iteration numbers for convergence under various exp(onent) and fra(ction) bit configurations for matrix crystm03. ............................... 107

5.2 List of symbols and descriptions. ........................................... 108

5.3 Platform Configuration. ..................................................... 119

5.4 Matrices in the evaluation. ..................................................... 121

5.5 The bit number for exponent and fraction of matrix block and vector segment in REFLOAT. ......................................................... 125

5.6 The absolute number of iterations to reaching convergence. .............. 126
Chapter 1

Introduction

Advances in deep learning (DL) have become the main drivers of revolutions in various commercial and enterprise applications, such as computer vision [3–5], social network [6–8], financial data analysis [9–11], healthcare [12–14] and scientific computing [15–17]. Due to the high demand of computing power in DL applications, we have recently witnessed a phenomenal trend in which the landscape of computing has shifted from general-purpose processors to domain-specific architectures [1, 2, 18–117]. By sacrificing some flexibility, such domain-specific accelerators are specialized for executing kernels of modern DL algorithms and therefore, are capable to deliver high performance with low power budget.

While the latest advances are pushing the envelope of DL acceleration for higher performance and energy efficiency, a recent study of chip specialization [118] has predicted an ultimate accelerator wall. More specifically, due to the constraints in the exploration of mapping computational problems (e.g., DL) onto hardware platforms with fixed hardware resources, the optimization space of chip specialization is limited by a theoretical roofline. Combining with recent slow CMOS technology scaling, the gains from specific accelerator designs will gradually diminish and the execution efficiency will eventually hit an upper-bound. On the other hand, the computational demands of emerging DL applications continue increasing in order to adapt to more complex models with deeper structures [119, 120] or more sophisticated learning methods [121, 122]. Efficient training acceleration of large-scale deep learning is a challenge.

It has been widely accepted that Moore’s Law is ending soon [123], which means that processor performance will no longer increase at the same pace as in recent decades. On the other side, the volume of data that computer systems process has skyrocketed over the last
decade. In conventional Von Neumann architecture where computation and data storage are separated, a large amount of data movements are also incurred due to the large number of layers and millions of weights. Such data movements quickly become a performance bottleneck due to limited memory bandwidth and more importantly, an energy bottleneck. A recent study [124] showed that data movements between CPUs and off-chip memory consumes two orders of magnitude more energy than a floating point operations. In an era of soon-ending Moore’s law and the increasing demand of deep learning, it is urgent to reduce the data movement and computing cost.

Processing-in-memory (PIM) is an efficient technique to reduce data movements in memory hierarchy. On the other hand, the emerging non-volatile memory, metal-oxide resistive random access memory (ReRAM) [125] has been considered as a promising candidate for future memory architecture due to its high density, fast read access and low leakage power. ReRAM-based PIM is a particularly appealing option because ReRAM provides both computation and storage capability. Such design incurs no increase in cost-per-bit implied by compute logic in traditional PIM based on DRAM. Recent works [88] demonstrated that ReRAM-based PIM offer great acceleration of the inference of Convolutional Neural Networks (CNNs) with low energy cost. However, how to accelerate training which is essential for deep learning inference with PIM architectures, remains challenged.

Besides deep learning, graph processing is another big-data applications. With the explosion of data collected from massive sources, graph processing received intensive interests due to the increasing needs to understand relationships. It has been applied in many important domains including cyber security [126], social media [127], PageRank citation ranking [128], natural language processing [129–131], system biology [132, 133], recommendation systems [134–136] and machine learning [137–139]. There are several ways to perform graph processing. The distributed systems [140–146] leverage the ample computing resources to process large graphs. However, they inherently suffer from synchro-
nization and fault tolerance overhead [147–149] and load imbalance [150]. Alternatively, disk-based single-machine graph processing systems [151–155] (a.k.a. out-of-core systems) can largely eliminate all the challenges of distributed frameworks. The key principle of such systems is to keep only a small portion of active graph data in memory and spill the remainder to disks. The third approach is the in-memory graph processing. The potential of in-memory data processing has been exemplified in a number of successful projects, including RAMCloud [156], Pregel [157], GraphLab [158], Oracle TimesTen [159], and SAP HANA [160].

It is well-known for the poor locality because of the random accesses in traversing the neighborhood vertices, and high memory bandwidth requirement, because the computations on data accesses from memory are typically simple. In addition, graph operations lead to memory bandwidth waste because they only use a small portion of a cache block. In conventional architecture, graph processing incurs significant amount of data movements and energy consumption.

Scientific computing, a subset of graph processing, is a collection of tools, techniques and theories required to solve science and engineering problems represented in mathematical systems [161]. Different from computer science that deals with quantities that are discrete, the underlying variables in scientific computing are continuous in nature, such as time, temperature, distance, density, etc. One of the most important aspects of scientific computing is obtaining solutions of the partial differential equations (PDEs) models. These solutions have a significant impact on the understanding of natural phenomena in science [162, 163] and the design and decision-making of engineered systems [164, 165]. Unfortunately, most problems in continuous mathematics cannot be solved accurately. In practice, they are converted to a linear system $Ax = b$ and then solved through an iterative solver that ultimately converges to a numerical solution [166, 167]. To obtain an accurate answer, intensive computing power [168, 169] is required to perform the sparse matrix-vector
multiplication (SpMV). In addition, a large memory footprint [170] is also essential for storing and retrieving the input and output data, as well as data generated during the course of the computation.

However, we are approaching the end of the scaling of Moore’s Law [171], and general-purpose platforms, such as CPUs and GPUs, will no longer benefit from the integration of cores [172]. Novel architecture paradigm is to be proposed to improve performance and efficiency on emerging application domains. Beyond conventional technology, emerging non-volatile memory such as resistive random access memory (ReRAM) is considered as a promising candidate for implementing processing-in-memory (PIM) accelerators [88, 89, 91, 92] that can provide orders of magnitude improvement of computing efficiency in the matrix-vector multiplication (MVM) operations. Note that most of these proposed ReRAM-based accelerators are limited to machine learning applications, and in most cases, machine learning applications can accept a low precision, e.g., less than 16-bit fixed-point [173]. Since the high-performance floating-point solver is a norm in scientific computing, it is an opportunity but also a challenge to leverage the power of parallel in-

**Figure 1.1:** An overview of my research.
situ processing in ReRAM to efficiently support floating-point SpMV and accelerate the mathematical model solving process in the field of science and engineering.

In this dissertation, I address the above mentioned challenges in deep learning and graph processing by architecture level optimizations. Related research was published on HPCA’20 [174], HPCA’19 [2], HPCA’18 [175] and HPCA’17 [92]. Another related work [176] is under review. Figure 1.1 shows a logical overview of my research.

1.1 Tensor Partitioning for Deep Learning Accelerators

To address the mismatch between the diminishing performance gains in hardware accelerators and the ever-growing computational demands, it is imperative to explore coarse-grained parallelism among multiple performance-bounded accelerators to support large-scale DL applications. In general, a deep neural network (DNN) model is a parametric function that takes a high-dimensional input and makes useful predictions (i.e., inference), such as a classification label. Model parameters, i.e., kernels or weights, are obtained through a large number of iterations in training process involving data forward, error backward and gradient calculation phases. The trained model can be used to perform inference function through only the data forward phase. Compared to inference, training is much more complicated and computational intensive. Hence, typically training is offloaded to high-end CPUs/GPUs and then the trained models are deployed to end/user devices. It is a natural need to have multi-accelerator architectures specialized for DNN training.

Given the high complexity of modern DNN models, finding the best distribution of computations on multiple accelerators is nontrivial. Moreover, due to the inherent performance difference of accelerators and the discrepancy of the communication bandwidth between them, ensuring high throughput and balanced execution is extremely challenging. The problem can be formulated as partitioning the model and data tensors among accelerators to enable parallel processing. There are two approaches: data parallelism, where each acceler-
ator replicates the model and processes different input data in parallel before applying the calculated gradients to update the model; and model parallelism, where each accelerator keeps part of the model and performs a part of computation based on the same input. The choice between the two parallelism configurations affects the overall performance since it incurs different communication patterns between the accelerators.

The current solutions to this problem are either purely empirical or incomplete — both lacking optimal guarantee. For example, for a given DNN, “One Weird Trick” (OWT) [1] empirically suggests to use data parallelism for convolutional (CONV) layers and model parallelism for fully-connected (FC) layers. HYPAR [2] proposes a principled approach to search for the optimized parallelism configuration to minimize data communication. Although it can achieve a much better result than OWT, HYPAR suffers from several limitations: 1) the search is based on an incomplete design space; 2) it can only handle DNN architectures with linear structure; 3) it lacks a cost model and uses only communication as the proxy for performance optimization; and most importantly 4) it assumes an homogeneous execution environment — the performance of each accelerator and the bandwidth between them are all the same. A truly optimal solution for this critical problem still does not exist yet.

The combination of model and data parallelism is explored in deep learning accelerator architectures [62,92] multi-GPU training systems [1,177–180]. Recursive methods [76,179] are proposed for tensor partitioning on multiple devices and dynamic programming methods [177, 179] are proposed for tensor partitioning layer-wisely. Inspired by those previous works [1, 76, 177–180], we present ACCPAR— a principled and systematic method of determining the tensor partition among heterogeneous accelerator arrays. Our solution is composed of several key innovations. First, we consider a complete tensor partition space in all three dimensions: batch size, input data size, and output data size. Hence, our solution is able to reveal previously unknown parallelism configuration. The completeness and
optimality of the searching algorithm is also guaranteed. Second, in order to better optimize performance, we propose a cost model considering both computation and communication cost of a heterogeneous execution environment, instead of using communication as the proxy for performance as in HYPAR. Third, ACCPAR offers flexible tensor partition ratio between the accelerators to match their unique computing power and network bandwidth. Finally, we propose a technique to handle the emerging multi-path patterns in modern DNNs such as ResNet [120]. ACCPAR significantly outperform the state-of-the-art solutions, offering the first complete solution for tensor partitioning on heterogeneous accelerator arrays.

We simulate ACCPAR on a heterogeneous accelerator array composed of both TPU-v2 and TPU-v3 accelerators for training of large-scale DNN models such as Alexnet [119], Vgg series [181] and Resnet series [120]. The average performance of “one weird trick” (OWT) [1], HYPAR [2] and ACCPAR, normalized to the baseline data parallelism on the heterogeneous accelerator array is $2.98\times$, $3.78\times$, $6.30\times$, respectively. For Vgg series, ACCPAR can achieve a speedup up to $16.14\times$, while the highest speedup of OWT and HYPAR are $8.24\times$ and $9.46\times$, respectively. For Resnet series, ACCPAR can achieve performance speedup from $1.92\times$ to $2.20\times$, while the ranges of speedup achieved by OWT and HYPAR are $1.22\times$ to $1.38\times$ and $1.03\times$ to $1.04\times$, respectively.

1.2 Layer-wise Pipelined Parallelism for Deep Learning Accelerators

Despite the recent progresses [88–90], the current schemes based on ReRAM lack important features to efficiently execute complete deep learning applications. First, they only focus on testing (inference) phase of CNN but do not support the more sophisticated and intensive training (learning) phase. Second, ISAAC [89] uses a very deep pipeline to improve system throughput. However, it is only beneficial when a large number of consecutive images can be fed into the architecture. This is not true in training phase as only a limited number of
consecutive image could be processed before weight updates. Third, the deep pipeline in ISAAC also introduces pipeline bubbles. Moreover, data organization and kernel mapping are not clearly addressed in PRIME [88].

To close the gaps of ReRAM-based acceleration for complete deep learning, this work proposes PIPEAYER, a ReRAM-based PIM accelerator that supports complete deep learning applications. Compared to previous works, we make the following contributions. (1) Accelerating both training and testing. Supporting training phase is more sophisticated and challenging because it involves weight updates and complex data dependencies. The previous schemes assume that the weight values are only written to ReRAM once at the start and never change. (2) The simple intra- and inter-layer pipeline designed for training. To ensure high throughput for CNNs of many layers with weight updates, PIPEAYER adopts a new pipelined architecture different from [89] so that data could continuously flow into the accelerator in consecutive cycles. It is essential to support pipelined training phase. Various data input and kernel mapping schemes could be exploited using our design to balance data processing parallelism and hardware cost (i.e. the number of replicated ReRAM arrays). (3) Spike-based data input and output. To eliminate the overhead of DACs and ADCs, PIPEAYER uses a spike-based scheme, instead of voltage-level based scheme for data input. Such design requires more cycles to inject data, however, the drawback is offset by the pipelined architecture for multiple layers. The data input scheme used in [89] is similar as our spike-based input, both eliminating DACs, but our Integration and Fire component eliminates ADCs while [89] keeps.

In the evaluation, we use ten networks. Six are popular large scale networks, AlexNet [119], and VGG-A, VGG-B, VGG-C, VGG-D, VGG-E [181], which are based on ImageNet [182]. Four networks based on MNIST [183] are built by ourselves. To evaluate PIPEAYER, we build a simulator based on NVSim [184]. And the baseline is a platform with the newest released GPU, GTX 1080. The experiment results show that, PIPEAYER achieves the
speedup of 42.45x compared with GPU platform on average. The average energy saving of PIPELAYER compared with GPU implementation is 7.17x.

1.3 Accelerator Architecture for Graph Processing

A graph can be naturally represented as an adjacency matrix and most graph algorithms can be implemented by some form of matrix-vector multiplications. However, due to the sparsity of graph, graph data are not stored in compressed sparse matrix representations, instead of matrix form. Graph processing based on sparse data representation involves: (1) bringing data for computation from memory based on compressed representation; (2) performing the computations on the loaded data. Due to the sparsity, the data accesses in (1) may be random and irregular. In essence, (2) performs simple computations that are part of the matrix-vector multiplications but only on non-zero operands. As a result, each computing core experiences alternative long random memory access latency and short computations. This leads to the well-known challenges in graph processing and other issues such as memory bandwidth waste [185].

The current graph processing accelerators mainly optimize the memory accesses. Specifically, Graphicionado [185] reduces memory access latency and improves throughput by replacing random accesses to conventional memory hierarchy with sequential accesses to scratchpad memory optimization and pipelining. Ozdal et al. [186] improves the performance and energy efficiency by latency tolerance and hardware supports for dependence tracking and consistency. TESSERACT [187] applies the principle of near-data processing by placing compute logics (e.g., in-order cores) close to memory to claim the high internal bandwidth of Hybrid Memory Cube (HMC) [188]. However, all architectures do little change on compute unit, — the simple computations are performed one at a time by instructions or specialized units.

To perform matrix-vector multiplications, two approaches exist that reflect two ends
of the spectrum: (1) the dense-matrix-based methods incur regular memory accesses and perform computations with every element in matrix/vector; (2) the sparse-matrix-based methods incur random memory accesses but only perform computations on non-zero operands. In this work, we adopt an approach that can be considered as the mid-point between these two ends that could potentially achieve better performance and energy efficiency. Specifically, we propose to perform sparse matrix-vector multiplications on data blocks of compressed representation. The benefit is two-fold. First, the computation and data movement ratio is increased. It means that the cost of bringing data could be naturally hidden by the larger amount of computations on a block of data. Second, inside this data block, computations could be performed in parallel. The downside is that certain hardware and energy will be wasted in performing useless multiplications with zero.

This approach could in principle be applied to the current GPUs or accelerators, but with the same amount of compute resources (e.g., SM in GPUs), it is unclear whether the gain would outweigh the inefficiency caused by the sparsity. Clearly, a key factor is the cost of compute logic, — with a low-cost mechanism to implement matrix-vector multiplications, the proposed approach is likely to be beneficial.

We demonstrate that the non-volatile memory, metal-oxide resistive random access memory (ReRAM) [125] could serve as the essential hardware building block to perform matrix-vector multiplications in graph processing. Recent works [88, 89, 92] demonstrate the promising applications of efficient in-situ matrix-vector multiplication of ReRAM on neural network acceleration. The analog computation is suitable for graph processing because: (1) The iterative algorithms could tolerate the imprecise values by nature; (2) Both probability calculation (e.g., PageRank and Collaborative Filtering) and typical graph algorithms involving integers (e.g., BFS/SSSP) are resilient to errors. Due to the low-cost and energy efficiency, matrix-based computation in ReRAM would not incur significant hardware waste due to sparsity. Such waste is only incurred inside the ReRAM crossbar
with moderate size (e.g., $8 \times 8$). Moreover, the sparsity is not completely lost, — if a small subgraph contains all zeros, it does not need to be processed. As a result, the architecture will mostly enjoy the benefits of more parallelism in computation and higher ratio between computation and data movements. Performing computation in ReRAM also enables near data processing with reduced data movements: the data do not need to go through the memory hierarchy like in the conventional architecture or some accelerators.

Applying ReRAM in graph processing poses a few challenges: (1) Data representation. Graph is stored in compressed format, not in adjacency matrix, to perform in-memory computation, data needs to be converted to matrix format. (2) Large graph. The real-world large graphs may not fit in memory. (3) Execution model. The order of subgraph processing needs to be carefully determined because it affects the hardware cost and correctness. (4) Algorithm mapping. It is important to map various graph algorithms to ReRAM with good parallelism.

We propose **GRAPHR**, a novel ReRAM-based accelerator for graph processing. It consists of two key components: memory ReRAM and graph engine (GE), which are both based on ReRAM but with different functionality. The memory ReRAM stores the graph data in compressed sparse representation. GE (ReRAM crossbars) perform the efficient matrix-vector multiplications on the sparse matrix representation. We propose a novel streaming-apply model and the corresponding preprocessing algorithm to ensure correct processing order. We propose two algorithm mapping patterns, parallel MAC and parallel add-op, to achieve good parallelism for different type of algorithms. **GRAPHR** can be used as a drop-in accelerator for out-of-core graph processing systems.

In the evaluation, we compare **GRAPHR** with a software framework [151] on a high-end CPU-based platform, GPU and PIM [187] implementations. The experiment results show that **GRAPHR** achieves a $16.01 \times$ (up to $132.67 \times$) speedup and a $33.82 \times$ energy saving on geometric mean compared to the CPU baseline. Compared to GPU, **GRAPHR** achieves
1.69× to 2.19× speedup and consumes 4.77× to 8.91× less energy. GRAPHR gains a speedup of 1.16× to 4.12×, and is 3.67× to 10.96× more energy efficiency compared to PIM-based architecture.

1.4 Low-Cost Floating-Point Processing in ReRAM

Previous work [189] is arguably the first to realize scientific computing using ReRAM. For accelerating the SpMV in an iterative solver, a whole matrix is partitioned into blocks and the input vector is partitioned into segments. Each matrix block is mapped to a cluster of ReRAM crossbars and the clusters process blocks of the matrix in parallel. For floating-point (64-bit) multiplication on a matrix block, the matrix block is expanded to 118 bits (53 bits for the fraction, 64 for the exponent padding and 1 bit for the sign) fixed-point format and mapped to 118 crossbars [189]. The mapping cost in [189] is high for ReRAM crossbars and leads to a low performance. Because the number of ReRAM crossbars on an accelerator is limited [89, 175, 189], the number of available clusters that can be used in parallel to process the matrix is also low because one cluster may include many crossbars. At the same time, a large number of crossbars in a cluster leads to a large number of cycles consumed by the floating-point multiplication for the matrix block with a vector segment.

In this work, we present REFLOAT, a data format and a supporting accelerator architecture, for low-cost floating-point processing in ReRAM for scientific computing. We study the effects of lower bits in both the exponent and the fraction for both matrix block and vector segment on (1) the number of required ReRAM crossbars to form a cluster to perform the floating-point multiplication on a matrix block, (2) the cycle number consumed to process the the floating-point multiplication on a matrix block and (3) the introduced accuracy loss. Then we present the conversion scheme from default double-precision floating format to REFLOAT format and the computing in REFLOAT format. The REFLOAT accelerator architecture is presented to support the low-cost floating-point processing in
REFLOAT format. In the evaluation part, we show REFLOAT performs $21.88 \times$ better than a GPU baseline and $4.30 \times$ faster than the state-of-the-art accelerator for scientific computing in ReRAM [189].
Chapter 2

Tensor Partitioning for Deep Learning Accelerators

2.1 Background and Motivation

2.1.1 DNN Training

<table>
<thead>
<tr>
<th>Table 2.1: Notations and descriptions.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Notation</strong></td>
</tr>
<tr>
<td>$F_l$</td>
</tr>
<tr>
<td>$E_{l+1}$</td>
</tr>
<tr>
<td>$W_l$</td>
</tr>
<tr>
<td>$\Delta W_l$</td>
</tr>
<tr>
<td>$B$</td>
</tr>
<tr>
<td>$D_{i,l}$</td>
</tr>
<tr>
<td>$D_{o,l}$</td>
</tr>
<tr>
<td>$c_i$</td>
</tr>
<tr>
<td>$p_{i,l}$</td>
</tr>
<tr>
<td>$b_i$</td>
</tr>
<tr>
<td>$\mathbb{A}(\cdot)$</td>
</tr>
<tr>
<td>$\mathbb{C}(\cdot)$</td>
</tr>
<tr>
<td>$\alpha, \beta$</td>
</tr>
</tbody>
</table>

DNN training involves three tensor computing phases at each layer: forward, backward and gradient. The notations and descriptions are listed in Table 2.1. In the forward phase, at layer $l$, the input feature map tensor ($F_l$) from a previous layer and the kernel/weight tensor ($W_l$) are multiplied in fully-connected (FC) layers or convolved in convolutional (CONV)
layers to generate the output feature map tensor, which is used as the input feature map
tensor for the next layer \( (F_{l+1}) \). Usually a non-linear activation \( f(\cdot) \) is performed on each
scalar of the feature map. Thus, the forward phase can be represented as \( F_{l+1} = f(F_l \otimes W_l) \),
where \( \otimes \) is either multiplication or convolution. In the backward phase, at layer \( l \), the error
tensor \( (E_l) \) is computed by \( E_l = (E_{l+1} \otimes W_l^T) \odot f'(F_l) \), where \( E_{l+1} \) is the error tensor from
layer \( l + 1 \), \( \odot \) is an element-wise multiplication, and \( f'(\cdot) \) is the derivative function of \( f(\cdot) \).
In the gradient phase, the gradient to the kernel/weight is computed by \( \Delta W_l = F_l^T \otimes E_{l+1} \).

The three tensor computation phases capture the common flow in many popular training
algorithms, such as Gradient Descent, Stochastic Gradient Descent, Mini-batch Gradient
Descent, Momentum \([190]\) and Adaptive Moment Estimation (Adam) \([191]\). For example,
Momentum method updates the parameter using \( v_t = \gamma \cdot v_{t-1} + \eta \cdot \nabla \theta J(\theta), \theta = \theta - v_t \),
where \( \theta \) is the parameter (weight), \( \nabla \theta J(\theta) \) is the gradient of a loss function \( J(\cdot) \) with
respect to \( \theta \), \( v \) is the velocity to record the historic gradient, \( \gamma \) is the momentum hyper
parameter and \( \eta \) is the learning rate, respectively.

## 2.1.2 Deep Learning Accelerator Architectures

Domain-specific computing architectures \([225–228]\) are considered as a promising solution
to accommodate the ever-growing intensive computing in various deep learning applications.
This view has also been confirmed by a flurry of DNN accelerators \([1, 2, 18–110]\) that
have emerged in recent years. Compared with general-purposed CPUs/GPUs, these custom
architectures achieved better performance and higher energy efficiency.

As summarized in Table 2.2, many designs include a vertical integration practice across
algorithm and hardware levels \([73–86]\) where DNN models are typically optimized prior to
being deployed for inference. Some designs investigate the dataflow (or data reuse pattern)
in DNN workloads \([53–61]\), among which Eyeriss \([54–56]\) is a representative design that
explores many data reuse opportunities existed in DNN execution. Processing-in-memory
Table 2.2: Landscape of DNN accelerators (accelerators highlighted in cyan are designed for training).

<table>
<thead>
<tr>
<th>Segment</th>
<th>DNN Accelerators &amp; Architectures</th>
</tr>
</thead>
<tbody>
<tr>
<td>Neuro Co-processor</td>
<td>SpiNNaker [18], Neuromorphic Acc. [19], TrueNorth [20–22], MT-spike [23], PT-spike [24]</td>
</tr>
<tr>
<td>DNN Co-processor</td>
<td>NPU [25], DianNao-family [26–29], Cambricon [30], Cambricon-x [31], TPU [32, 33], ScaleDeep [34], Stripes [35], Neural Cache [192, 193], Diffy [194]</td>
</tr>
<tr>
<td>FPGA</td>
<td>FPGA-DCNN [36], Embedded-FPGA-CNN [37], FPGA-Exploration [38], OpenCL-FPGA-CNN [39] [195], Caffeine [40], Deep Burning [41], FPGA-DPCNN [42], TABLA [43], DNNWEAVER [44], FP-DNN [45], FPGA-LSTM [46], ESE [47], FPGA-Dataflow [48], FPGA-BNN [49], FPGA-Utilization [50], FFT-CNN [51], VIBNN [52], iSwitch [196], ShortCut-Mining [197], FA3C [198], E-RNN [199]</td>
</tr>
<tr>
<td>Dataflow</td>
<td>Neuflow [53], Eyeriss [54–56], Flexflow [57], Fused-CNN [58], CNN-Paritition [59], <strong>GANAX</strong> [60], UCNN [61], TANGRAM [200], Sparse-Systolic [201], MAERI [202]</td>
</tr>
<tr>
<td>PIM</td>
<td>Neurocube [62], XNOR-POP [63], DRISA [64], 3DICT [65], NAND-NET [203], SCOPE [204], Promise [205]</td>
</tr>
<tr>
<td>Light Models</td>
<td>EIE [66], SC-DCNN [67], SCNN [68], Escher [69], LookNN [70], Bit-Pragmatic DNN [71], Bit Fusion [72], Cnvlutin [101], TIE [206], Laconic [207], ADM-NN [208], <strong>Gist</strong> [209]</td>
</tr>
<tr>
<td>Co-Design</td>
<td>DPS-CNN [73], Mineriva [74], MoDNN [75], MeDNN [76], AdaLearner [77], Stitch-X [78], <strong>PIM-DNN</strong> [79], Scalpel [80], CirCNN [81], CoSMIC [82], SnaPEA [83], OLAcce [84], Prediction-based DNN [85], PERMDNN [86], MnFast [210], DNN Computation Reuse [211], <strong>vDNN</strong> [212], Compressing-DMA-Engine [213], AxTrain [214], Eager pruning [215], Bit-Tactical [216], GENESYS [217]</td>
</tr>
<tr>
<td>Emerging Tech.</td>
<td>TETRIS [87], RENO [90], PRIME [88], ISAAC [89], Memristive Boltzmann Machine [91], <strong>PipeLayer</strong> [92], Atomlayer [93], ReCom [94], <strong>ReGAN</strong> [95], ReRAM ACC. [96], EMAT [97], ReRAM-BNN [98], ZARA [99], Snrram [100], Sparse ReRAM Engine [218], RedEye [219], Quantum-SCNN [220], FloatPIM [221], PUMA [222], FPISA [223]</td>
</tr>
<tr>
<td>Toolset/Framework</td>
<td><strong>Data Parallelism</strong> [108], OWT [1], NEUTRAMS [102], Perform-ML [103], Adaptive-Classifier [104], DNNBuilder [105], Group Scissor [106], FFT-CNN [107], HyPar [2], NNest [224]</td>
</tr>
</tbody>
</table>

(PIM) based designs are also proposed to reduce costly off-chip memory accesses [62–65]. Lightweight models are also introduced [66–72] to reduce computational effort. Many designs based on emerging memory technologies, such as resistive random access memory (ReRAM) technology [88–100, 229] with 3D stacking technology [62, 65, 87], are also proposed.
2.1.3 Motivation

Although domain-specific architectures have effectively addressed the challenges of the ending of Moore’s law [230], the recent study of chip specialization [118] has clearly demonstrated the diminishing specialization returns and the ultimate accelerator wall. In other words, it is unlikely to further achieve fine-grained optimizations on a single accelerator. To satisfy the computation and memory requirement for large DNN models and datasets that typically cannot be satisfied by a single accelerator, a natural solution is coarse-gained DNN execution on an accelerator array. On the other hand, as highlighted in cyan in Table 2.2, only a few of the existing DNN accelerators are designed for training. Among these designs, strict constraints are often applied to the models that can be supported. For example, [1, 2, 108] only considered homogeneous platforms, where the computation capability and the network bandwidth for each accelerator are identical. In reality, however, it is more important to explore solutions for an array of heterogeneous accelerators with various computation capacity and network bandwidth. For example, though a more powerful TPU-v3 was released, the early deployed TPU-v2 may not retire immediately considering the deployment cost and the need for supporting various acceleration workloads. They are in fact both available off-the-shelf [231]. It is important to optimize large-scale DNN training acceleration when both TPU versions are used simultaneously. To achieve high throughput and balanced execution, we need to efficiently distribute data and model tensors among accelerators with the awareness of heterogeneous computation capability and network bandwidth. A principled and systematic approach is needed to overcome the challenge of handling the complexity of DNN models and heterogeneous hardware execution environment.
2.2 Tensor Partitioning Space

Compared with DNN inference, DNN training is more complicated because of the three computation phases involved in training, i.e., forward, backward and gradient. The tensors and computations in the three phases are closely coupled together. We need first construct a complete set of the basic tensor partitioning types.

2.2.1 Problem Statement

We first consider FC layers and later show that the solution can be naturally extended to CONV layers. In FC layers, DNN training involves three tensor computing phases:

- **Forward:** \( F_{l+1} = f(F_l \times W_l) \),
- **Backward:** \( E_l = (E_{l+1} \times W_l^T) \odot f'(F_l) \),
- **Gradient:** \( \triangle W_l = F_l^T \times E_{l+1} \).

Using the notations in Table 2.1, the shape of the tensors in the above three phases are as below. Here we do not include the element-wise multiplications \( \odot \) in the space relations since they can be performed in place.

- **Forward:** \((B, D_{o,l}) \leftarrow (B, D_{i,l}) \times (D_{i,l}, D_{o,l})\),
- **Backward:** \((B, D_{i,l}) \leftarrow (B, D_{o,l}) \times (D_{o,l}, D_{i,l})\),
- **Gradient:** \((D_{i,l}, D_{o,l}) \leftarrow (D_{i,l}, B) \times (B, D_{o,l})\).

For illustration purpose, this section considers a simple case of an array with two accelerators. The problem is to exhaustively and systematically enumerate all possible partitions of the tensors involved in the three phases among the two accelerators, and understand the corresponding data communication and replication requirements. This is critical because the partition will determine the communication between the accelerators and affect overall training performance. We will also explain why the current solutions [1, 2] failed to provide a complete and comprehensive solution.
2.2.2 Partitioning in Three Dimensions

We note that the two matrices in each of the pairs \((F_l, E_l)\) and \((F_{l+1}, E_{l+1})\) have the same shape. We assume that \(F_l\) and \(E_l\) (also \(F_{l+1}\) and \(E_{l+1}\)) are partitioned in the same manner. This constraint is intuitive since otherwise additional communication will be unnecessarily incurred, contradicting our goal of minimizing communication between the accelerators.

We see only three dimensions appear in the three tensor computing phases: \(B\) (batch size), \(D_{o,l}\) (output data size of layer \(l\)), and \(D_{i,l}\) (input data size of layer \(l\)). Therefore, we can naturally focus on the partition in these three dimensions. For the partitions in one dimension, we assume that the same partition parameter is used for this dimension in every tensor to avoid additional communication.

Key observation: The dimensions are not independent. In fact, only one dimension can be “free” in a partition.

We explain observation using an example: consider the forward phase and the partition in \(B\) dimension. Since we will have only two partitions, for \((B, D_{i,l})\) \((F_l)\), the \(D_{i,l}\) dimension should not be partitioned. This also determines that \((D_{i,l}, D_{o,l})\) \((W_l)\) should not be partitioned in \(D_{i,l}\) dimension, otherwise the matrix multiplication cannot be performed. The only case left is the \(D_{o,l}\) dimension of \(W_l\). Suppose we partition that, the combination of multiplication of the local partitions in each accelerator does not lead to a complete result of \(F_l\) with shape \((B, D_{i,l})\). Specifically, depending on the partition, only the upper left and lower right sub-matrix, or upper right and lower left sub-matrix are computed. Therefore, \(D_{o,l}\) dimension of \(W_l\) cannot be partitioned. In fact, the whole \(W_l\) needs to be replicated on the two accelerators to compute the complete \(F_l\). The other scenarios can be considered similarly.

With the assumption that \(F_l\) and \(E_l\) (also \(F_{l+1}\) and \(E_{l+1}\)) use the same partition, and the fact that only one dimension is free in a partition, there are only three partition types. We discuss them one by one in the following.
The partitioning ratio for one accelerator is $\alpha$ and the partitioning ratio for the other accelerator is $\beta = 1 - \alpha$. Shadow tensors are assigned to one accelerator and non-shadow tensors are assigned to the other accelerator.

$\otimes$ denotes element-wise addition of two tensors.

**Figure 2.1**: Three basic tensor partitioning types.

**Type-I: Partitioning $B$ Dimension**

In Type-I, we partition the $B$ dimension in the three tensor computing phases, as shown in Figure 2.1(a).

In forward phase, $F_{l+1} = F_l \times W_l$. The element $F_{l+1}[b, qo]$ in $F_{l+1}$ can be computed as

$$F_{l+1}[b, qo] = \sum_{qi \in \{1, \ldots, D_{o,l}\}} F_l[b, qi] \times W_l[qi, qo], \quad (2.1)$$

where $b \in \{1, 2, \ldots, B\}, qo \in \{1, 2, \ldots, D_{o,l}\}$. We assume the ratio of computation to be assigned to one accelerator is $\alpha$, $0 \leq \alpha \leq 1$, and the ratio for the other is $\beta$, $\beta = 1 - \alpha$. The set $\{1, 2, \ldots, B\}$ is partitioned into two subsets $\{1, 2, \cdots, \alpha B\}$ and $\{\alpha B + 1, 2, \cdots, B\}$. Specifically, $F_l[1 : \alpha B :]$ is assigned to one accelerator and $F_l[\alpha B + 1 : B :]$ is assigned to the other. The two accelerators process disjoint subsets of the batch, and perform the computation indexed by the corresponding $b$ of the two subsets.

As discussed before, to ensure the validity of matrix multiplication and to get the complete results of $F_{l+1}$, $W_l$ is replicated in the two accelerators. After matrix multiplication, each accelerator produces a portion of results based on the same partition in $B$ dimen-
sion. Specifically, \( F_{l+1}[1 : \alpha B, :] \) is produced by one accelerator and \( F_{l+1}[\alpha B + 1 : B, :] \) is produced by the other.

In backward, an element \( E_l[b, q_i] \) in \( E_l \) can be computed as

\[
E_l[b, q_i] = \sum_{q_o \in \{1, \ldots, D_o, l\}} E_{l+1}[b, q_o] \times W_l^T[q_o, q_i].
\] (2.2)

Due to the constraint that \( F_l \) and \( E_l \) (also \( F_{l+1} \) and \( E_{l+1} \)) use the same partition, one accelerator keeps \( E_{l+1}[1 : \alpha B, :] \) and produces \( E_{l+1}[1 : \alpha B, :] \) with replicated \( W_l^T \). The other accelerator handles the other portion: \( E_l[\alpha B + 1 : B, :] \) and \( E_{l+1}[\alpha B + 1 : B, :] \).

The common pattern for forward and backward phases is that after replicating \( W_l \), accelerators can perform computation locally and produce disjoint parts of the result matrix, which can be combined to obtain the complete result. However, this pattern does not exist in gradient phase as the two accelerators are not able to complete the computation individually.

In gradient phase, an element \( \nabla W_l[q_i, q_o] \) in \( \nabla W_l \) can be computed as

\[
\nabla W_l[q_i, q_o] = \sum_{b \in \{1, \ldots, B\}} F_l^T[q_i, b] \times E_{l+1}[b, q_o].
\] (2.3)

Based on the same partition of \( B \) dimension in \( F_l^T \) and \( E_{l+1} \), the two accelerators can perform local matrix multiplications. Each of them will produce the result matrix of shape \((D_{l, l}, D_{o, l})\), the same as \( \nabla W_l \). To get the final results in Equation (2.3), element-wise additions need to be performed to combine the partial results:

\[
\nabla W_l[q_i, q_o] = \sum_{b \in \{1, \ldots, \alpha B\}} F_l^T[q_i, b] \times E_{l+1}[b, q_o] \\
+ \sum_{b \in \{\alpha B + 1, \ldots, B\}} F_l^T[q_i, b] \times E_{l+1}[b, q_o].
\] (2.4)

The computation pattern in gradient phase implies that communication is needed to obtain the final results of \( \nabla W_l \), because one of the accelerators needs to perform the partial sum. We call it as intra-layer communication. We can see that such communication happens at different phases for different types of partition.

21
Type-II: Partitioning $D_{i,l}$ Dimension

The partition in $D_{i,l}$ dimension is shown in Figure 2.1(b). To perform matrix multiplication, $D_{i,l}$ dimension of $F_l$ is partitioned in the same way. Based on this, in forward phase, each accelerator will compute a result matrix of shape $(B, D_{o,l})$. Similar to the case of $\Delta W_l$ in Type-I, computing the complete $F_{l+1}$ requires element-wise addition in this partitioning:

$$F_{l+1}[b, q_o] = \sum_{q_i \in \{1, \ldots, \alpha D_{i,l}\}} F_l[b, q_i] \times W_l[q_i, q_o] + \sum_{q_i \in \{\alpha D_{i,l} + 1, \ldots, D_{i,l}\}} F_l[b, q_i] \times W_l[q_i, q_o].$$

(2.5)

Since $F_{l+1}$ and $E_{l+1}$ follow the same partition, in backward phase, $E_{l+1}$ is replicated in the two accelerators. This allows each of the accelerators produces a disjoint part of result of $E_l$. The partition and replication are similar to that used in gradient phase. A key difference between Type-I and Type-II is that the intra-layer communication incurs at forward phase, instead of gradient phase.

Type-III: Partitioning $D_{o,l}$ Dimension

The partitioning $D_{o,l}$ dimension is shown in Figure 2.1(c). $F_l$ needs to be replicated to compute complete $F_{l+1}$ in forward phase. It is the case overlooked by all previous solutions. Essentially, it means that the input feature maps of the same batch are replicated into the two accelerators, instead of partitioning $B$. It may sound not intuitive since we want to have accelerators process the same data. However, we show that it is an important partition in the design space that presents the same trade-off in terms of communication just as in Type-I and Type-II.

Similar to gradient phase of Type-I and forward phase of Type-II, the backward phase
of Type-III requires element-wise addition of partial results:

\[
E_l[b, qi] = \sum_{qo \in \{1, \ldots, \alpha D_o, l\}} E_{l+1}[b, qo] \times W^T_l[qo, qi]
\]

\[
+ \sum_{qo \in \{\alpha D_o, l+1, \ldots, D_o, l\}} E_{l+1}[b, qo] \times W^T_l[qo, qi].
\]

(2.6)

2.2.3 Extension to CONV

In the previous sessions, we used matrix-matrix multiplication in FC to illustrate the three types of partitions, which can be conceptually visualized in Figure 2.1. For CONV, the three partitioning types are still valid. However, \(F_l[b, qi], F_{l+1}[b, qo], E_l[b, qi]\) and \(E_{l+1}[b, qo]\) are no longer scalars but are 2-dimensional matrices. Therefore, \(F_l, F_{l+1}, E_l\) and \(E_{l+1}\) are 4-dimensional tensors, i.e., \((\text{batch, channel}) \times (\text{width, height})\). Similarly, \(\Delta W_l[qi, qo]\) and \(W_l[qi, qo]\) are also 2-dimensional matrices rather than scalars. Thus, \(\Delta W_l\) and \(W_l\) are 4-dimensional tensors, i.e., \((\text{input channel, output channel}) \times (\text{kernel width, kernel height})\). The multiplication \((\times)\) in Equation (2.1), (2.2), (2.3), (2.4), (2.5), (2.6) then become convolution \((\otimes)\). The additional dimensions (4D vs. 2D) and more complex operations \((\times \text{ vs. } \otimes)\) only imply different amount of computation but not affect the partition types based on existing dimensions \((B, D_{i,l}, \text{and } D_{o,l})\).

2.2.4 Completeness

Table 2.3: Rotational Symmetry of the Three Tensor Multiplications.

<table>
<thead>
<tr>
<th>Multiplication</th>
<th>L Shape</th>
<th>R Shape</th>
<th>Partition Dim</th>
<th>Psum Shape</th>
<th>Basic Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>(F_{l+1} = F_l \times W_l)</td>
<td>((B, D_{i,l}))</td>
<td>((B, D_{i,l}), (D_{i,l}, D_{o,l}))</td>
<td>(D_{i,l})</td>
<td>((B, D_{o,l}))</td>
<td>Type-II</td>
</tr>
<tr>
<td>(E_l = E_{l+1} \times W_l)</td>
<td>((B, D_{i,l}))</td>
<td>((B, D_{o,l}), (D_{i,l}, D_{o,l}))</td>
<td>(D_{o,l})</td>
<td>((B, D_{i,l}))</td>
<td>Type-III</td>
</tr>
<tr>
<td>(\Delta W_l = F_l^\top \times E_{l+1})</td>
<td>((D_{i,l}, D_{o,l}))</td>
<td>((B, D_{i,l}), (B, D_{o,l}))</td>
<td>(B)</td>
<td>((D_{i,l}, D_{o,l}))</td>
<td>Type-I</td>
</tr>
</tbody>
</table>

In the three phases, only three dimensions appear and we have shown that only one dimension can be partitioned at a time. Thus, the three types we derived constitute the
complete partition space. Table 2.3 summarized the key features of the partitions. The LHS Shape and RHS Shape respectively indicate the shapes of the metrics on the left-hand and right-hand side of the equation for each phase. The Psum Shape is the shape of the matrices containing partial results in two accelerators that need to be combined using element-wise additions. It happens when the matrix appears on the LHS. This is also the shape of the matrix that needs to be replicated if it appears on the RHS. From Table 2.3, we can observe a rotational symmetry on each column.

### 2.2.5 Problems of “One Weird Trick” & HyPar

Two solutions were recently proposed to address the same problem that is addressed by this work, — communication and parallelism between accelerators. However, neither of these two solutions is complete.

Krizhevsky [1] proposed “one weird trick” (OWT) to configure CONV layers with data parallelism and FC layers with model parallelism to get a higher performance. It is certainly better than just using data parallelism for all layers, however it does not provide any insight on why this trick works and whether it is the best we can do. Therefore, this solution is fundamentally empirical.

HyPar [2] is a more recent and principled approach to optimize the parallelism configurations also by partitioning the layers based on the intra-layer and inter-layer communication. However, it only considers the same two basic partitions in OWT, — data parallelism and model parallelism. In fact, they correspond to Type-I and Type-II in Figure 2.1, respectively. Therefore, the parallelism setting in HyPar is not complete. Even if it is based on a more systematic approach to explore the partition space, it cannot find the optimal solution based on incomplete basic partition types. Specifically, HyPar will miss one intra-layer communication pattern (Type-III) and five inter-layer communication patterns (see more details in Section 2.3.1). Moreover, HyPar always partitions the tensors equally, so it cannot capture
the performance heterogeneity among accelerators.

### 2.3 Cost Model

To search the optimal partition of layers, we propose a cost model for multiple accelerators. We consider the computation by individual accelerator and the communication between accelerators as two major affecting DNN training performance. Compared to HYPAR [2], which uses communication cost as the proxy for performance, the cost model of ACCPAR takes both communication cost $E_{cm}$ and computation cost $E_{cp}$ into consideration. The optimization target is to minimize overall cost.

#### 2.3.1 Communication Cost Model

Assuming the network bandwidth for accelerator $i$ is $b_i$, and $T$ is the accessed tensor needs to be transferred from one to the other, we define the communication cost $E_{cm}$ for the tensor transfer as

$$E_{cm} = \frac{A(T)}{b_i}. \quad (2.7)$$

The tensor size $A(T)$ is defined as the product of the lengths of all dimensions. For example, the size of a 4-by-5 matrix is 20, and the size of a kernel whose input channel is 16, kernel window width is 3, kernel window length is 3 and output channel is 32, is $4,608 = 16 \times 3 \times 3 \times 32$. Next, we will determine what the remotely-accessed tensor $T$ is.

### Intra-layer Communication Cost

As discussed in Section 2.2, for each of the three basic tensor partitioning types, there is one and only one computation phase requires remote accessing.

In Type-I, the gradient phase requires remote accessing (Equation (2.4)). For the accelerator whose partitioning ratio is $\alpha$, for each $b \in \{1, \cdots, \alpha B\}$, the intermediate tensor
Table 2.4: Intra-layer communication cost of the three basic tensor partitioning types.

<table>
<thead>
<tr>
<th>Basic Type</th>
<th>Intra-layer Communication Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type-I</td>
<td>$\frac{\Delta(W_l)}{b_i}$</td>
</tr>
<tr>
<td>Type-II</td>
<td>$\frac{\Delta(F_{l+1})}{b_i}$</td>
</tr>
<tr>
<td>Type-III</td>
<td>$\frac{\Delta(E_l)}{b_i}$</td>
</tr>
</tbody>
</table>

Note that intra-layer communication cost is not dependable on the partitioning ratio $\alpha$ because intermediate results are accumulated locally and partial sum tensors are accessed remotely.

size is $\Delta(F_l^T[:,b] \times E_{l+1}[b,:]) = D_{l,l} \cdot D_{o,l} = \Delta(\Delta W_l) = \Delta(W_l)$. Those intermediate tensors ($\forall b \in \{1,\cdots,\alpha B\}$) are accumulated locally ($\sum_{q_i \in \{1,\cdots,\alpha D_{l,l}\}} (\cdot)$) by the accelerator $i$ to reduce remote accessing by the other accelerator $j$. Also, the accelerator $j$ performs local accumulation $\sum_{q_i \in \{\alpha D_{l,l}+1,\cdots,D_{l,l}\}} (\cdot)$. Thus, the size of the tensor remotely accessed by accelerator $i$ from accelerator $j$ is $\Delta(W_l)$ rather than $(1 - \alpha) \cdot B \cdot \Delta(W_l)$. With Equation (2.7), we get the intra-layer communication cost for accelerator $i$ to remotely access the partial sum tensor in accelerator $j$ is $\frac{\Delta(W_l)}{b_i}$.

For Type-II and Type-III, readers can follow the similar idea to get the intra-layer communication cost for the two basic tensor partitioning types. We list the intra-layer communication cost of the three basic tensor partitioning types in Table 2.4.

### Inter-layer Communication Cost

Table 2.5: Inter-layer communication cost between the three basic tensor partitioning types.

<table>
<thead>
<tr>
<th>Layer $l$</th>
<th>Layer $l + 1$</th>
<th>Type-I</th>
<th>Type-II</th>
<th>Type-III</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type-I</td>
<td>$0$</td>
<td>$\frac{\alpha \beta \Delta(F_{l+1})}{b_i}$</td>
<td>$\frac{\beta \Delta(F_{l+1})}{b_i}$</td>
<td></td>
</tr>
<tr>
<td>Type-II</td>
<td>$\frac{\beta \Delta(E_{l+1})}{b_i}$</td>
<td>$\frac{\beta \Delta(E_{l+1})}{b_i}$</td>
<td>$0$</td>
<td></td>
</tr>
<tr>
<td>Type-III</td>
<td>$\frac{\alpha \beta \Delta(F_{l+1}) + \alpha \beta \Delta(E_{l+1})}{b_i}$</td>
<td>$0$</td>
<td>$\frac{\beta \Delta(F_{l+1})}{b_i}$</td>
<td></td>
</tr>
</tbody>
</table>

Since each layer is assigned a basic tensor partitioning type, when switching content from one layer to the next layer, an accelerator may require remote accessing. That is the
Shadow tensors are held by one accelerator (whose partitioning ratio is $\alpha$) and non-shadow tensors are held by the other accelerator (whose partitioning ratio is $\beta$).

**Figure 2.2**: Inter-layer communication patterns between three basic tensor partitioning types.

Inter-layer communication. There are two tensor conversions may require remote accessing, i.e., (1) the conversion of the output feature map tensor $F_{l+1}$ in layer $l$ to the input feature map tensor $F_{l+1}$ in layer $l+1$ and (2) the conversion of the output error tensor $E_{l+1}$ in layer $l+1$ to the input error tensor $E_{l+1}$ in layer $l$. As we have three basic tensor partitioning types, there are nine inter-layer communication patterns between the basic types, as shown in Figure 2.2. Tensors in layer $l$ are colored in green and Tensors in layer $l+1$ are colored in blue.
(a) Type-I to Type-I, (f) Type-II to Type-III and (h) Type-III to Type-II.

In the tensor conversion of the three patterns, since the (green) tensors in layer $l$ and the (blue) tensors in layer $l+1$ has the same partitioning, there is no conversion, and the inter-layer communication cost is 0.

(c) Type-I to Type-III, (d) Type-II to Type-I, (e) Type-II to Type-II and (i) Type-III to Type-III.

We take Figure 2.2(c) as an example. In the tensor conversion from Type-I to Type-III, in the forward phase, the accelerator $i$ (whose partitioning ratio is $\alpha$) holds green shadow tensor $F_{l+1}$ in layer $l$, but in the next layer $l+1$, the accelerator need the whole blue shadow tensor $F_{l+1}$. The difference is the black part, and the black tensor incurs remote accessing to the other accelerator $j$ (whose partitioning ratio is $\beta$). Thus the inter-layer communication amount is $(\beta B) \times D_{o,l} = \beta \Lambda(F_{l+1})$, and the inter-layer communication cost by accelerator $i$ to remotely access the black tensor in accelerator $j$ is $\frac{\beta \Lambda(F_{l+1})}{b_j}$. Reversely, the inter-layer communication cost for the accelerator $j$ is $\frac{\alpha \Lambda(F_{l+1})}{b_j}$ in this case. Note that the inter-layer communication cost for (c) Type-I to Type-III, (d) Type-II to Type-I, (e) Type-II to Type-II and (i) Type-III to Type-III are the same, but the shapes of conversion tensors are not the same.

(b) Type-I to Type-II and (g) Type-III to Type-I.

We take Figure 2.2(b) as an example. In the tensor conversion from Type-I to Type-II, in the forward phase, the accelerator $i$ (whose partitioning ratio is $\alpha$) holds green shadow tensor $F_{l+1}$ $(\alpha B, D_{o,l})$ in layer $l$, but in the next layer $l+1$, the accelerator need the whole blue shadow tensor $F_{l+1}$ $(B, \alpha D_{o,l})$. The difference is the black part, and again the black tensor incurs remote accessing to the other accelerator $j$ (whose partitioning ratio is $\beta$). Thus the inter-layer communication amount is $(\beta B) \times \alpha D_{o,l} = \alpha \beta \Lambda(F_{l+1})$, and
the inter-layer communication cost by accelerator $i$ to remotely access the black tensor in 
accelerator $j$ is $\frac{\alpha \beta \hat{A}(F_{l+1})}{b_j}$. Reversely, the inter-layer communication cost for the 
accelerator $j$ is $(1-\alpha)(1-\beta)\hat{A}(F_{l+1}) \cdot \frac{\beta \alpha \hat{A}(F_{l+1})}{b_j}$ in this case. Note that the inter-layer communication 
cost for (b) Type-I to Type-II and (g) Type-III to Type-I are the same, but the shapes of 
conversion tensors are not the same.

We list the inter-layer communication cost for the nine patterns of the tensor conversion 
between the three basic partitioning types in Table 2.5.

### 2.3.2 Computation Cost Model

We assume the tensor computation density of an accelerator $i$ is $c_i$ and the amount of floating 
point operations to perform the multiplication of two tensors $T_1 \times T_2$ 
is $C(T_1 \times T_2)$. For an accelerator with a partitioning ratio $\alpha$, the effective amount floating 
point operations performed is $\alpha \cdot C(T_1 \times T_2)$. We can define the computation cost $E_{cp}$ for 
an accelerator $i$ to perform the computation as

$$
E_{cp} = \frac{\alpha \cdot C(T_1 \times T_2)}{c_i}.
$$

(2.8)

**Table 2.6:** The amount of floating point operations (FLOP) in the three multiplications.

<table>
<thead>
<tr>
<th>Multiplication</th>
<th># FLOP</th>
</tr>
</thead>
<tbody>
<tr>
<td>$F_{l+1} = F_l \times W_l$</td>
<td>$\hat{A}(F_{l+1}) \cdot (D_{i,l} + D_{i,l} - 1)$</td>
</tr>
<tr>
<td>$E_l = E_{l+1} \times W_l$</td>
<td>$\hat{A}(E_l) \cdot (D_{o,l} + D_{o,l} - 1)$</td>
</tr>
<tr>
<td>$\triangle W_l = F_l \times E_{l+1}$</td>
<td>$\hat{A}(W_l) \cdot (B + B - 1)$</td>
</tr>
</tbody>
</table>

The most important step to get the computation cost is to calculate the number of floating 
point operations (FLOP) of a tensor multiplication. In the forward phase, to get the output 
tensor $F_{l+1}$, the number of FLOP is $(B \cdot D_{o,l}) \cdot (D_{i,l} + D_{i,l} - 1) = \hat{A}(F_{l+1}) \cdot (D_{i,l} + D_{i,l} - 1)$.
The numbers of FLOP for the three multiplications are listed in Table 2.6.
2.3.3 Discussion on Convolutions

We can easily expand the communication cost and computation cost from fully-connected layers to convolutional layers. In convolutions, \( F_l, F_{l+1}, E_l \) and \( E_{l+1} \) are 4-dimensional tensors, i.e., (batch, channel, height, width). We can view the four dimensional tensors as three dimensional tensors, but the third and fourth dimension is a meta dimension, i.e., (batch, channel, [height, width]). The kernel \( W_l \) are also 4-dimensional tensors, i.e., (input channel, output channel, kernel height, kernel width), and we can also view it as a three dimensional tensor, and the second dimension is a meta dimension, i.e., (input channel, [kernel height, kernel width], output channel). Thus, the communications costs listed in Table 2.4 and 2.5 keep the same formats.

In a matrix-matrix multiplication \( M_C = M_A \times M_B \), assume the shape of \( M_C, M_A, M_B \) is \((M_C; N_C), (M_C, P), (P, N_C)\) respectively. The idea to to calculate the number of floating points performed is to multiply the number of output elements and the the number of floating points for each element. In the matrix-matrix multiplication, the number of output elements is \( M_C \times N_C = A(M_C) \). For each output element, the number of multiplications performed is \( P \) and the number of additions performed is \( P - 1 \). So the total number of FLOP is \( A(M_C) \cdot (P + P - 1) \). To find the number of FLOP for a convolutional layers, we need only to find the number of FLOP for the convolution for one element in the output tensor because the number of elements of a tensor \( T_{out} \) is always \( A(T_{out}) \) no matter what the dimension it is. The number of multiplications performed is (input channel) \times (kernel height) \times (kernel width) and number of additions performed is ((input channel) \times (kernel height) \times (kernel width) - 1). Note that (input channel) is \( D_{i,l}, D_{o,l} \) or \( B \) in the three multiplications respectively, and (kernel height) \times (kernel width) is actually the 2D feature map or kernel size, i.e., the size of the feature map or kernel except the input and output channel. So for convolutional layers, the number of floating point operations is the entries in Table 2.6 multiply the 2D feature map or kernel size.
2.4 Partitioning Algorithm

In this section, we explain the ACCPAR partitioning method. Like recent work HyPar [2], we determine the partitioning for each layer in a DNN model by a layer-wise dynamic partitioning scheme. However, ACCPAR is much more general for three reasons: 1) the algorithm considers the complete search space discussed in Section 2.4; 2) it can be parameterized with arbitrary partitioning ratio based on heterogeneous compute, communication cost and effective bandwidth between accelerator groups; 3) it can handle multiple paths in DNNs. As a result, we will see in Section 2.5 that ACCPAR achieves considerable speedups over HYPAR.

2.4.1 Layer-wise Partitioning

![Figure 2.3](image-url)

**Figure 2.3:** Layer-wise partitioning is determined by dynamic programming to minimize the computation cost and the communication cost.

![Figure 2.4](image-url)

(a) Layer $L_i$ and $L_{i+1}$ are connected by path P1 and path P2. (b) Partitioning for ($L_i, t = \text{Type-I}$) to ($L_{i+1}, t = \text{Type-I}$), (c) Partitioning for ($L_i, t = \text{Type-II}$) to ($L_{i+1}, t = \text{Type-I}$), and (d) Partitioning for ($L_i, t = \text{Type-III}$) to ($L_{i+1}, t = \text{Type-I}$).

**Figure 2.4:** Dynamic programming on multi-paths.
To find the best partitioning for each layer in a DNN to minimize communication and improve performance, an intuitive way is to enumerate all possible configurations by brute force. Unfortunately, it will result in a $O(3^N)$ complexity for a DNN with $N$ layers — not a practical solution. Following the dynamic programming approaches [2, 177, 179], we reduce search complexity to $O(N)$ by dynamic programming.

Figure 2.3 illustrates the layer-wise partitioning procedure. For each layer, we determine the minimum cost based on the three basic partitioning types from the first layer till the current layer. We denote the accumulative cost up to layer $L_i$ when it is in state $(L_i, t)$ as $c(L_i, t)$ — layer $L_i$ chooses a basic partitioning type $t \in \mathcal{T} = \{\text{Type-I, Type-II, Type-III}\}$. Based on the cost model in Section 2.3, the accumulative cost given partition choice $t$ of the current layer $L_{i+1}$ ($c(L_{i+1}, t)$) can be calculated inductively with the cost of the previous layer $L_i$ ($c(L_i, tt)$):

$$c(L_{i+1}, t) = \min_{t \in \mathcal{T}} \{c(L_i, tt) + E_{cp}(t) + E_{cm}(tt, t)\}.$$

Here, $E_{cp}(t)$ is the computation cost for the current layer $L_{i+1}$ for a type $t$, and $E_{cm}(tt, t)$ is the sum of the intra-layer communication cost for a type $t$ and the inter-layer communication cost when transition from state $(L_i, tt)$ to state $(L_{i+1}, t)$. For each basic partitioning type, during the algorithm execution we need to record the path to a previous layer for backtracking after going through all layers, shown as the black arrows in Figure 2.3. In this manner, after we compute the accumulative cost of the last layer, we have obtained the cost state of the last layer and all previous layers with backtracking. The algorithm starts by initializing $c(L_0, t)$ for the three basic partitioning types in layer $L_0$ to 0.

We employ a hierarchical (recursive) partition for multiple hierarchies similar to [2, 76, 179]. The idea is to apply the layer-wise partitioning recursively on a partitioned hierarchy to partition on an accelerator array.
### 2.4.2 Handling Multiple Paths

Different from HYPAR, ACCPAR is able to determine the partition for the multi-path typologies that are common in Resnet [120]. Figure 2.4 shows an example where there are two paths between layer $L_i$ and $L_{i+1}$. Path P1 consists of one weighted layer and Path P2 consists of two weighted layers. The key idea for multi-path partitioning is to (1) enumerate the partition state of layer $L_{i+1}$ ($L_{i+1}, t \in \mathcal{T}$), (2) enumerate the partition state of layer $L_i$ ($L_i, tt \in \mathcal{T}$), (3) perform individual layer-wise partitioning for each path between the two states ($L_i, tt$) and ($L_{i+1}, t$) for each combination, (4) determine the lowest cost for the state ($L_{i+1}, t$).

We need to determine the lowest cost for each state, i.e., enumerate the three colored circles in Figure 2.4 in Layer $L_{i+1}$. For example, ($L_{i+1}, t = \text{Type-I}$) is one of the three possible states shown as the yellow circle at Layer $L_{i+1}$ in Figure 2.4. We then enumerate the three colored circles at Layer $L_i$. Figure 2.4(b) starts from ($L_i, t = \text{Type-I}$), we search the three paths in P1 and the three paths in P2 between the two states ($L_i, tt$) and ($L_{i+1}, t$) for each combination, (4) determine the lowest-cost path for each combination. Similarly, we compute the lowest-cost path from ($L_i, t = \text{Type-II}$) to ($L_{i+1}, t = \text{Type-I}$) as shown in Figure 2.4(c), and the lowest-cost path from ($L_i, t = \text{Type-III}$) to ($L_{i+1}, t = \text{Type-I}$) as shown in Figure 2.4(d). With the three paths, i.e., the lowest-cost paths (1) from ($L_i, t = \text{Type-I}$) to ($L_{i+1}, t = \text{Type-I}$), (2) from ($L_i, t = \text{Type-II}$) to ($L_{i+1}, t = \text{Type-I}$) and (3) from ($L_i, t = \text{Type-III}$) to ($L_{i+1}, t = \text{Type-I}$), we can finally determine the lowest cost to reach state ($L_{i+1}, t = \text{Type-I}$) and record the lowest-cost path from Layer $L_i$ to state ($L_{i+1}, t = \text{Type-I}$). Following the similar procedure, we can determine the lowest-cost path to state ($L_{i+1}, t = \text{Type-II}$) and state ($L_{i+1}, t = \text{Type-III}$). Since we have searched the optimal states for Layer $L_{i+1}$, the optimal states in the last layer of the DNN model will be satisfied.
2.4.3 Partitioning Ratio

ACCPar allows the partition ratio to be adjusted for heterogeneous accelerators to balance the communication and computation costs of the individual accelerators. For an accelerator with a partitioning ratio $\alpha$, the computation and communication cost are both a function of $\alpha$ and a partitioning $p_{i,l}$, i.e., $E_{\text{cp}}(\alpha, p_{i,l})$ and $E_{\text{cm}}(\alpha, p_{i,l})$. To calculate the partition ratio for achieving the best performance, we need to find the ratio to balance the sum of computation cost and communication cost among two accelerator groups. From Equation (2.7), (2.8) and Table 2.4, 2.5 we can see that computation and communication cost are both linear with respect to the partition ratio: $E_{\text{cp}}(\alpha, p_{i,l}) = E_{\text{cp}}(p_{i,l})|_\alpha$ and $E_{\text{cm}}(\alpha, p_{i,l}) = E_{\text{cm}}(p_{i,l})|_\alpha$. For the accelerator with a partitioning ratio $\beta$ and a partitioning $p_{j,l}$, we can get $E_{\text{cp}}(p_{j,l})|_\beta$ and $E_{\text{cm}}(p_{j,l})|_\beta$. To determine the partitioning ratio, we just need solve the linear equation

$$E_{\text{cp}}(p_{i,l})|_\alpha + E_{\text{cm}}(p_{i,l})|_\alpha = E_{\text{cp}}(p_{j,l})|_\beta + E_{\text{cm}}(p_{j,l})|_\beta.$$  (2.10)

2.5 Evaluation

2.5.1 Evaluation Setup

We use nine DNNs to evaluate ACCPar: Lenet [232], Alexnet [119], Vgg11, Vgg13, Vgg19 [181], and Resnet18, Resnet34, Resnet50 [120]. We train Lenet on MNIST [233] dataset and other eight DNN models are trained with ImageNet [234].

<table>
<thead>
<tr>
<th></th>
<th>TPU-v2</th>
<th>TPU-v3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cores</td>
<td>$4 \times 2$</td>
<td>$4 \times 2$</td>
</tr>
<tr>
<td>FLOPS</td>
<td>180T</td>
<td>420T</td>
</tr>
<tr>
<td>HBM Memory</td>
<td>64GB</td>
<td>128GB</td>
</tr>
<tr>
<td>Memory Bandwidth</td>
<td>2400GB/s</td>
<td>4800GB/s</td>
</tr>
<tr>
<td># Accelerators</td>
<td>128</td>
<td>128</td>
</tr>
</tbody>
</table>

We build a in-house simulator to model the performance of the accelerator array in tensor
processing unit TPU-v2 and TPU-v3. In the simulation, we derive the tensor accessing traces (loading and storing) and partial sum computation (MULT and ADD) traces for the simulation and then we calculate the time consuming for the computation and data accessing. The trace granularity for FC layer is element-wise (i.e., 1) and for CONV is kernel-wise (e.g., 3x3). While TPU-v1 is designed for DNN inference [33], TPU-v2 and TPU-v3 target DNN training. Table 2.7 lists the specifications for these two accelerators. An accelerator is a board that holds the processing units — 2 cores per chip and 4 chip per board. The peak floating point operations per second (FLOPS) are 180T for TPU-v2 and 420T for TPU-v3, and the memory for the two accelerators are 64GB high bandwidth memory (HBM) and 128GB HBM [231]. The memory bandwidth for TPU-v2 is 2400GB/s. Since the memory bandwidth for TPU-v3 is not available, we assume a 4800GB/s memory bandwidth for TPU-v3. For the network, the maximum data rate per core is 2Gb/s [235]. We set the network data rate for TPU-v2 as 8Gb/s and that for TPU-v3 as 16Gb/s. The number of accelerators for TPU-v2 and TPU-v3 are both 128. The data format used in the DNN training is bfloat, Google’s 16-bit floating point data format for training. We also set the mini batch size to be 512.

To evaluate the effectiveness of ACCPAR, we compare it against data parallelism (DP) [108], “One Weird Trick” (OWT) [1] and HYPAR [2]. In DP, each accelerator maintains a local copy of DNN model, while training samples are partitioned among these accelerators. In OWT, the CONV layers in a DNN model is configured for data parallelism, and the FC layers are configured for model parallelism. In HYPAR, both CONV and FC layers can be configured for data parallelism or model parallelism to minimize overall communication. Data communication, operational computation and hardware heterogeneity are jointly considered in ACCPAR for collaborative optimization. Here we use DP as the baseline, and the performance and training throughput of OWT, HYPAR and ACCPAR are all normalized to the DP design.
2.5.2 Heterogeneous Array

We first evaluate the performance for a heterogeneous accelerator array using the same number of accelerators with different performance: 128 TPU-v2 accelerators and 128 TPU-v3 accelerators.

![Figure 2.5](image)

Figure 2.5: The speedup of data parallelism (DP), “one weird trick” (OWT) [1], HYPAR [2] and AccPar in a heterogeneous accelerator array.

The normalized performance of DP, OWT and HYPAR are shown in Figure 2.5. The geometric mean of speedup (the throughput improvement) in DP, OWT, HYPAR and ACCPAR are $1.00 \times$, $2.98 \times$, $3.78 \times$, $6.30 \times$, respectively. For Vgg series, ACCPAR can get a speedup up to $16.14 \times$, while the highest speedup of OWT and HYPAR are $8.24 \times$ and $9.46 \times$. For Resnet series, ACCPAR can get speedups from $1.92 \times$ to $2.20 \times$, while the ranges of speedup achieved by OWT and HYPAR are $1.22 \times$ to $1.38 \times$ and $1.03 \times$ to $1.04 \times$, respectively.

We see that the speedups achieved by ACCPAR is significantly higher than OWT and HYPAR. In Figure 2.5, on the four Vgg series, the speedups of OWT range from $5.33 \times$ to $8.25 \times$, and the speedups of HYPAR range from $5.92 \times$ to $9.46 \times$. However, the
speedups of AccPAR range from $9.75 \times$ to $16.14 \times$, which are significantly higher than OWT and HYPAR. The improvements of AccPAR come from: (1) the complete partitioning type search space, which includes all three types of tensor partitioning settings to reduce communication; and (2) the communication and computation joint optimization considering heterogeneity. With the communication and computation by different accelerators balanced by certain partitioning ratio, the idle time due to heterogeneous communication/computation capability under equal partitioning in OWT [1] and HYPAR [2] are greatly alleviated.

From Figure 2.5, we also notice that the speedups of Resenet series are lower than that of Vgg series. Between the two series, the main difference is that the model sizes of Vgg series are larger than those of Resnet series, while the computation densities of Resnet series are higher than those of Vgg series. Among the three basic tensor partitioning types, Type-II and Type-III partitions the weight of layers, i.e., the model, while Type-I partitions the feature maps, i.e., the data. In data parallelism, all layers are configured by Type-I. Thus, for DNNs with a large model size, such as Vgg series, Type-II and Type-III are favored due to the potential greater reduction of communication when partitioning the model. On the other side, for DNNs with higher computation density, Type-I is favored for greater potential reduction of the communication when partitioning feature maps. Moreover, because the difference of mode sizes among different layers is smaller than the difference of feature maps, the relative benefits achieved by Type-II and Type-III are smaller than those of Type-I. The above analysis explains the reason why speedups of Resenet series are lower than that of Vgg series, since DNNs in former series mainly benefit from Type-II and Type-III — model partition, the trend is true for not only AccPAR but also HYPAR and OWT. However, even for Resnet series, the speedups of AccPAR are considerably higher than OWT and HYPAR: the highest speedup by OWT and HYPAR on the Resnet series are $1.04 \times$ and $1.38 \times$ respectively, but the highest speedup by AccPAR is $2.20 \times$. This means that AccPAR is 112% better than OWT and 59% better than HYPAR.
2.5.3 Homogeneous Array

We then evaluate the performance for a homogeneous accelerator array, where 128 accelerators employed are of the same type, i.e., TPU-v3. The speedups of data parallelism (DP), “one weird trick” (OWT), HYPAR and ACCPAR are shown in Figure 2.6. The results are normalized to data parallelism. The geometric mean of DP, OWT, HYPAR and ACCPAR are 1.00×, 2.94×, 3.51×, 3.86×, respectively. For Resnet series, the DNNs are very “deep” (i.e., the number of layers is very large), and all layers except the last fully-connected layer are convolutional layers. While OWT, HYPAR, and ACCPAR all try to explore other parallelism settings or partitions rather than static data/model parallelism, we observe in the results that these three schemes eventually configure most of layers in Resnet series with data parallelism or Type-I partition. This is the reason why they achieve lower speedups on Resnet series than Vgg series. In comparison, the DNNs in Vgg series contain a variety of layers, so OWT, HYPAR, and ACCPAR can effectively explore larger search space with more diverse parallelism/partition settings. With homogeneous accelerator array, for a specific
DNN model, we observe the increasing speedups of DP, OWT, HYPAR and ACCPAR. It is the direct consequence of the increasing flexibility.

![Diagram showing partitioning types](image)

**Figure 2.7**: The selected partitioning types for the weighted layers in Alexnet. The number of hierarchies is 7 and the batch size is 128.

In Figure 2.7, we also show the selected partitioning types by ACCPAR for the weighted layers in Alexnet. In the three fully-connected layers, i.e., fc1, fc2 and fc3, Type-II and Type-III partitions are used to minimize the communication by maintaining a part of weight locally and communicating the feature maps. In the convolutional layers, i.e., cv1 to cv5, we can see that Type-I partition are mostly but not solely selected. ACCPAR allows all Type-I, Type-II and Type-III to be selected to reduce the total communication further. Especially, with the increase of hierarchy level, more layers are configured with Type-II or Type-III. This illustrates the importance of having a complete search space as in ACCPAR.

### 2.5.4 Scalability with Hierarchy Levels

In this section, we study the scalability of ACCPAR on various hierarchies on the heterogeneous accelerator array. Figure 2.8 shows the speedups of DP, OWT, HYPAR and ACCPAR on Vgg19 for hierarchy level from \( h = 2 \) to \( h = 9 \). With the increase of hierarchy level, we partition the tensor for a finer-grained level. From Figure 2.8 we see that the speedups
of OWT and HYPAR tend to be saturated, while the speedups of ACCPAR continue to increase with the increase of the partitioning hierarchies. For OWT, the convolutional layers are always configured with data parallelism and the fully-connected layers are always configured with model parallelism. While it provides more flexible than configuring all layers with data parallelism in DP, it is still a static configuration. HYPAR and ACCPAR are flexible configurations because they do not flexibly set a class of layer to be a specific parallelism or partition, instead they flexibly explore the possible configuration to minimize the communication or cost. Compared with HYPAR, ACCPAR benefits from a complete partitioning type sets and the heterogeneity-aware configuration. The flexibility of the four schemes from low to high is: DP ≺ OWT ≺ HYPAR ≺ ACCPAR. Table 2.8 shows the comparison of DP, OWT, HYPAR, and ACCPAR.

**Table 2.8**: The comparison of flexibility of DP, OWT, HYPAR and ACCPAR.

<table>
<thead>
<tr>
<th></th>
<th>DP</th>
<th>OWT</th>
<th>HYPAR</th>
<th>ACCPAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static</td>
<td>Static</td>
<td>Flexible</td>
<td>Flexible</td>
<td></td>
</tr>
<tr>
<td>Low</td>
<td>High</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Chapter 3

Layer-wise Pipelined Parallelism for Deep Learning Accelerators

3.1 Background

3.1.1 Basics of Deep Neural Network

The core component of many deep learning applications (on computer vision [236–240], data mining [241–244], language processing [245–249] and etc.) is convolutional neural network (CNN).

![Figure 3.1: Convolutional neural network (CNN).](image)

Figure 3.1 illustrates the organization of an example CNN, which has 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. We also denote the size of $d$ in layer $l$ as $(X_l \times Y_l \times C_l)$, so $0 \leq x \leq X_d - 1, 0 \leq y \leq Y_d - 1, 0 \leq c \leq C_d - 1$ 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 [233], $K_x$ and $K_y$ are both 5).

d_{l+1} is computed as:

$$
d_{l+1}[x,y,c] = \sum_{c_l=0}^{C_l-1} \sum_{k_x=0}^{K_x-1} \sum_{k_y=0}^{K_y-1} K_l[k_x,k_y,c_l,c] \times d_l[x+k_x,y+k_y,c_l]. \tag{3.1}
$$

To perform Equation (3.1), 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:

$$
d_{l+1}[x,y] = \frac{1}{K_xK_y} \sum_{k_x=0}^{K_x-1} \sum_{k_y=0}^{K_y-1} d_l[K_x x + k_x, K_y y + k_y, c]. \tag{3.2}
$$

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 $\vec{d}_l$ and $\vec{d}_{l+1}$). If the previous layer is convolution or pooling, the size of $\vec{d}_l$ is $X_l \times Y_l \times C_l$. If the previous layer is also inner product, then the size of $\vec{d}_l$ is the size of the output vector from $l$. $\vec{d}_{l+1}$ is a $n \times 1$ 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 $\vec{d}_l$. $\vec{b}$ is a vector of bias.

The vector of $l+1$ is computed as:

$$
\vec{d}_{l+1} = W_{l+1-l}\vec{d}_l + \vec{b}. \tag{3.3}
$$
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) \(\text{max}(0,x)\).

### 3.1.2 Data Forward and Backward in a Neural Network

Neural network has two phases: training (learning) phase and testing (inference) phase. In testing phase, the weights of a neural network have been determined and the task is to use the network on input samples, e.g. to recognize who is the person in an image. In testing phase, input data flow through the layers consecutively in forward direction. It is shown in Figure 3.2, \(d_{l-1}\) is the output of layer \(l-1\) and the input of layer \(l\). The forward process can be represented as the two equations indicated above. The computation is the same as what we discussed in Section 3.1.1.

Before a neural network for a specific application can be deployed, it needs to be trained and generate the weights. This is done in training phase, where data not only move forward but also backward, — to update weights based on errors. In training phase, a cost function is defined to quantitatively evaluate how well the outputs of a neural network compare to 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 \(J(W,b) = -\sum_{i,j} \mathbf{1}(y^i = t^i) \log p(y^i = t^i)\) is the softmax loss function.

The error \(\delta\) 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 \(\delta_l = (W_{l+1})^\top \delta_{l+1} \circ f'(u_l)\).

---

1In this chapter, we use \(\nabla X\) to denote the partial derivative of a loss function \(J(\cdot)\) with respect to a variable \(X\), i.e., \(\nabla X = \frac{\partial J}{\partial X}\).
Figure 3.2: Data flows between two adjacent layers

And with a ReLU activation function, the error can be rewritten as
\[ \delta_l = (W_{l+1})^\top \delta_{l+1} \circ f'(d_l) \]
So that the backward partial derivatives to \( W_l \) is \( \frac{\partial J}{\partial W_l} = d_{l-1}(\delta_l)^\top \). And the backward partial derivatives to \( b_l \) is \( \frac{\partial J}{\partial b_l} = \delta_l \). Now we can use the gradient descent method to update the weights of neural network.

We see that the training phase is more complex than testing phase due to weight updates. It also introduces more data dependencies (e.g. weight update to a layer depend on the previous layer’s error and the data of the earlier forward computation). Therefore, training phase is time consuming and could take 14 to 21 days [181].

### 3.1.3 ReRAM Basics

The resistive random access memory (ReRAM) [125] is an emerging non-volatile memory with appealing properties of high density, fast read access and low leakage power. ReRAM has been considered as a promising candidate for future memory architecture. A primary
application of ReRAM is to be used as an alternate for main memory [250–252]. Figure 3.3 (a) demonstrates the metal-insulator-metal (MIM) structure of an ReRAM cell. It has a top electrode, a bottom electrode and a metal-oxide layer sandwiched between electrodes. By applying an external voltage across it, an ReRAM cell can be switched between a high resistance state (HRS or OFF-state) and a low resistance state (LRS or On-state), which are used to represent the logical "0" and "1", respectively, as shown in Figure 3.3 (b). The endurance of ReRAM could be up to $10^{12}$ [253, 254], alleviating the lifetime concern faced by other non-volatile memory, such as PCM [255].

Figure 3.3: Basics of ReRAM.

The ReRAM features the capability to perform in-situ matrix-vector multiplication [256, 257] as shown in Figure 3.3 (c), which utilizes the property of bitline current summation in ReRAM crossbars to enable computing with high performance and low energy cost. While conventional CMOS based system showed success on neural network acceleration [27, 43, 101], recent works [88–90, 92] demonstrated that ReRAM-based architectures offer significant performance and energy benefits for the computation and memory intensive neural network computing.

Recent works [88–90] demonstrated that ReRAM-based architecture could accelerate matrix-vector multiplication in neural computation. However, the current works lack important features to efficiently execute complete deep learning applications. First, they do not support training that involves weight update and complex data dependencies. Second, the deep pipeline in ISAAC [89] is not beneficial to training phase where the length
of consecutive images that could enter the pipeline is limited by batch size. Also, it is vulnerable to pipeline bubble.

### 3.2 PIPELAYER Architecture

In this section, we first analyze the general training procedure and outline the required mechanisms to support it. Then we present the ideas of intra- and inter-layer pipeline that explore the parallelism and support training naturally.

PIPELAYER architecture directly leverages ReRAM cells to perform computation without the need for extra processing units. Resistive random access memory (ReRAM or RRAM) is a type of non-volatile memory that stores information by changing the cell resistances. This work focuses on a subset of resistive memories, — metal-oxide ReRAM, which uses metal oxide layers as switching materials. More details on the technology could be found in [88, 253–255, 258–262].

Our design partition the ReRAM-based main memory into two regions: morphable subarrays (Morp) and memory subarrays (Mem). The morphable subarrays can perform both computation and data (weights) storage, the two modes can be configured. In computation mode, the morphable subarrays perform matrix-vector multiplications. The memory subarrays are similar to conventional memory and have data storage capability. Due to space limit, readers could refer to [88] for details on the ReRAM basics.

#### 3.2.1 Training Support

For simplicity, Figure 3.4 shows the configuration of PIPELAYER to process the training of a 3-layer CNN. The rectangles are one layer of morphable subarrays. The circles are memory subarrays to store intermediate results transferred between morphable subarrays for different layers.
Data dependencies exist in forward and backward computations. The computation timing and dependencies are shown in Figure 3.4. 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. At the end of $T_1$, the results are written to a memory subarray, $d_1$. For clarity, we use red dashed lines to show the system state between two consecutive cycles. Note that the cycles in Figure 3.4 (e.g. $T_0$, $T_1$, ...) are logical cycles, depending on implementation, each cycle could take several physical clock cycles. We will show in Section 3.2.2 that the number of physical cycles is determined by parallelism and hardware resource. The term "cycle" from this point all refer to the logical cycle. The solid lines between rectangles (morphable subarrays) and circles (memory subarrays) indicate the data dependencies. The following forward layers are performed similarly.

The results of forward computation are eventually stored in $d_3$ before backward computation starts at $T_4$. The backward computations generate errors ($\delta_l$) ($l$ is the layer) and partial derivatives to $b$ and $W$ ($\nabla b_l$ and $\nabla W_l$). As the first step, the error for the last (third) layer ($\delta_3$) is computed in $T_4$. It is stored in memory subarrays served as the input for computations in the next cycle.
In $T_5$, two computations happen in parallel: (1) partial derivatives ($\nabla W_3$) is computed by previous results in $d_2$ and $\delta_3$; (2) errors ($\delta_2$) of the second layer is computed from $\delta_3$. Both of the computations depend on $\delta_3$, which is computed in $T_4$. $\nabla W_3$ is stored in memory subarrays, which will be used to update weights in $A3$ and $A32$ later. Note that $(W_3)^*$ is a reordered form of $W_3$ and will be discussed in Section 3.3.3. Finally, in $T_7$, the partial derivatives for the first layer ($\nabla W_1$) is computed and stored in memory subarrays.

In training, batch size (denoted as $B$) is used to specify the number of images that can be processed before a weight update. If $B = 1$, $\nabla W_1$, $\nabla W_2$ and $\nabla W_3$ are used to update the weights in $A1$, $A2$, $A3$ and $A22$, $A32$. If $B > 1$, $\nabla W_1$, $\nabla W_2$ and $\nabla W_3$ are stored in the buffers in a special way so that the average of all partial derivatives can be computed automatically.

Training phase is more complicated than testing phase because it involves data dependencies between outputs from previous layers and the compute of partial derivatives and errors for weight update. Memory subarrays are used as buffers to store intermediate results, this greatly reduce data movements and energy consumption by avoiding data being transferred across memory hierarchy. When not necessary (e.g. with only testing), some of them can be converted to morphable subarrays.

Within a cycle, different operations could be performed depending on the phase of computation. Table 3.1 shows the operations in four possible cases. In an implementation, the cycle time has to allow the longest sequence of operations to fit. Note that the cycle time is not a bottleneck, because memory is much slower than processors.

### 3.2.2 Intra Layer Parallelism

#### Data Input and Kernel Mapping

We propose a basic scheme of data input and the mapping of kernels to a ReRAM array, illustrated in Figure 3.5. In this example, data of layer $l$, kernels and data of layer $l + 1$ have a size of $114 \times 114 \times 128$, $3 \times 3 \times 128 \times 256$ and $112 \times 112 \times 256$, respectively.
Table 3.1: Operations in a cycle.

<table>
<thead>
<tr>
<th></th>
<th>Memory Read</th>
<th>Spike Driver</th>
<th>Morp. Subarray</th>
<th>I&amp;F</th>
<th>Act. Activation</th>
<th>Mem. Subarray</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forward</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Backward 1</td>
<td>$d_L$</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>*</td>
<td>$\nabla b_L$</td>
</tr>
<tr>
<td>Backward 2</td>
<td>$\nabla b_l$</td>
<td>✓</td>
<td>$A_{L1}(d_{l-1}):A_{L2}((W_{l-1})^*)$</td>
<td>✓</td>
<td>✓</td>
<td>$\nabla W_l$</td>
</tr>
<tr>
<td>Update</td>
<td>$\nabla W_l$</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td>$W_l$</td>
</tr>
</tbody>
</table>

$L$: number of layers; $l$: layer index.

For each channel in layer $l+1$, the corresponding kernel has a size of $3 \times 3 \times 128$. That means the size of an input vector (yellow bar) to the ReRAM array at one cycle is $1152 \times 1$ (actually it should be $1153 \times 1$, we neglect the bias for express clarity). And in the next cycle, the kernel window shifts right or down and we have the another $1152 \times 1$ yellow bar from input. Consequently, input data enter ReRAM array in a sequential mode. It takes 12544 cycles to get all outputs of layer $l+1$.

Mapping all kernels to the same ReRAM array is not realistic. Weights of one kernel is mapped to cells of one bit line, for example, the blue cuboid is mapped to the blue bar in the array and it is the same case for the red, green and the rest cuboids, resulting in 256 bit lines and 1152 world lines for the ReRAM array. Therefore, the ReRAM array needs to have a size of $1152 \times 256$. In Section 3.2.2, we will present a more realistic design, but before that, let us explore the approach of ISAAC [89] to improve the naive design.

**Pipeline in ISAAC [89]**

The pipeline used in ISAAC [89] attempts to improve the throughput of the architecture by computing small tiles of a layer and then let the next layer use the partial output as input in the next cycle. It could notably improve through put by the deep pipeline. The advantage is that if a large number of input data could be fed into the pipeline continuously, after a (large) number of initial cycles to fill the pipeline, results could be generated each cycle.
Figure 3.5: A basic scheme for data input and kernel mapping.

Unfortunately, this assumption is untrue for neural network training phase, where weights are updated at the end of a batch. The new inputs in the next batch need to be processed based on the updated weights and we cannot have a large number of consecutive input data for the pipeline. Therefore, the pipeline design in ISAAC does not fit well for training phase.

Another issue is the vulnerability to pipeline bubble, leading to execution stall. Consider a network where all kernels are of size $2 \times 2 \times 1$. One point $p_{50}$ in layer $l_5$ depends on 4, 16, 64, 256 points in layer $l_4$, $l_3$, $l_2$ and $l_1$, which means the computation of $p_{50}$ is stalled when any of the 340 ($= 4 + 16 + 64 + 256$) points is delayed, which could further stall computations depending on $p_{50}$. Moreover, data dependencies are complex in training phase, layer $l$ may not only depend on layer $l - 1$, but also from earlier layers (e.g. $\delta_1$ in 3.4 depends on $\delta_2$ and early data $d_1$). It further increase the chance of stalls. [89] mentioned pipeline imbalance issue but similar issues could easily occur with more complex training
phase and it is extremely hard to predict and avoid all of them.

**Parallelism Granularity**

![Diagram showing parallelism granularity](image)

**Figure 3.6**: A balanced scheme for data input and kernel mapping.

The ReRAM array size could be made feasible by partition. We can decompose the $1152 \times 256$ matrix to a group of 18 (=9 × 2) matrices and map each of them to a $128 \times 128$ ReRAM array (shown in the right part of Figure 3.6). We can get the right results by collecting array outputs horizontally and summing them vertically.

To improve performance, we define a metric called parallelism granularity, denoted as $G$, indicating the number of duplicated copies of ReRAM arrays that store the same weights. If $G = 1$, the design is equivalent to the naive scheme. If $G = 12544$, the results of a layer could be generated in just one cycle but the hardware cost is prohibitive.

Essentially, parallelism granularity allows explore the trade-off between hardware resource of ReRAM array and performance. A good trade-off requires a carefully chosen $G$. Figure 3.6 shows an example with $G = 256$. 

51
3.2.3 Inter Layer Parallelism

Figure 3.7: Layer-wise pipeline.

In training, the input data are normally processed in batch. The inputs in the same batch are all processed based on the same weights at the start of the batch. The weight updates due to each input are stored and only applied to at the end of a batch. Therefore, no dependency exists among data inputs inside a batch. We propose an architecture to support pipelined training. The performance gain is due to the fact that $B$ is normally much larger than 1 (e.g. 64), otherwise, each input needs to be processed sequentially.

Based on Figure 3.4, the pipelined execution is shown in Figure 3.7. The execution with intra- and inter-layer pipeline could achieve high throughput without the drawbacks of previous work. In the execution, a new input can enter the pipeline every cycle within a batch. At the end of batch, a new input belonging to the next batch cannot enter the pipeline until all inputs in previous batch are processed and weights are updated. In another word, a new batch has to wait the “tail” of previous batch to drain from the pipeline.

Figure 3.8 (a) shows the latency of the baseline architecture without pipeline. Assuming the neural network has $L$ layers, we can see that the forward computation of one input takes $L$ cycles, the backward computation of one input takes $(L + 1)$ cycles, at the end of a batch, one cycle is dedicated to the weight update. Therefore, a batch takes $(2L + 1)B + 1$ cycles.
For a total number of $N$ inputs, the total number of training cycles is $(2L + 1)N + N/B$.

Now let us consider the pipelined execution, the latency is illustrated in Figure 3.8 (b). Within a batch, a new input could enter every cycle. Similar to the baseline case, the first weight update is generated after $(2L + 1)$ cycles. Then there will be $(B - 1)$ cycles until the end of batch. Finally, there is one cycle to apply all weight updates within the batch. Since the forward and backward computation are intertwined with each other, we do not distinguish the forward and backward cycles. From the above analysis, the total number of cycles to process $N$ inputs with $L$ layers is $(N/B)(2L + B + 1)$.

**Figure 3.8:** Latency of PIPEAYER architecture.

**Figure 3.9:** Memory subarray design.
The major implication of pipelining is the increased requirement of memory subarrays. The problem is illustrated in Figure 3.9. Consider the same example in Figure 3.4, since now A1 will produce output every cycle, we need to have more buffers so that the data which will be used later are not over written. Specifically, referring to Figure 3.4, $d_1$ computed in $T_1$ will be used after 5 cycles, in $T_6$. Therefore, between A1 and A2, 5 buffers are needed. A1 should logically keep a pointer pointing to the current buffer that the output should be written to. After an output, the pointer is increased by one. Logically we implement circular buffers. When it reaches 5, it will wrap around and return to the first entry. When the output is written again to the first entry, the useful data ($d_1$) used to compute $\nabla W_2$ can be safely overwritten because $\nabla W_2$ has been computed. According to this, in Figure 3.4, A1 will write 1st-, 2nd-, 3rd-, 4th- and 5th-entry in buffer between A1 and A2 in $T_1$, $T_2$, $T_3$, $T_4$, $T_5$, consecutively. Since $\nabla W_2$ is computed from the data in the 1st-entry in the buffer in $T_5$, in the next cycle, A1 could overwrite the 1st-entry of the buffer. In general, the buffer requirement at $l$-th layer (assuming the total number of layers is $L$) is $2(L - l) + 1$.

Another implication of pipelining is that at the same cycle, there will be a read and a write on the same buffer. To support this, we need to duplicate the single buffer. This happens for the buffer at $d_3$, $\delta_1$, $\delta_2$, $\delta_3$. Table 3.2 compares the number of cycles and the cost of (groups of) arrays of nonpipelined and pipelined architectures.

**Table 3.2**: Cycles and cost in PIPELAYER architecture.

<table>
<thead>
<tr>
<th></th>
<th>Non-pipelined</th>
<th>Pipelined</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Forward Cycles</strong></td>
<td>$LN$</td>
<td>$(N/B)(2L+B+1)$</td>
</tr>
<tr>
<td><strong>Backward Cycles</strong></td>
<td>$(L+1)N + N/B$</td>
<td>$(L+1)N + N/B$</td>
</tr>
<tr>
<td><strong>Morp Subarrays</strong></td>
<td>$GL + G(2L - 1)$</td>
<td>$GL + G(L - 1) + BL$</td>
</tr>
<tr>
<td><strong>Mem Subarrays</strong></td>
<td>$2L$</td>
<td>$2L$</td>
</tr>
</tbody>
</table>

$G$: parallelism granularity; $L$: number of layers; $B$: batch size; $N$: number of total input images.
3.3 Implementation of PIPELAYER

3.3.1 Overall Architecture

Figure 3.10 shows the architecture of PIPELAYER. Our design leverages ReRAM cells to perform computation without the need for extra processing units. The design partitions the ReRAM-based main memory into two regions: morphable subarrays and memory subarrays. The morphable subarrays can perform both computation and data (weights) storage, the two modes can be configured. In computation mode, the morphable subarrays could perform matrix-vector multiplications. The memory subarrays are the same as conventional memory and have data storage capability. They are used to store intermediate results between layers and keep data that will be used in Figure 3.10 only shows the morphable subarrays and memory subarrays.
a. Spike Driver. To reduce the area and energy overhead of voltage-level based input scheme in [88], we use a weighted spike coding scheme similar to ISAAC [89]. The spike driver converts the input to a sequence of weighted spikes. In weight update, spike driver serves as write driver to tune weights stored in the ReRAM array.

b. Integration and Fire. It is a required component of a spike-based scheme. It integrates input current and generates output spikes. The output spikes are connected to a counter, so essentially the analog currents are converted into digital values. The data input scheme used in [89] is similar as our spike input, both eliminating DACs, but our integration and fire components eliminates ADCs while [89] keeps.

c. Activation. It implements the activation function defined in a CNN algorithm. When morphable subarrays are configured as memory, it is bypassed.

d. Connection. This component connects morphable and memory subarrays. The outputs from morphable subarrays (when they are in computation mode) need to be written to memory subarrays so that they are used as input for the morphable subarrays for the next layer in next cycle.

e. Control. It offloads the computation from the host CPU and orchestrates the data transfers between memory subarrays and morphable subarrays in training and testing with based on the algorithm configurations (e.g. batch size).

### 3.3.2 Component Designs

#### Spike Driver

We use a weighted spike coding scheme. For a $N$-bit input value, we need $N$ time slots for potential spikes. The spike driver design is shown in Figure 3.10 (a). Inside the driver, $N$ levels of reference voltage, ranging from $V_0/2^{N-1}$ to $V_0/2$ are generated. A timing control component shifts key $K1$ to connect one of the reference voltages at one time slot. Note that the shifting is non-decreasing: the voltage of output spike increases as time slot progresses,
thus the data input is of a Least Significant Bit First (LSBF) mode. $K_2$ is controlled by the input data to decide whether the spike output is generated or not.

Another function of spike driver is to write or program data to a subarray, where $K_1$ is connected to a programing voltage. $K_2$ tunes the resistance of a ReRAM cell controlled by input data which serve as programing control sequences. Spike divers are reused between adjacent subarrays.

**Integration and Fire**

Figure 3.10 (b) shows the design of Integration and Fire circuit. A controlled current source serves as a current follower to inject an equivalent strength current as that from a bit line to a capacitor. As a result, the voltage on this capacitor is accumulated.

When the voltage on the capacitor increases to the threshold ($V_{th}$) of the comparator, an output spike is generated and counted by a counter. Note that the current on the bit line is the accumulation of products of spikes (input data) and ReRAM cell conductances (corresponding weights). For a fixed threshold $V_{th}$, a $K$ times stronger current will makes the comparator to generate $K$ times of output spikes. As a result, the number of spikes generated by the comparator is actually the accumulation of products of input data and corresponding weights. A group of integration and fire components are connected to bit lines of a subarray and they can generate an vector in parallel.

**Activation Function**

Figure 3.10 (c) shows the design of activation component. The activation component consists of a subtractor and a look up table (LUT). Since positive weights and negative weights are mapped to two subarrays, we need to collect the results $D_P$ from a positive subarray and $D_N$ from a negative subarray. Our activation component is configurable by different LUTs to realize the activation function defined in specific algorithm. In this
work, we mainly focus on rectified linear unit (ReLU), which is widely used in deep neural networks. A register is used to keep the max value of a sequence, which realizes max pooling.

### 3.3.3 Error Backward

Before generating weight updates by partial derivatives, the error for each layer should be computed. In forward progress, data propagate from layer $l$ to layer $l+1$, while in backward progress, errors propagate from layer $l$ inversely to layer $l−1$. Figure 3.11 shows three kinds of error backward in convolutional neural networks.

![Figure 3.11: Error backward for (a) activation function, (b) max pooling layer and (c) convolution layer.](image)

Error backward for activation function is computed as $\delta_{l−1} = f'(u_l) \circ \delta_l$, which is an element-wise operation. The yellow squares in Figure 3.11 (a) represent an element of error $\delta_l$ in layer $l$ and error $\delta_{l−1}$ in layer $l−1$. Note the activation function used is ReLU, which means $f'(\cdot)$ is either 0 or 1. The backward is to AND (&) 0 or 1 with $\delta_{l−1}$, which is done by activation components (component c in Figure 3.10). Thanks to ReLU, we have $f'(u_l) = f'(d_l)$, there is no need to store intermediate data $u_l$ in forward progress.

Figure 3.11 (b) shows the error backward for a pooling layer. Error elements in layer $l−1$ are copied to the positions which is the max values in windows of data in layer $d_l$. For example, the pink square, 3, in layer $l−1$, is copied to the lower right of the pink window in layer $l$, while the rest three squares in the window get 0. The four squares in layer $l$ are propagated to the upper left, lower left, lower right and upper right of four windows in
layer \( l-1 \). Mathematically, max pooling can be viewed as one kind of activation function. The error backward for a pooling layer is also performed in the activation component (component c in Figure 3.10). With \( d_l \) already stored in memory subarrays, the index for the max element in a window can be found.

For error backward, we can reuse the logic in a convolution layer. In Figure 3.11 (c), a \( 3 \times 3 \) kernel is used in forward progress. In the forward progress, to generate the nine elements in the red window in layer \( l \), all of the (light, medium and dark) blue squares in layer \( l-1 \) are convoluted with the \( 3 \times 3 \) kernel and the dark blue square (with red edges) are exactly multiplied with each elements in the kernel. In the backward, to get the error for the dark blue square, we can just convolute the nine elements in the red window in layer \( l \) with the kernel.

\[
\delta_{l-1} = \text{conv2}(\delta_l, \text{rot180}(K), \text{'full'})
\]

is the computation for error backward for convolution layer. For elements at edges, zero paddings are added. Figure 3.12 shows the error backward for the convolution layer in Figure 3.5, where error in layer \( l-1 \) are zero-padded by 2 at each edge, so the length and width increase to 116. Also, the kernels are reordered. In this way, we can perform convolutions by using the data input and kernel mapping scheme discussed in Section 3.2.2.
### 3.3.4 Weight Update

#### Computing Partial Derivatives

The partial derivatives to bias is the sum of all related error: \( \nabla b = \sum_{u,v}(\delta_l)_{u,v} \). It is done by reading bitline with the input spikes representing 1. The partial derivative to a weight is the sum of element-wise products of error and patches/windows of data: \( \nabla W = \sum_{u,v}(\delta_l)_{u,v} \cdot (d_{l-1})_{u,v} \). It can be converted to convolution.

In Figure 3.13, the convolution of one blue channel of error in layer \( l-1 \) and the total 128 channels of data in layer \( l \) is the partial derivatives to weights of the blue kernel (size of \( 3 \times 3 \times 128 \)). All the kernels (red, green and others) are similar. In the convolution, data in layer \( l \) (yellow slices) can be viewed as convolution kernels while error in layer \( l-1 \) (doted slices) can be viewed as convolution data. Therefore, the convolution is done by mapping the yellow slices to ReRAM arrays and sending the backward error to the arrays.

![Figure 3.13: The computation of partial derivatives for kernels.](image)

**Writing New Weights to Morphable Subarrays**

With the partial derivatives for weights \( \nabla W \) and bias \( \nabla b \), old weights and bias that read from morphable subarrays. The new (updated) weights and bias can be computed and written to morphable subarrays.

In weight updating, we need to compute the average of the \( B \) (\( B \) is the batch size)
partial derivatives, and this can be done by the input spikes representing $1/B$. With current accumulation property of bitline, the read out data is actually the averaged partial derivatives. At the same time, old weights are read out and new weights are computed. The LUT in activation components are bypassed when reading. Different from default reading, $D_P$ of the subtractors are connected to old weights and $D_N$ connected to the averaged partial derivatives. As a result, the output data are the new weights/bias, which is (old weights/bias minus averaged partial derivatives). And finally, the new weights/bias to the corresponding morphable subarrays.

### 3.4 Discussion

#### 3.4.1 Resolution and Accuracy

In practice, the ReRAM cell can only support limited precisions. For example, [88] used 6-bit ReRAM cells and [256] used 5 bits. Such limitation does not affect some neural networks due to their inherent error tolerance.

![Figure 3.14: Tradeoff between resolution and accuracy.](image)

We conducted a set of experiments to understand the trade-off between resolution (of ReRAM cells) and accuracy (of applications). We use five applications: M-1, M-2 and M-3 are multilayer perceptron neural networks while M-C and C-4 are convolutional neural networks. Figure 3.14 shows the normalized accuracy (to float resolution in original
implementation) of using different number of bits. We see that for M-1, M-2, M-3, using 4-bit just slightly decreases the accuracy. However, the accuracy is more sensitive to resolution for convolutional neural networks, as accuracies of M-C and C-4 drop sharply from 3-bit. Especially for C-4, even at 4-bit resolution, the normalized accuracy is only around 0.2.

When higher resolution is needed, we can use multiple arrays with lower resolution, similar to [89]. The default resolution of PIPELAYER is 16-bit, the same as [27, 89], and the resolution of ReRAM cells used in PIPELAYER is 4-bit.

![Figure 3.15: Weight partition.](image)

In testing phase, we can easily get 16-bit results by adding four shifted 4-bit results, as shown in Figure 3.15 (a). Here, the same data input four groups of 4-bit arrays. The four groups store 4-bit weights for the the 15..12, 11..8, 7..4 and 3..0 segment respectively. We can shift and add the results from the four groups and get the results for 16-bit resolution. In training phase, we need to first read the old four segments of 4-bit weights then shift them to get the old weights. With the partial derivatives, we get the new weights and write back to the four groups to finish the update. It is shown in figure 3.15 (b).

### 3.4.2 Application Programming Interface

Thanks to the layer wise inter layer pipeline of PIPELAYER, the system can be configured by layer interactively. Before discussing layer wise function invocation, we provide
two functions, Copy_to_PL and Copy_to_CPU to transfer data from CPU main memory to PIPELAYER or inversely. For one layer configuration, a topology set function Topology_set is invoked. This function configures the connections and datapath of $G$ groups of arrays as shown in Figure 3.6, while $G$ can be set by programmer or automatically optimized by compiler. Next, the weight load function Weight_load is invoked, which load pre-trained weights to the arrays in testing phase, or initial weights in training phase. After all layers are configured, running with pipeline can be started by function Pipeline_Set. Finally, one of the running mode functions Train or Test starts the execution.

3.5 Evaluation

3.5.1 Benchmarks

The benchmark networks used are based on databases MNIST [183] and ImageNet [182]. MNIST is a classical database of handwritten digits for pattern recognition. It consists of a training set of $60K$ images and a inference set of $10K$ images. And the labels for the images range from 0 to 9. Images of MNIST are 28-by-28 gray images. ImageNet provides $14.2M$ images and $21.8K$ indexed synsets. Most of the largest and state-of-the-art performance neural networks are based on ImageNet.

For ImageNet, we select six popular large scale networks, i.e. AlexNet [119], and VGG-A, VGG-B, VGG-C, VGG-D, VGG-E [181]. The topologies and hyperparameters of these networks can be found from the references. For MNIST, we build four neural networks as our benchmarks, the hyper parameters are shown in Table 3.3.

In Table 3.3, $N_1$-$N_2$ represents an inner product layer where there are $N_1$ perceptrons in layer $l$ and $N_2$ neurons in layer $l + 1$ (e.g. $784 - 100$ represents an inner product layer where there are 784 neurons in the first layer and 100 neurons in the second layer). Conv$K$x$C$ represents a convolution layer where the kernel size is $K$ by $K$ and the number of channels in the next layer is $C$. 
### Table 3.3: Hyper parameters of networks on MNIST.

<table>
<thead>
<tr>
<th>Network</th>
<th>Hyper Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mnist-A</td>
<td>784-100-10</td>
</tr>
<tr>
<td>Mnist-B</td>
<td>784-500-250-10</td>
</tr>
<tr>
<td>Mnist-C</td>
<td>784-1500-1000-500-10</td>
</tr>
<tr>
<td>Mnist-0</td>
<td>conv5x10-360-10</td>
</tr>
</tbody>
</table>

### 3.5.2 Experiment Setup

In our experiments, we compare PIPELAYER with a GPU-based platform. Benchmarks running on the platform are based on the widely used framework Caffe [263].

To run the GPU platform, we set the running mode in the Caffe prototxt to GPU. The GPU used is the newest released GTX 1080. Table 3.4 shows the parameters of our GPU platform. The run times for GPU platform are measured by caffe and the energy costs are measured by the tool `nvidia-smi` provided by NVIDIA CUDA.

### Table 3.4: Configurations of the GPU platform.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CPU:</strong> Intel Xeon E5-2630 V3, 8 cores, 2.40 GHz, ((32 + 32))KB L1 Cache, ((32 + 32))KB L1 Cache, 20 MB L3 Cache.</td>
<td>128 GB</td>
</tr>
<tr>
<td><strong>Storage</strong></td>
<td>1 TB</td>
</tr>
<tr>
<td><strong>Graphic Card</strong></td>
<td>NVIDIA Geforce GTX 1080</td>
</tr>
<tr>
<td><strong>Architecture</strong></td>
<td>Pascal</td>
</tr>
<tr>
<td><strong>CUDA Cores</strong></td>
<td>2560</td>
</tr>
<tr>
<td><strong>Base Clock</strong></td>
<td>1607 MHz</td>
</tr>
<tr>
<td><strong>Compute Capability</strong></td>
<td>6.1</td>
</tr>
<tr>
<td><strong>Graphic Memory</strong></td>
<td>8 GB GDDR5X</td>
</tr>
<tr>
<td><strong>Memory Bandwidth</strong></td>
<td>320 GB/s</td>
</tr>
<tr>
<td><strong>CUDA Version</strong></td>
<td>8</td>
</tr>
</tbody>
</table>

To evaluate PIPELAYER, we build a simulator based on NVSim [184]. The read/write latency, read/write energy cost used in the simulator are 29.31 ns/50.88 ns per spike, 1.08 pJ/3.91 nJ per spike, which are reported in [264]. And the area model is based on data...
reported in [252].

3.5.3 Performance Results

Figure 3.16 shows the performance comparisons of training and inference (testing) phase. For each application, we report the execution time comparison of GPU, non-pipelined and pipelined PIPELAYER (PIPELAYER). The GPU implementation is used as the baseline and running times of benchmarks on different settings are normalized to it.

**Figure 3.16:** Speedups of networks in both training and inference.

Compared to GPU platform, the geometric mean of overall (training + inference), training and inference speedup achieved by pipelined PIPELAYER architecture are 33.85x, 53.22x and 42.45x, respectively. Among all applications in both training and inference, the highest speedup achieved by non-pipelined PIPELAYER is 20.81x, while for pipelined PIPELAYER, the highest speedup is 146.58x.

PIPELAYER architecture archives lower speedups in training phase than in inference phase. It is because training requires more intermediate data processing and weight updates.

The pipelined PIPELAYER achieves a geometric mean speedup of 42.45x, significantly better than non-pipelined variants because the computations and data movements are performed in a highly parallel manner.

Intuitively, as the depth and layers of the neural networks increase, PIPELAYER architecture will achieve higher speedups compared to conventional architecture. However, we
found that it is not always the case. For example, the speedup of Mnist-C is larger than AlexNet in training. That is because Mnist-C is a multilayer perceptron network, whose weights are all matrices and can be directly mapped to ReRAM arrays.

### 3.5.4 Energy Results

Figure 3.17 shows energy savings in GPU platform and PIPELAYER (pipelined). The higher the bars are, the more energy efficient the corresponding architectures are. We show results for both training and inference.

![Energy Savings Graph](image)

**Figure 3.17**: Energy savings for PIPELAYER.

The geometric mean of energy saving compared to GPU in training and inference are 6.52x and 7.88x, respectively and the overall geometric mean of energy saving is 7.17x. The highest energy saving for training and inference are 27.03x (Mnist-C) and 70.03x (Mnist-A), respectively. We see that energy saving of PIPELAYER in training is slightly less than that in inference. This is due to the extra morphable and memory subarrays needed.

In summary, we see from the results that PIPELAYER architecture provides significant performance improvement with drastic energy reduction. It comes from two sources. First, the computation cost is reduced, the matrix-vector multiplications commonly used in CNNs
are performed by analog circuits, which is much more energy efficient than conventional GPU implementation. Second, the data movements in memory hierarchy are greatly reduced. In PIPELAYER, the data movements between different levels of memory hierarchies in conventional architectures are replaced by data movements in memory itself through layers.

### 3.5.5 Sensitivity of Parallelism Granularity

This section analyzes the effect of parallelism granularity on PIPELAYER. Table 3.5 shows the default parallelism granularity for each convolution layer of 5 VGG networks. We use a scalar $\lambda$ to enlarge or reduce the parallelism granularity of a network for $\lambda$ times.

**Table 3.5:** Optimized parallelism granularity $G$ configurations of each convolution layer in VGGs.

<table>
<thead>
<tr>
<th>Layer</th>
<th>VGG-A</th>
<th>VGG-B</th>
<th>VGG-C</th>
<th>VGG-D</th>
<th>VGG-E</th>
</tr>
</thead>
<tbody>
<tr>
<td>conv11</td>
<td>1024</td>
<td>1024</td>
<td>1024</td>
<td>1024</td>
<td>1024</td>
</tr>
<tr>
<td>conv12</td>
<td>-</td>
<td>1024</td>
<td>1024</td>
<td>1024</td>
<td>1024</td>
</tr>
<tr>
<td>conv21</td>
<td>256</td>
<td>256</td>
<td>256</td>
<td>256</td>
<td>256</td>
</tr>
<tr>
<td>conv22</td>
<td>-</td>
<td>256</td>
<td>256</td>
<td>256</td>
<td>256</td>
</tr>
<tr>
<td>conv31</td>
<td>64</td>
<td>64</td>
<td>64</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>conv32</td>
<td>64</td>
<td>64</td>
<td>64</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>conv33</td>
<td>-</td>
<td>-</td>
<td>64</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>conv34</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>64</td>
</tr>
<tr>
<td>conv41</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>conv42</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>conv43</td>
<td>-</td>
<td>-</td>
<td>16</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>conv44</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>16</td>
</tr>
<tr>
<td>conv51</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>conv52</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>conv53</td>
<td>-</td>
<td>-</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>conv54</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>4</td>
</tr>
</tbody>
</table>

In Figure 3.18, $\lambda = 0$ means the parallelism granularity of all layers are $G = 1$ and $\lambda = \infty$ means $G$ is set to the maximum value for each layer. Figure 3.18 shows that the speedup (compared with GPU) increase monotonically with $\lambda$. However, the increase of parallelism granularity will also cause the increase in area, as shown in Figure 3.19. Therefore, choosing the suitable parallelism granularity to explore the balance between
Figure 3.18: Speedups vs. parallelism granularity.

speedup and area is critical. We choose the balanced parallelism granularity for the networks as the default values shown in Table 3.5.

Figure 3.19: Area vs. parallelism granularity.

3.5.6 Computation Efficiency Results

The area of PIPELAYER is 82.63mm$^2$. The computational efficiency of PIPELAYER is 1485GOPS/s/mm$^2$, higher than both DaDianNao [27] (63.46GOPS/s/mm$^2$) and ISAAC [89] (479.0GOPS/s/mm$^2$). On the other side, the power efficiency of PIPELAYER is 142.9GPOS/s/W, lower than DaDianNao (286.4GPOS/s/W) and ISAAC (380.7GPOS/s/W). In training phase, intermediate data $d$ are used as the kernels to compute partial derivatives. As discussed in section 3.3.4, $d$ is written to morphable subarrays for the convenience of
partial derivatives computing, which means the groups of "storing" arrays are converted to "computing" arrays. For this reason, PIPELAYER gains a higher computational efficiency. However, the lower power efficiency is because we write all of data to ReRAM arrays, while DaDianNao and ISAAC write to eDRAMs.
Chapter 4

Accelerator Architecture for Graph Processing

4.1 Background and Motivation

4.1.1 Graph Processing

![Graph processing in vertex-centric programs.](image1)

**Figure 4.1**: Graph processing in vertex-centric programs.

![Edge-centric processing and dual sliding windows.](image2)

**Figure 4.2**: (a) Edge-centric processing and (b) dual sliding windows.

Graph algorithms traverse vertices and edges to discover interesting graph properties based on relationship. A graph could be naturally considered as an adjacency matrix, where the rows correspond to the vertices and the matrix elements represent the edges. Most graph algorithms can be mapped to matrix operations. However, in reality, the graph is sparse, which means that there would be many zeros in the adjacency matrix. This property incurs the waste of both storage and compute resources. Therefore, the current graph processing systems use the format that is suitable for sparse graph data. Based on such data structures,
the graph processing can be essentially considered as implementing the matrix operations on the sparse matrix representation. In this case, individual (and simple) operations in the whole matrix computation are performed by the compute units (e.g., a core in CPU or an SM in GPU) after data is fetched. In the following, we elaborate the challenge of random accesses in various graph processing approaches. More details on sparse graph data representation will be discussed in Section 4.1.3.

To provide an easy programming interface, the vertex-centric program featuring “Think Like a Vertex (TLAV)” [157] was proposed as a natural and intuitive way for human brain to think of the graph problems. Figure 4.1 (a) shows an example, an algorithm could first access and process the red vertex in the center with all its neighbors through the red edges. Then it can move to one of the neighbors, the blue vertex on the top, accessing the vertex and the neighbors through the blue edges. After that, the algorithm can access another vertex (the green one on the left), which is one of the neighbors of the blue vertex.

In graph processing, the vertex accesses lead to random accesses because the neighbor vertices are not stored continuously. For edges, they incur local sequential access because the edges related to a vertex are stored continuously but global random accesses because algorithm needs to access the edges of different vertices. The concepts are shown in Figure 4.1 (b). The random accesses lead to long memory latency and, more importantly, the bandwidth waste, because only a small portion of data are accessed in a memory block. In a conventional hierarchical memory system, this leads to the significant data movements.

Clearly, reducing random accesses is critical to improve the performance of graph processing, this is particularly crucial for the disk-based single machine graph processing systems (a.k.a out-of-core systems [151–154]), because the random disk I/O operations are much more detrimental to the performance. In this context, the memory is considered small and fast (therefore can afford random accesses) while disk is considered large and slow (therefore should be only accessed sequentially). The edge-centric model in X-Stream [153]
is a notable solution for reducing random accesses. Specifically, the edges of a graph partition are stored and accessed sequentially in disk and the related vertices are stored and accessed randomly in memory. Such setting is feasible because typically the vertex data are much smaller than edge data. During process, X-Stream generates Updates in scatter phase, which incurs sequential writes, and then, applies these Updates to vertices in gather phase, which incurs sequential reads. The concepts are shown in Figure 4.2 (a).

A notable drawback of X-stream is that the number of updates may be as large as that of edges, GridGraph [151] proposed optimizations to further reduce the storage cost and data movements due to updates. The solution is based on the dual sliding windows (shown in Figure 4.2 (b)), which partitions edges into blocks and vertices into chunks. On visiting the edge blocks, the source vertex chunk (orange) is accessed and updates are directly applied to the destination vertex chunk (blue). This requires no temporary update storage as in X-Stream. Edge blocks can be accessed in source oriented order (shown in Figure 4.2 (b)) or destination oriented order. Note that the dual sliding window mechanism is based on edge-centric model.

4.1.2 Graph Processing Accelerators

Due to the wide applications of graph processing and its challenges, several hardware accelerators were recently proposed. Ahn et al. [187] proposes TESSERACT, the first PIM-based graph processing architecture. It defines a generic communication interface to map graph processing to HMC. At any time, each core can Put a remote memory access and get interrupted to receive and execute the Put calls from other cores. Ozdal et al. [186] introduces an accelerator for asynchronous graph processing, which features efficient hardware scheduling and dependence tracking. To use the system, programmers have to understand its architecture and modify existing code. Graphicionado [185] is a customized graph accelerator designed for high performance and energy efficiency based
on off-chip DRAM and on-chip eDRAM. It modifies the graph data structure and data path to optimize graph access patterns and designs specialized memory subsystem for higher bandwidth. These accelerators all optimize the memory accesses, reducing the latency or better tolerating the random access latency. [265] and [266] focused on the data coherence in memory which can be accessed by instructions from host CPU and the accelerator, and [266] also considered atomic operations. Other PIM-based accelerators for graph processing are [267, 268].

4.1.3 Graph Representation

The processing of the matrix-vector multiplications on small data blocks could increase the ratio between computation and data movement and reduce the pressure on memory system. With its matrix-vector multiplication capability, ReRAM could naturally perform the low-cost parallel dense operations on the sparse sub-matrices (subgraphs), enjoying the benefits without increasing hardware and energy cost.

The key insight of GRAPHR is to still store the majority of the graph data in the compressed sparse matrix representation and process the subgraphs in uncompressed sparse matrix representation. In the following, we review several commonly used sparse representations.

\[ \begin{array}{cccc} 0 & 0 & 3 & 8 \\
1 & 0 & 0 & 7 \\
2 & 1 & 0 & 0 \\
3 & 0 & 4 & 0 \\
0 & 1 & 2 \\
\end{array} \begin{array}{cccc} (row, val) & colptr & (col, val) & rowptr \\
(2,1) & 0 & (2,3) & 0 \\
(3,4) & 1 & (3,8) & 1 \\
(0,3) & 2 & (2,7) & 2 \\
(1,7) & 3 & (0,1) & 3 \\
(0,8) & 4 & (1,4) & 4 \\
(3,2) & 5 & (3,2) & 5 \\
\end{array} \begin{array}{c} (row, col, val) \\
(0,2,3) \\
(0,3,8) \\
(1,2,7) \\
(2,0,1) \\
(3,1,4) \\
(3,3,2) \\
\end{array} \]

**Figure 4.3:** (a) Sparse matrix and its compressed representations in: (b) CSC, (c) CSR, and (d) COO.

The three major compressed sparse representations are compressed sparse column
(CSC), compressed sparse row (CSR) and coordinate list (COO). They are illustrated in Figure 4.3. In the CSC representation, non-zeros are stored in column major order as (row index, value) pairs in a list, the number of entries in the list is the number of non-zeros. Another list of column starting pointers indicate the starting index of a row in the (row,val) list. For example, in Figure 4.3 (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 the number of columns + 1. Compressed sparse row (CSR) is similar to CSC, with row and column alternated. For coordinate list (COO), each entry is a tuple of (row index, column index, value) of the nonzeros.

![Figure 4.4](image)

**Figure 4.4:** (a) A directed graph and its representations in (b) adjacency matrix and (c) coordinate list.

In GRAPHR, we assume coordinate list (COO) representation to store a graph. Given a graph in Figure 4.4 ¹ (a), its (sparse) adjacency matrix representation and COO representation (partitioned into four 4 × 4 subgraphs) are shown in Figure 4.4(b) and (c), respectively. In this example, the coordinate list saves 61% storing space, compared with the adjacency matrix representation. For real-world graphs with high sparsity, the saving is even high: the coordinate list can only take 0.2% of space for WikiVote [269] compared to an adjacency

---

¹As shown in Section 4.1.3, each entry in a coordinate list should be a three-element tuple of {Source ID, Destination ID, Edge Weight}. To simplify the example, we use an unweighted graph, so a two-element tuple of {Source ID, Destination ID} is sufficient to represent one edge.
matrix.

4.2 GraphR Architecture

4.2.1 Sparse Matrix Vector Multiplication (SpMV) and Graph Processing

```plaintext
//Phase 1: compute edge values
for each edge E(V,U) from active vertex V do
    E.value = processEdge(E.weight, V.prop)
end for

//Phase 2: reduce and apply
for each edge E(U,V) to vertex V do
    V.prop = reduce(V.prop, E.value)
end for
```

Figure 4.5: Vertex Programming Model

Figure 4.5 shows the general vertex programming model for graph processing. It follows the principle of “Think Like The Vertex” [157]. In each iteration, the computation can be divided into two phases. In the first phase, each edge is processed with `processEdge` function, based on edge weight and active source vertex V’s property ($V.prop$), it computes a value for each edge. In the second phase, each vertex reduces values of all incoming edges ($E.value$) and generates a property value, which is applied to the vertex as its updated property.

Figure 4.6 (a) shows vertex program in graph view. $V_{15}$, $V_7$, $V_1$ and $V_2$ each executes a `processEdge` function, the results are stored in corresponding edges to $V_4$. After all edges are processed, $V_4$ executes reduce function to generate the new property and update itself. In graph processing, the operation in `processEdge` function is multiplication, to generate the new property for each vertex, what essentially needed is the Multiply-Accumulate (MAC) operation. Moreover, the vertex program of each vertex is equivalent to a sparse

75
Figure 4.6: GRAPHR Key insight: supporting graph processing with ReRAM crossbars.

matrix vector multiplication (SpMV) shown in Figure 4.6 (b). Assume $A$ is the sparse adjacency matrix, and $x$ is a vector containing $V$.prop of all vertices, the vertex program of all vertices can be computed in parallel in matrix view as $A^T x$. As shown in Figure 3.3 (c), ReRAM crossbar could perform matrix-vector multiplication efficiently. Therefore, as long as a vertex program can be expressed in SpMV form, it can be accelerated by our approach. We will discuss in Section 4.3 on the different patterns to express algorithms in SpMV.

From Figure 4.6 (b), we see that the size of the matrix and vector are $|V| \times |V|$ and $|V|$, respectively, where $V$ is the number of vertices in the graph. In ideal case, if we are given a ReRAM crossbar (CB) of size $|V| \times |V|$, the vertex program of each vertex can be executed in parallel and the new value (i.e., $V$.prop in Figure 4.5) can be computed in one
cycle. More importantly, the memory to store $A^T$ can directly perform such in-memory computation without data movement. Unfortunately, this is unrealistic, the size of CB is quite limited (e.g., $4 \times 4$ or $8 \times 8$), to realize this idea, we need to use Graph Engines (GEs) composed of small CBs to process subgraphs in certain manner. This is the ultimate design goal of GRAPHR architecture. We face several key problems. First, as discussed in Section 4.1.3, graph is stored in compressed format, not in adjacency matrix, to perform in-memory computation, data needs to be converted to matrix format. Second, the real-world large graphs may not fit in memory. Third, the order of subgraph processing needs to be carefully determined, because this affects the hardware cost to buffer the temporary results for reduction.

To solve these problems, Figure 4.6 (d) shows the high level processing procedure. The GRAPHR architecture has both memory and compute module, each time, a block of a large graph is loaded into GRAPHR’s memory module (in blue) in compressed sparse representation. If the memory module can fit the whole graph, then there is only one block, otherwise, a graph is partitioned into several blocks. Inside GRAPHR, a number of GEs (in orange), each of which is composed of a number of CBs, “scan” and process (similar to a sliding window) subgraphs in streaming-apply model (Section 4.2.3). To enable in-memory computation, graph data are converted to matrix format by a controller. Such conversion is straightforward with preprocessed edge list (Section 4.2.4). The intermediate results of subgraphs are stored in buffers and simple computation logics are used to perform reduction.

GRAPHR can be used in two settings: 1) multi-node: one can connect different GRAPHR nodes and connect them together to process large graphs. In this case, each block is processed by a GRAPHR node. Data movements happen between GRAPHR nodes. 2) out-of-core: one can use one GRAPHR node as an in-memory graph processing accelerator to avoid using multiple threads and moving data through memory hierarchy. In this case, all blocks are processed consecutively by a single GRAPHR node. Data movements happen
between disk and GRAPHR node. In this work, we assume out-of-core setting, so we do not consider communication between nodes and its supports, we leave this as future work and extension. Next, we present the overall GRAPHR architecture and its workflow.

### 4.2.2 GraphR Architecture

Figure 4.7 shows the architecture of one GRAPHR node. GRAPHR is a ReRAM-based memory module that performs efficient near data parallel graph processing. It contains two key components: memory ReRAM and graph engine (GE). Memory ReRAM stores graph data in original compressed sparse representation. GEs perform efficient matrix-vector multiplications on matrix representation. A GE contains a number of ReRAM crossbars (CBs), Drivers (DRVs), Sample and Hold (S/H) components placed in mesh, 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 register (RegI/RegO) are used to cache data flow. We discuss the detail of several components as follows.

Driver (DRV) is used to 1) load new edge data to ReRAM crossbars for processing; and 2) input data into ReRAM crossbars for matrix-vector multiplication.

Sample and Hold (S/H) is used to sample analog values and hold them before converting...
to a digital form.

Analog to Digital Converter (ADC) converts analog values to digital format. Because ADCs have relatively higher area and power consumption, ADCs are not connected to every bitlines of ReRAM crossbars in a GE but shared between those bitlines. If the GE cycle is 64\(ns\), we can have one ADC working at 1.0\(GSps\) to convert all data from eight 8-bitline crossbars within one GE.

sALU is a simple customized algorithmic and logic unit. sALU performs operations that cannot be efficiently performed by ReRAM crossbars, such as comparison. The actual operation performed by sALU depends on algorithm and can be configured. We will show more details when we discuss algorithms in Section 4.3.

Data Format is not practical for ReRAM cells to support a high resolution. Recent work [256] reported 5-bit resolution on ReRAM programing. To alleviate the pressure of diver, we conservatively assume the 4-bit ReRAM cell. To support higher computing resolution, e.g., 16 bit, the Shift and Add (S/A) unit is employed. A 16-bit fixed point number \(M\) can be represented as \(M = [M_3, M_2, M_1, M_0]\), where each segment \(M_i\) is a 4-bit number. We can shift and add results from four 4-bit-resolution ReRAM crossbars, i.e. \(D_3 \ll 12 + D_2 \ll 8 + D_1 \ll 4 + D_0\) to get a 16-bit result.

When sALU and S/A are bypassed, a graph engine could be simply considered as a memory ReRAM mat. A similar scheme of reusing ReRAM crossbars for computing and storing is employed in [88].

The I/O interface is used to load graph data and instructions into memory ReRAM and controller, respectively. In GRAPHR, controller is the software/hardware interface that could execute simple instructions to: 1) coordinate graph data movements between memory ReRAM and GEs based on streaming-apply execution model; 2) convert edges in preprocessed coordinate list (assumed in this work but can work with other representations as well) in memory ReRAM to sparse matrix format in GEs; 3) perform convergence check.
Figure 4.8: Workflow of \textsc{GraphR} in out-of-core setting.

```
while(true)
  load edges for next subgraph into GEs;
  process (\texttt{processEdge}) in GE;
  reduce by sALU;
  if(check_convergence())
    break;
}
```

Figure 4.9: Controller operation.

Figure 4.8 shows the workflow of \textsc{GraphR} in an out-of-core setting. \textsc{GraphR} node can be used as a drop-in in-memory graph processing accelerator. The loading of each block is performed in software by an out-of-core graph processing framework (e.g., Grid-Graph [151]). The integration is easy because it already contains the codes to load block from disk to DRAM, we just need to redirect data to \textsc{GraphR} node. Since edge list is preprocessed in certain order, loading each block only involves sequential disk I/O. Inside \textsc{GraphR} node, the data is initial loaded into memory ReRAM, the controller manages the data movements between GEs in streaming-apply manner. Edge list preprocessing for \textsc{GraphR} needs to be carefully designed and it is based on the architectural parameters.
Figure 4.9 shows the operations performed by controller.

### 4.2.3 Streaming-Apply Execution Model

![Streaming-Apply Execution Model Diagram](image)

**Figure 4.10:** Streaming-apply execution model.

In **GRAPH**, all GEs process a subgraph, the order of processing is important, since it affects hardware resource requirement. We present streaming-apply execution model shown in Figure 4.10. The key insight is that subgraphs are processed in a streaming manner and the reductions are performed on the fly by sALU. There are two variants of this model: column-major and row-major. During execution, RegI and RegO are used to store source vertices and updated destination vertices. In column-major order in Figure 4.10 (a), subgraphs with same destination vertices are processed together. The required RegO size is the same as the number of destination vertices in a subgraph. In row-major order in Figure 4.10 (b), subgraphs with same source vertices are process together. The required RegO size is the total number of destination vertices of all subgraphs with the same source vertices. It is easy to see that row-major order requires larger RegO. On the other side, row-major order incurs less read of RegI, — only one read is needed for all subgraphs with same source vertices. In **GRAPH**, we use column-major order since it requires less registers, and in ReRAM, the write cost is higher than read cost.

Figure 4.10 (c) shows the global processing order in a complete out-of-core setting. We...
see that the whole graph is partitioned into 4 blocks, three of them are stored in disk in COO representation, one is stored in GraphR’s memory ReRAM also in COO representation. Only the subgraph being processed by GEs is in sparse matrix format. Due to the sparsity of graph data, if the subgraph is empty, then GEs can move down to the next subgraph. Therefore, the sparsity only incurs waste inside the subgraph (e.g., when one GE has an empty matrix but others do not).

### 4.2.4 Graph Preprocessing

To support streaming-apply execution model and conversion from coordination list representation to sparse matrix format, edge list needs to be preprocessed so that edges of consecutive subgraphs are stored together. It also ensures sequential disk and memory access on block/subgraph loading. In the following, we explain the preprocessing procedure in detail.

**Figure 4.11**: Preprocessing edge list.
Given some architectural parameters, the preprocessing is implemented in software and performed only once. Figure 4.2.4 illustrates these parameters and the subgraph order we should produce. This order is determined by streaming-apply model. $C$ is the size of CB; $N$ is the number of CBs in a GE; $G$ is the number of GEs in a GRAPHR node; and $B$ is the number of vertices in a block (i.e., block size). Also, $V$ is the number of vertices in the graph. Among the parameters, $C$, $N$ and $G$ specify the architecture setting of a GRAPHR node, $B$ is determined by the size of memory ReRAM in the node. In the example, the graph has 64 vertices ($V = 64$) and each block has 32 vertices ($B = 32$), so the graph is partitioned into 4 blocks, each one will be loaded from disk to GRAPHR node. Further, $C = 4, N = 2, G = 2$, so the subgraph size is $C \times (C \times N \times G) = 4 \times (4 \times 2 \times 2) = 4 \times 16$. Therefore, each block is partitioned into 16 subgraphs, the number of each is the global subgraph order. Our goal is to produce an ordered edge list according this order so that edges could be loaded sequentially.

Before presenting the algorithm, we assume that the original edge list is first ordered based on source vertex and then for all edges with the same source, they are ordered by destination. In another word, edges are stored in row-major order in matrix view. We also assume that in the ordered edge list, edges in a subgraph is stored in column-major order. Considering the problem in matrix view, each edge $(i, j)$, we first compute a global order ID ($I(i, j)$), so we have a 3-tuple: $(i, j, I(i, j))$. This global order ID takes all zeros into account, for example, if there are $k$ zeros between two edges in the global order, the different of global order ID of them is still $k$. Then, all 3-tuples could be sorted by $I(i, j)$. The ordered edge list is obtained if we output them according to this order. The space and time complexity of this procedure are $O(V)$ and $O(V \log V)$, respectively, with common method. The key problem here is to compute $I(i, j)$ for each $(i, j)$. Without confusion, we simply denote $I(i, j)$ as $I$. We show that $I$ can be computed in a hierarchical manner.

Let $B_T$ denote the global block order of the block that contains $(i, j)$. We assume that
blocks are also processed in column-major order: $B_{(0,0)} \rightarrow B_{(1,0)} \rightarrow B_{(0,1)} \rightarrow B_{(1,1)}$. The coordinates of a block $B$ are:

$$B_i = \left\lfloor \frac{i}{B} \right\rfloor, B_j = \left\lfloor \frac{j}{B} \right\rfloor$$  \hspace{1cm} (4.1)

Based on column-major order, $B_I$ is:

$$I_B = B_j + (V/B) \times B_j$$  \hspace{1cm} (4.2)

We assume that $B$ can divide $V$, and similarly, $C$ can divide $B$, $C \times N \times B$ can divide $B$. Otherwise, we can simply pad zeros to satisfy the conditions, it will not affect the results since these zeros do not correspond to actual edges. The block corresponding to $B_I$ start with the following global order ID:

$$start\_global\_ID(B_I) = B_I \times B^2/(C^2 \times N \times G) + 1$$  \hspace{1cm} (4.3)

Next, we compute $SI$, $(i, j)$’s global subgraph ID. The coordinates of the edge from the start of a block $B_I (B_i, B_j)$ are:

$$i' = i - B_i \times B, j' = j - B_j \times B$$  \hspace{1cm} (4.4)

The relative coordinates of the subgraph from the start of correspond block are:

$$SI_{i'} = \left\lfloor \frac{i'}{C} \right\rfloor, SI_{j'} = \left\lfloor \frac{j'}{C \times N \times G} \right\rfloor$$  \hspace{1cm} (4.5)

Then, we can compute $SI$:

$$SI = (S_{i'} + S_{j'} \times B/C) + start\_global\_ID(B_I)$$

$$= (S_{i'} + S_{j'} \times B/C) + B_I \times B^2/(C^2 \times N \times G) + 1$$  \hspace{1cm} (4.6)

Finally, we compute $SubI$, the relative order of $(i, j)$ from its corresponding subgraph $(SI)$. The coordinates are:

$$SubI_i = i - (B \times B_i) - (S_i \times C),$$  \hspace{1cm} (4.7)

$$SubI_j = j - (B \times B_j) - (S_j \times C)$$
Since edges in a subgraph are assumed to be stored in column-major order, $SubI$ is:

$$SubI = SubI_i + (SubI_j - 1) \times C$$  \hspace{1cm} (4.8)

With $SI$ and $SubI$ computed, we get $I$:

$$I = (SI - 1) \times (C^2 \times N \times G) + SubI$$  \hspace{1cm} (4.9)

### 4.2.5 Discussion

```plaintext
//Phase 1: compute edge values
for each edge E(V,U) from active vertex V do
    E.value = r * V.prop / V.outdegree
end for

//Phase 2: reduce and apply
for each edge E(U,V) to vertex V do
    V.prop = sum(E.value) + (1-r) / Num_Verex
end for

Figure 4.12: PageRank in vertex programming.
```

```plaintext
//Phase 1: compute edge values
for each edge E(V,U) from active vertex V do
    E.value = E.weight + V.prop
    activate(U);
end for

//Phase 2: reduce and apply
for each edge E(U,V) to vertex V do
    V.prop = min(V.prop,E.value)
end for

Figure 4.13: SSSP in vertex programming.
```

Table 4.1 compares different architectures for graph processing. GRAPHR improves over previous architectures due to two unique features. First, the computation is performed in analog manner, others use either instructions or specialized digital processing units.
This provides the excellent energy efficiency. Second, all disk and memory accesses in \textsc{GraphR} are sequential, this is due to preprocessing and less flexibility in scheduling vertices. It is a good trade-off because it is highly energy efficient to perform parallel operations in ReRAM CBs. We believe that \textsc{GraphR} is the first architecture scheme using ReRAM CBs to perform graph processing, and the work presents detailed and complete solution to integrate it as a drop-in accelerator for out-of-core graph processing systems. The architecture, streaming-apply execution model and the preprocessing algorithms are all novel contributions. Also, \textsc{GraphR} is general because it could accelerate all vertex programs that can be performed in SpMV form.

### 4.3 Mapping Algorithms in GE

In this section, we discuss two patterns when mapping algorithms to GEs: parallel MAC and parallel add-op. We use a typical example for each category (i.e., PageRank and SSSP,
respectively) to explain the insights. More examples (but not all) of supported algorithms in GRAPHR are listed in Table 4.2. The first two are parallel MAC pattern and the last two are parallel add-op pattern.

4.3.1 Parallel MAC

In an algorithm, if processEdge function performs a multiplication, which can be performed in each CB cell, we call it parallel MAC pattern. The parallelization degree is roughly \((C \times C \times N \times G)\) (see parameter in Figure 4.8).

PageRank [128] is an excellent example of this pattern. Figure 4.12 shows its vertex program. It does the following iterative computation: \(\overrightarrow{PR}_{t+1} = r\overrightarrow{M} \cdot \overrightarrow{PR}_t + (1 - r) \overrightarrow{e} \). \(\overrightarrow{PR}_t\) is the PageRank at iteration \(t\), \(\overrightarrow{M}\) is a probability transfer matrix, \(r\) is the probability of random surfing and \(\overrightarrow{e}\) is a vector of probabilities of staying in each page.

We consider a small subgraph that could be processed by a 5 × 4 CB (the additional row is to perform addition). It contains at most 16 edges represented by the intersection of 4 rows and 4 columns in the sparse matrix (shown in Figure 4.15 (a)). Thus, the block is related to 8 vertices (i.e., \(i0 \sim i3, j0 \sim j3\)). The following are the parameters for the 4 × 4 block PageRank shown in Figure 4.15 (b1): \(\overrightarrow{M} = [0, 1/2, 1, 0; 1/3, 0, 0, 1/2; 1/3, 0, 0, 1/2; 1/3, 1/2, 0, 0]\), \(\overrightarrow{e} = [1/4, 1/4, 1/4, 1/4]^T\), \(r = 4/5\).

We define \(\overrightarrow{M}_0 = r\overrightarrow{M}\) and \(\overrightarrow{e}_0 = (1 - r) \overrightarrow{e}\), so that \(\overrightarrow{PR}_{t+1} = \overrightarrow{M}_0 \overrightarrow{PR}_t + \overrightarrow{e}_0\). In CB in Figure 4.15 (b2, b3), the values are already scaled with \(r\). Figure 4.15 (b3) shows the mapping of PageRank algorithm to a CB. The additional row is used to implement the addition of \(\overrightarrow{e}_0\). The sALU is configured to perform add operation in the reduce function to add PageRank values (Figure 4.14 (a)). To check convergence, the new PageRank value is compared with it in the previous iteration, the algorithm is converged if the difference is less than a threshold.

87
4.3.2 Parallel Add-Op

In an algorithm, if processEdge function performs an addition, which can be performed in for each CB row at a time, we call it parallel add-op pattern. The op specifies the operation in reduce that is implemented in sALU. The parallelization degree is roughly \( (C \times N \times G) \).

Single Source Shortest Path (SSSP) [270] is a typical example of this pattern. Figure 4.13 shows the vertex program. We see that processEdge performs an addition and reduce performs min operation. Therefore, sALU is configured to perform min operation shown in Figure 4.14 (b).

In SSSP, each vertex \( v \) is given a distance label \( dist(v) \) that maintains the length of the shortest known path to vertex \( v \) from the source. The distance label is initialized to 0 at the source and \( \infty \) at all other nodes. Then the SSSP algorithm iteratively applies the relaxation operator, which is defined as follows: if \( (u,v) \) is an edge and \( dist(u) + w(u,v) < dist(v) \), the value of \( dist(v) \) is updated to \( dist(u) + w(u,v) \). An active vertex is relaxed by applying this operator to all the edges connected to it. Each relaxation may lower the distance label of some vertex, and when no further lowering can be performed anywhere in the graph, the resulting vertex distance labels indicate the shortest distances from the source to the vertex, regardless of the order in which the relaxations were performed. Breadth-first numbering of a graph is a special case of SSSP where all edge labels are 1.

We explain the mapping of SSSP algorithm to a CB using a small 8-vertex subgraph.
corresponding to a 4 × 4 block in sparse matrix, as shown in Figure 4.15 (c1). It can be represented by adjacency matrix: \( W = [M, 1, 5, M; M, M, 3, 1; M, M, M; M, M, 1, M] \) where \( M \) indicates no edge connected two vertices and \( M \) is set to a reserved maximum value for a memory cell in a CB. The values are stored in the CB shown in Figure 4.15 (c2).

Given a vertex \( u \) and \( \text{dist}(u) \), the row in the adjacency matrix \( W \) for \( u \) indicates \( w(u, v) \). We could perform the relaxation operator (i.e., addition) for \( u \) in parallel. Here, SpMV is only used to select a row in CB by multiplying with an one-hot vector.

The relaxation operator of \( u \) involves reading: 1) \( \text{dist}(u) \): it is computed iteratively from the source, for the source vertex, the initial value is zero; 2) The vector of the \( \text{dist}(v) \) before the relaxation operator: it is a vector indicating the distance between source and all other vertices and is also computed iteratively from the source. In our example, for the source vertices \((i0, i1, i2, i3)\), the initial value is \([4, 3, 1, 2]\); 3) The vector of the \( w(u, v) \): it is the

---

Figure 4.15: The processing of (a) PageRank and (b) SSSP in GRAPHR
distance from \( u \) to the destination vertices in the subgraph, and can be obtained by reading a row in adjacency matrix \( \mathbf{W} \).

Figure 4.15 (c3) shows the process to perform SSSP in a \( 5 \times 4 \) CB. The last row (green squares) is set to a fixed value 1, which is used to add \( \text{dist}(u) \) (the input to the last wordline) to each \( w(u,v) \) in the relaxation operator. The initial value for \( \text{dist}(u) \) for the destination vertices \( (j_0, j_1, j_2, j_3) \) are \([7, 6, M, M]\). The vector of \( w(u,v) \) is obtained by activating the wordline associated to vertex \( v \). In the time slot \( (t = 1) \), wordline 0 (for source vertex \( i_0 \)) is activated (the red square with input value 1) and a value 4 (this is the current value in \( \text{dist}(v) \) for source \( i_0 \)) is input to the last wordline (the green box line). The vector of the \( \text{dist}(v) \) for the destination vertices \( ((j_0, j_1, j_2, j_3)) \) is set as the value of the output at each bitline, which is \([7, 6, M, M]\). With this mapping, the current summation in bitline in Figure 4.15 (c3-1) is \([1 \times M + 4 \times 1, 1 \times 1 + 4 \times 1, 1 \times 5 + 4 \times 1, 1 \times M + 4 \times 1] = [M, 5, 9, M]\). It is the \( \text{dist}(u) + w(u,v) \) computed in parallel, where \( u \) is the source vertex. Then the distance of source to each vertex \( v \) is compared with the initial \( \text{dist}(v) \) \(([7, 6, M, M])\) by an array of comparators, and in the final output of bitline, we get \([7, 5, 9, M]\), which is the updated \( \text{dist}(v) \) vector after time slot \( t = 1 \). The parallel comparisons are performed by vertex-related operations in. Different algorithms may require different functions on vertices.

<table>
<thead>
<tr>
<th>Applications</th>
<th>Vertex Property</th>
<th>processEdge()</th>
</tr>
</thead>
<tbody>
<tr>
<td>SpMV</td>
<td>Multiplication Value</td>
<td>( \text{E.value} = \text{V.prop} / \text{V.outdegree} * \text{E.weight} )</td>
</tr>
<tr>
<td>PageRank</td>
<td>Page Rank Value</td>
<td>( \text{E.value} = \text{r} \times \text{V.prop} / \text{V.outdegree} )</td>
</tr>
<tr>
<td>BFS</td>
<td>Level</td>
<td>( \text{E.value} = 1 + \text{V.prop} )</td>
</tr>
<tr>
<td>SSSP</td>
<td>Path Length</td>
<td>( \text{E.value} = \text{E.weight} + \text{V.prop} )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Applications</th>
<th>reduce()</th>
<th>Active Vertex List</th>
</tr>
</thead>
<tbody>
<tr>
<td>SpMV</td>
<td>( \text{V.prop} = \text{sum(E.value)} )</td>
<td>Not Required</td>
</tr>
<tr>
<td>PageRank</td>
<td>( \text{V.prop} = \text{sum(E.value)} + (1-\text{r}) / \text{Num.Vertex} )</td>
<td>Not Required</td>
</tr>
<tr>
<td>BFS</td>
<td>( \text{V.prop} = \text{min(V.prop, E.value)} )</td>
<td>Required</td>
</tr>
<tr>
<td>SSSP</td>
<td>( \text{V.prop} = \text{min(V.prop, E.value)} )</td>
<td>Required</td>
</tr>
</tbody>
</table>

After time slot \( t = 1 \), we move to the next vertex in Figure 4.15 (c3-2), where we i)
activate the second wordline; ii) change the input to the last wordline to 3 (the distance label for source vertex i1); and iii) set the intermediate \( \text{dist}(v) \) to be the final output of bitline in time slot \( t = 1 \), which is \([7, 5, 9, M]\). Similar as Figure 4.15 (c3-1), the the current summation in bitline is \([M, M, 6, 4]\) and it is compared with \([7, 5, 9, M]\), yielding the final output of bitline for time slot \( t = 2 \) as \([7, 5, 6, 4]\). We indicate the updated distance label using orange squares. The operations in time slot \( t = 3 \) and \( t = 4 \) can be performed in the similar manner.

Initially, before processing the block, the active indicator for each destination vertex is set to be FALSE. After the serial processing in CB, the active indicators for all vertices which have been updated (marked in orange in Figure 4.15 (c3)) are set to be TRUE. This indicates that they are active for next iteration. In our example, \( j_1, j_2, j_3 \) are marked active. Referring to Figure 4.15, this means that the distance labels for these vertices have been updated. Because we are processing the block in CB, the active indicator for destination vertex may be accessed for multiple times, but if it is set be TRUE at least one time, this vertex is active in next iteration. Globally, after all active source vertices and corresponding edges are processed in an iteration, source vertex properties (values and active indicators) that hold the old values are updated by the properties of the same vertices in destination. The new source vertex properties are used in the next iteration. We can check if there are still active vertices to determine the convergence.

### 4.4 Evaluation

#### 4.4.1 Graph Datasets and Applications

Table 4.3 shows the datasets used in our evaluation. We use seven real-world graphs. For WikiVote(WV), Slashdot(SD), Amazon(AZ), WebGoogle(WG), LiveJournal(LJ), Orkut(OK) and Netflix(NF). We run pagerank(PR), breadth first search(BFS), single source shortest path (SSSP) and sparse matrix-vector multiplication (SpMV) on the first six datasets. On
Table 4.3: Graph datasets.

<table>
<thead>
<tr>
<th>Dataset</th>
<th># Vertices</th>
<th>#Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>WikiVote (WV) [269]</td>
<td>7.0K</td>
<td>103K</td>
</tr>
<tr>
<td>Slashdot (SD) [269]</td>
<td>82K</td>
<td>948K</td>
</tr>
<tr>
<td>Amazon (AZ) [269]</td>
<td>262K</td>
<td>1.2M</td>
</tr>
<tr>
<td>WebGoogle (WG) [269]</td>
<td>0.88M</td>
<td>5.1M</td>
</tr>
<tr>
<td>LiveJournal (LJ) [269]</td>
<td>4.8M</td>
<td>69M</td>
</tr>
<tr>
<td>Orkut (OK) [271]</td>
<td>3.0M</td>
<td>106M</td>
</tr>
<tr>
<td>Netflix (NF) [272]</td>
<td>480K users, 17.8K movies</td>
<td>99M</td>
</tr>
</tbody>
</table>

Netflix (NF), we run collaborative filtering (CF), and the feature length used is 32.

4.4.2 Experiment Setup

In our experiments, we compare GRAPHR with a CPU baseline platform, a GPU platform and PIM-based architecture [187]. PR, BFS, SSSP and SpMV running on the CPU platform are based on the software framework GridGraph [151], while collaborative filtering is based on GraphChi [152]. PR, BFS, SSSP and SpMV running on GPU platform are based on Gunrock [273], while CuMF_SGD [274] is the GPU framework for CF. We evaluate PIM-based architecture on zSim [275], a scalable x86-64 multicore simulator. We modified zSim with HMC memory and interconnection model, heterogeneous compute units, on-chip network and other hardware features. The results are validated results with NDP [276], which also extends zSim for HMC simulation. In all experiments, graph data could fit in memory. We also exclude the disk I/O time from the execution time of the CPU/GPU-based platform.

Specifications of the CPU and GPU platforms are shown in Table 4.4 and Table 4.5. The CPU energy consumption is estimated by Intel Product Specifications [277] while NVIDIA System Management Interface (nvidia-smi) is used to estimate energy consumption by GPU. The execution times for CPU/GPU platform are measured in the computing frameworks.
Table 4.4: Specifications of the CPU platform.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Intel Xeon E5-2630 V3, 8 cores, 2.40 GHz, 8 × (32 + 32) KB L1 Cache, 8 × 256KB L2 Cache, 20 MB L3 Cache</td>
</tr>
<tr>
<td>Memory</td>
<td>128 GB</td>
</tr>
<tr>
<td>Storage</td>
<td>1 TB</td>
</tr>
<tr>
<td></td>
<td>2 CPUs, a total number of 32 threads.</td>
</tr>
</tbody>
</table>

Table 4.5: Specifications of the GPU platform.

<table>
<thead>
<tr>
<th>Graphic Card</th>
<th>NVIDIA Tesla K40c</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture</td>
<td>Kepler</td>
</tr>
<tr>
<td># CUDA Cores</td>
<td>2880</td>
</tr>
<tr>
<td>Base Clock</td>
<td>745 MHz</td>
</tr>
<tr>
<td>Graphic Memory</td>
<td>12 GB GDDR5</td>
</tr>
<tr>
<td>Memory Bandwidth</td>
<td>288 GB/s</td>
</tr>
<tr>
<td>CUDA Version</td>
<td>7.5</td>
</tr>
</tbody>
</table>

To evaluate GRAPHR, for the ReRAM part, we use NVSim [184] to estimate time and energy consumption. The HRS/LRS resistance are 25M/50K Ω, read voltage (Vr) and write voltage (Vw) are 0.7V and 2V respectively, and current of LRS/HRS are 40 uA and 2 uA respectively. The read/write latency and read/write energy cost used are 29.31ns / 50.88ns, 1.08 pJ / 3.91nJ respectively from data reported in [264]. The programming of a bipolar ReRAM cell is to change (from High to Low) or inverse. For multi-level cell, the programming is to change the resistance to a middle state between High and Low, and the middle state is determined by the programming voltage pulse length. Actually, the difference between High and Low is the worse case. Note that [256,262] describe a ReRAM programming prototype, which includes: 1) writing circuitry; 2) ReRAM cell/array; and 3) conversion circuitry. They demonstrated the possibility of 1% accuracy for multi-level cell. However, in a real production system, only “writing circuitry” and “ReRAM cell/array” are needed, there is no need to consider the conversion, as we just need to “acquiesce” a writing precision. Therefore, this energy cost estimation for 4-bit cell programming is reasonable.
and more conservative than two recent ReRAM-based accelerators [88, 89].

For on-chip registers, we use CACTI 6.5 [278] at 32nm to model. For Analog/Digital converters, we use data from [279]. The system performance is modeled by code instrumentation. The ReRAM crossbar size $S$, number of ReRAM crossbars per graph engine $C$ and number of graph engines is 8, 32, 64 respectively.

### 4.4.3 Performance Results

Figure 4.16 compares the performance of GRAPHR and CPU platform. The CPU implementation is used as the baseline and execution times of applications of GRAPHR are normalized to it. Compared to CPU platform, the geometric mean of speedups with GRAPHR architecture on all 25 executions is $16.01 \times$. Among all applications on the datasets, the highest speedup achieved by GRAPHR is $132.67 \times$, and it happens on SpMV on WikiVote dataset. The lowest speedup GRAPHR achieved is $2.40 \times$, on SSSP using OK dataset. PageRank and SpMV are parallel MAC pattern and have higher speedup compared to CPU-based platform. For BFS and SSSP which are parallel add-op pattern, GRAPHR achieves lower speedups only due to parallel addition.

![Figure 4.16: GRAPHR speedup normalized to CPU platform.](image)

94
4.4.4 Energy Results

Figure 4.17 shows the energy savings in GRAPHR architecture over CPU platform. The geometric mean of energy savings of all applications compared to CPU is $33.82 \times$ . The highest energy achieved by GRAPHR is $217.88 \times$, which is on sparse matrix-vector multiplication on SD dataset. The lowest energy saving achieved by GRAPHR happens on SSSP on OK dataset, which is $4.50 \times$. GRAPHR gets the high energy efficiency from the non-volatile property of ReRAM and the in-situ computation capability.

4.4.5 Comparison to GPU Platform

GPUs take advantage of a large amount of threads (CUDA cores) for high parallelism. The GPU used in the comparison has 2880 CUDA cores, while in GRAPHR we have a comparable number ($2048 = 32 \times 64$) of crossbars. To compare with GPU, we run PageRank and SSSP on LiveJournal dataset, and collaborative filtering (CF) on Netflix dataset.

The performance and energy saving normalized to CPU are shown in Figure 4.18 (a) and (b). Overall, the performance of GRAPHR is higher than GPU with considering the data transfer time between CPU memory and GPU memory, — an overhead GRAPHR does not incur. GRAPHR has performance gains ranging from $1.69 \times$ to $2.19 \times$ compared to GPU. More importantly, GRAPHR consumes $4.77 \times$ to $8.91 \times$ less energy than GPU.
Figure 4.18: GRAPHR (a) performance and (b) energy saving compared to GPU platform.

Figure 4.18 (a) shows that, GRAPHR achieves higher speedups on PageRank and CF, where MACs dominate the computation and are fully supported by GRAPHR and GPU. For SSSP, the vertex-related traversing in GRAPHR requires accessing to main memory and storage. In GPU, a cache based memory hierarchy better supports the random accessing. So the speedup of GRAPHR is lower. The reason why GRAPHR still has gain in SSSP is due to sequential access pattern. For energy saving, GRAPHR is better than GPU. Besides energy saving by in-situ computation in GEs and the in-memory processing system design, from Fig 17 in [185] we see that in conventional CMOS system, static energy consumption by eDRAM (memory) incurs the majority of energy consumption. As the technology node scales down, leakage power dominates in CMOS system. In contrast, ReRAM has almost 0 static energy leakage, so GRAPHR has higher energy saving.

4.4.6 Comparison to PIM Platform

We compare GRAPHR with a PIM-based architecture (i.e., Tesseract [187]). The performance and energy saving normalized to CPU are shown in Figure 4.19 (a) and (b). GRAPHR gains a speedup of $1.16 \times$ to $4.12 \times$, and is $3.67 \times$ to $10.96 \times$ more energy efficiency compared to PIM-based architecture.
4.4.7 Sensitivity to Sparsity

We use \( \frac{\text{#Edge}}{\text{(#Vertex)}^2} \) to represent the density of a dataset, and with the density decreases, the sparsity increases. Figure 4.20 (a) and (b) shows the performance and energy saving of \text{GRAPHR} (compared to the CPU platform) with the density of datasets. With the sparsity increases, the performance and energy saving slightly decreases. Because with the sparsity increases, the number of edge blocks to be traversed will increase, which slows down the edge accessing time and consumes more energy.
Chapter 5

Low-Cost Floating-Point Processing in ReRAM

5.1 Background

5.1.1 MVM Acceleration in ReRAM

Resistive Random Access Memory (ReRAM) [125, 280] has recently demonstrated tremendous potential to efficiently accelerate the computing kernel, i.e., MVM, in machine learning models. Conceptually, each element in a matrix $M$ is mapped to the conductance state of an ReRAM cell, while the input vector $x$ is encoded as voltage levels that are applied on the wordlines of the ReRAM crossbar. In this way, the current accumulation on bitlines is proportional to the dot-product of the stored conductance and voltages on the wordlines, representing the result $y = M \times x$. Such in-situ computation significantly reduces the costly memory access in MVM processing engines [256], and mostly importantly, provides massive opportunities to exploit the inherent parallelism in an $N \times N$ ReRAM crossbar.

ReRAM-based MVM processing engines are fixed-point hardware in nature due to the fact that the matrix and the vector are respectively represented in discrete conductance states and voltage levels [125]. If ReRAM is used to support floating-point MVM operation, a large number of crossbars will be provisioned for mantissa alignment, resulting in very high hardware cost. Fortunately, the fixed-point precision requirement is acceptable for machine learning applications thanks to the low-precision and quantized neural network algorithms [173, 281–285]. Many fixed-point based accelerators [88, 89, 91, 92, 218, 222, 223, 286] are built with the ReRAM MVM processing engine and achieve reasonably good classification accuracy.
Figure 5.1 shows an example of computing fixed-point MVM in Eq. (5.1) by utilizing ReRAM-based MVM engine with single-bit precision. Before computation, decimal
integers in both the matrix and the vector are converted to binary bits, where we set the precision for the matrix and input vector are 4-bit. The matrix is bit-sliced into 4 one-bit matrices and then mapped to four crossbars, i.e., M-b3, M-b2, M-b1 and M-b0 in Figure and the input vector is bit-sliced into 4 one-bit vectors, i.e., V-b3, V-b2, V-b1 and V-b0. The multiplication is performed in pipeline. The multiplication results in each crossbars are initiated by a zero vector S0. In the first cycle C1, the most significant bit (MSB) vector V-b3 is applied on wordlines of the four crossbars, and the multiplication results, denoted by O0, of V-b3 with M-b3, M-b2, M-b1 and M-b0 are obtained in parallel. In cycle C2, S0 is right-shifted by 1 bit to get S1, and V-b2 is input to the crossbars to get the multiplication results O1. Similar operations are performed in C3 and C4. After C4, we get S4, the multiplication results of the input vector with four bit-slices if the matrix. In the following threes cycles C5 to C7, we shift and add S4 from the four crossbars to get the final multiplication result. For the fixed-point multiplication of an $N_M$-bit matrix with an $N_v$-bit vector, the processing cycle count is $C_{int} = N_v + (N_M - 1)$.

### 5.1.2 Fixed-Point and Floating-Point

![Figure 5.2: The bit layout of an (a) 8-bit signed integer and (b) a 64-bit double-precision floating-point number.](image)

We use an 8-bit signed integer and an IEEE 754-2008 standard [287] 64-bit double-precision floating-point number as an example to compare the difference between fixed-point and floating-point numbers. Typically, these designs refer to the format used to store and manipulate the digital representation of the data. As demonstrated in Figure 5.2 (a), fixed-
point numbers represent integers – positive and negative whole numbers – via a sign bit followed by multiple (e.g., $i$-bit) value bits, yielding a data range of $-2^i$ to $2^i - 1$. IEEE 754 double-precision floating-point numbers shown in Figure 5.2 (b) are designed to represent and manipulate rational numbers, where a number is represented with one sign bit ($s$), 11-bit exponent ($e$), and 52-bit fraction ($b_{51}b_{50}...b_0$). The value of a double-precision floating-point is interpreted as $(-1)^s \times (1.b_{51}b_{50}...b_0) \times 2^{(e-1023)}$, yielding a dynamic data range from $\pm 2.2 \times 10^{-308}$ to $\pm 1.8 \times 10^{308}$.

Different data formats have different computational performance, and hence resulting in different final precision. From data storage perspective, the storage characteristic and space required for fixed-point and floating-point numbers are also significantly different. The choice of appropriate data format is influenced by both the application’s computational accuracy requirements and the available hardware storage resources. However, in general, floating-point is a norm for high-precision computations because it can support a wider range of data values with the ability to precisely represent very small or very large numbers.

### 5.1.3 Scientific Computing

```plaintext
1 initiate x = x0
2 while (not converge) do
3     //Step 1: compute the residual
4     r = b - A * x
5     //Step 2: compute the correction
6     compute p
7     //Step 3: update the current solution
8     x = x + p
9 end while
```

Code 5.1: The iterative solver.

Scientific computing is an interdisciplinary science that solves computational problems in a wide range of disciplines including physics, mathematics, chemistry, biology, engineering, and other natural sciences subjects [288–290]. Those complex computing problems are
normally modeled by systems of large-scale Partial Differential Equations (PDEs). Since it is almost impossible to directly solve the analytical solution of those PDEs, a common practice is to discretize continuous PDEs into a linear model $Ax = b$ [166, 167] to be solved by numerical methods. The numerical solution of this linear system is usually obtained by an iterative solver [291–293]. Code 5.1 illustrates a typical computing process in iterative methods. The vector $x$ that needs to be solved is firstly initialized usually to an all-zero vector $x_0$, followed by three steps in the main body: (1) the residual (error) of the current solution vector is calculated as $r = b - Ax$; (2) to improve the performance of the estimated solution, a correction vector $p$ is computed based on the current residual $r$; (3) the current solution vector is improved by adding the correction vector as $x = x + p$, aiming to reduce the possible residuals produced in the next calculation iteration. The iterative solver stops when a defined convergence criteria is satisfied. Two widely used convergence criteria are (1) that the iteration index is less than a preset threshold $K$ or (2) that the L-2 norm of the residual ($\text{res} = ||b - Ax||^2$) is less than a preset threshold $\tau$. Notably, all the values involved in Code 5.1 are implemented as double-precision floating-point numbers to meet the high-precision requirement of mainstream scientific applications.

The various iterative methods follow the above computational steps and differ only in the calculation of the correction vectors. Among all candidate solutions, Krylov subspace approach is the standard method today. In this work, we focus on two representative Krylov subspace solvers – Conjugate Gradient (CG) [294] and Stabilized BiConjugate Gradient (BiCGSTAB) [295]. The computational kernels of these two methods are large-scale sparse floating-point matrix-vector multiplication $y = Ax$ as aforementioned, which requires the support of floating-point precision computation, imposing significant challenges to the underlying computing hardware.
5.2 Motivation and ReFLOAT Ideas

5.2.1 Motivation

Given the huge success of ReRAM-based high-parallel MVM engines in machine learning [88, 89, 91, 92, 221, 286], it is naturally attractive to design accelerators that extend ReRAM-based computing support to floating-point SpMV in scientific computing. Unfortunately, existing ReRAM-based accelerators are limited to models that have inherent tolerance for low-precision fixed-point approximation (e.g., machine learning) and are rarely explored to support floating-point processing in scientific computing. To bridge this gap, the most relevant work along this direction is proposed in [189]. In this work, the sparse matrix in scientific computing is first partitioned into multiple blocks. Each floating-point element in the block is converted to a fixed-point number by adding approximately the same number of padding bits as the effective data bit. The fixed-point number with extended bits are then deployed onto the fixed-point ReRAM crossbars [189] for processing. In order to facilitate the quantitative assessment of system performance and hardware overhead, we carefully selected the cycle number $T$ and the crossbar number $C$ as two metrics for measurement: (1) $T$ is directly related to the performance because a smaller $T$ indicates higher performance and vice versa, and (2) $C$ directly reflects the ability to execute floating-point MVMs in parallel under a limited number of on-chip ReRAMs [89, 175, 189]. Specifically, the smaller $C$, the stronger the parallelism. The primary goal of this work focuses on techniques that reduce the cost for floating-point processing in ReRAM.

Crossbar number. Supposing we are computing the multiplication of a matrix block $M$ with a vector segment $v$. In the matrix block $M$, the number of fraction bits is $f_M$ and the number of exponent bits is $e_M$. In the vector segment $v$, the number of fraction bits is $f_v$ and the number of exponent bits is $e_v$. To map the matrix fraction to ReRAM crossbars, we need $(f_M + 1)$ ReRAM crossbars because the fraction is normalized to a value with a leading 1. For example, (52+1) crossbars are needed to represent the 52-bit fraction in ...
double floating-point precision in [189]. To map the matrix exponent to ReRAM crossbars, we need $2^e_M$ ReRAM crossbars for $e_M$-bit exponent states, which is called padding in [189] where 64-bit paddings are needed for an $e_M = 6$. Thus, $C$ is calculated as

$$C = 4 \times (2^e_M + f_M + 1),$$  \hspace{1cm} (5.2)$$

where the leading multiplier 4 is contributed from sign bits of the matrix block and the vector segment.

Cycle number. Supposing the precision of digital-analog converters is 1-bit as that in [89, 189]. The number of value states in the vector segment is $(2^{e_v} + f_v + 1)$. For each input state, we need $(2^e_M + f_M + 1)$ to perform the shift-and-add to reduce the partial results from the ReRAM crossbars. For a higher computation efficiency, a pipelined input and reduce scheme [89] is used. Thus, $T$ is calculated as

$$T = (2^{e_v} + f_v + 1) + (2^e_M + f_M + 1) - 1.$$  \hspace{1cm} (5.3)$$

Cycle/crossbar number vs. accuracy loss. The bit number (of exponents and fractions in matrix blocks and vector segments) bridges the connection between the hardware cost and the processing accuracy. A better configuration of bit numbers may lead to low hardware cost, i.e., fewer cycles and crossbars, with an acceptable accuracy.

The cycle number for (a) various configurations of exponent bit number for vector segment and matrix block and (b) various configurations of fraction bit number for vector segment and matrix block and (c) the crossbar number for various configurations of fraction and fraction bit number for matrix block.

**Figure 5.3**: The cycle number and crossbar number under various bit length.
Cost exploration. We explore the effect of bit number on cycle number and crossbar number, shown in Figure 5.3. From Figure 5.3 (a) we can see with the increase of the exponent bit number for vector segment and matrix block, the cycle number increases exponentially. In Figure 5.3 (b) with the increase of the fraction bit number for vector segment and matrix block, the cycle number increases, but not as fast as that in 5.3 (a). From Figure 5.3 (c) we can see the increase of exponent and fraction bit number for matrix block leads to more crossbars, and the crossbar number increase faster with the increase of exponent bit number than that of fraction bit number. As a conclusion, larger bit number for matrix block and vector segment leads to higher hardware cost.

List 5.1: A data slice of crystm03. Each line contains row index, column index and edge value.

Locality in real-world data. We show a data slice of the crystm03 matrix from the SuiteSparse matrix collection [296] in List 5.1. As we can see, the index of the row and the column of the near entries are close, eg., 113xx, 117xx, 118xx, 1129x. The values of the entries are also close, i.e., they are around $10^{-13}$ to $10^{-15}$. Actually we can use 11290 as a shared bias for the row index and only one decimal digit to represent the column index and we can use $10^{-14}$ as a shared exponent bias for the values in this example. As a result, the required bits to represent the data slice is significantly reduced and consequently the computation cost. We call this phenomenon Index and Value Locality. The index and value similarity
commonly exists because the sparse matrices are collocated from real-word scenarios. Thus the elements in the matrix have spatial similarity with the nearby elements. The index and value similarity also provide use the opportunity for low cost floating-point computing and reduced memory size for the matrix.

5.2.2 Discussion

The bit numbers for exponent and fraction in matrix block contribute to the the number \( C \) of crossbars required for one floating-point multiplication on a matrix block and the processing cycle number \( T \). The bit numbers for exponent and fraction in vector segment contribute to the processing cycle number \( T \). Lower bits lead to lower cost for floating-point multiplication in ReRAM, thus a naive approach is to truncate bits for low computing latency and resource demand. However, bit truncation may lead to significantly slower convergence and, more importantly, nonconvergence. We show the iteration numbers for convergence under various exponent and fraction bit configurations in Table 5.1. In default double-precision, it takes 80 iterations to convergence. If we fix the exponent bits and truncate fraction bits, a 21-bit fraction takes 27 additional iterations and a fraction less than 21 bits leads to nonconvergence. If we fix the fraction bits and truncate exponent bits, 7-bit exponent increases the iteration number from 80 to 20620, and a exponent less than 7 bits leads to nonconvergence. In state-of-the-art ReRAM accelerator for scientific computing [189], to reduce the processing cost, Feinberg et al. assumed 6 bits for the exponent (64-bit padding). The fatal problem in [189] is that the solvers are not functional correct because of nonconvergence.

It is a challenge to reduce bit numbers for low-cost processing but maintain convergence at the same time. To tackle this challenge, in this work, for low-cost floating-point processing in ReRAM, we propose a data format rather than the default double-precision floating-point format and an accelerator architecture to support the proposed data format.
Table 5.1: The iteration numbers for convergence under various exp(onent) and fra(ction) bit configurations for matrix crystm03.

<table>
<thead>
<tr>
<th>exp</th>
<th>11</th>
<th>11</th>
<th>11</th>
<th>11</th>
<th>11</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>frac</td>
<td>52</td>
<td>30</td>
<td>29</td>
<td>28</td>
<td>27</td>
<td>26</td>
</tr>
<tr>
<td>#ite</td>
<td>80</td>
<td>82(+2)</td>
<td>82(+2)</td>
<td>83(+3)</td>
<td>83(+3)</td>
<td>84(+4)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>exp</th>
<th>11</th>
<th>11</th>
<th>11</th>
<th>11</th>
<th>11</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>frac</td>
<td>25</td>
<td>24</td>
<td>23</td>
<td>22</td>
<td>21</td>
<td>20</td>
</tr>
<tr>
<td>#ite</td>
<td>90(+10)</td>
<td>93(+13)</td>
<td>93(+13)</td>
<td>95(+15)</td>
<td>107(+27)</td>
<td>NC</td>
</tr>
</tbody>
</table>

NC indicates nonconvergence.

5.3 REFLOAT Data Format

In this section, we introduce the REFLOAT data format and illustrate the conversion between a coordinate (COO) format and REFLOAT. We also discuss how to perform computation on REFLOAT format to achieve high performance with minimal hardware overhead.

5.3.1 ReFloat Overview

![Figure 5.4: The conversion of index and value in floating-point format to REFLOAT format.](a) A block in full precision. (b) A block in REFLOAT.

In general, we use \( \text{ReFloat}(b,e,f) \) to represent the REFLOAT format, where \( b \) denotes
Table 5.2: List of symbols and descriptions.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ReFloat($b, e, f$): ReFloat format notation.</td>
<td></td>
</tr>
<tr>
<td>$2^b$</td>
<td>The size of a square block.</td>
</tr>
<tr>
<td>$e$</td>
<td>The number of exponent bits for a matrix block.</td>
</tr>
<tr>
<td>$f$</td>
<td>The number of fraction bits for a matrix block.</td>
</tr>
<tr>
<td>$A$</td>
<td>A sparse matrix.</td>
</tr>
<tr>
<td>$b$</td>
<td>The bias vector for a linear system.</td>
</tr>
<tr>
<td>$x$</td>
<td>The solution vector for a linear system.</td>
</tr>
<tr>
<td>$r$</td>
<td>The residual vector for a linear system.</td>
</tr>
<tr>
<td>$a$</td>
<td>A scalar of $A$.</td>
</tr>
<tr>
<td>$(a)_e$</td>
<td>The exponent of $a$, $(a)_e \in {0, 1, 2, \ldots}$.</td>
</tr>
<tr>
<td>$(a)_f$</td>
<td>The fraction of $a$, $(a)_f \in (1, 2)$.</td>
</tr>
<tr>
<td>$A_c$</td>
<td>A block of the sparse matrix $A$.</td>
</tr>
<tr>
<td>$(i, j)$</td>
<td>The index for the block $A_c$.</td>
</tr>
<tr>
<td>$(ii, jj)$</td>
<td>The index for the scalar $a$ in the block $A_c$.</td>
</tr>
<tr>
<td>$(iii, jjj)$</td>
<td>The index for the scalar $a$ in the matrix $A$.</td>
</tr>
<tr>
<td>$e_b$</td>
<td>The number of bias bits for the exponents of scalars in the block $A_c$.</td>
</tr>
<tr>
<td>$e_v$</td>
<td>The number of exponent bits for a vector segment.</td>
</tr>
<tr>
<td>$f_v$</td>
<td>The number of fraction bits for a vector segment.</td>
</tr>
</tbody>
</table>

the matrix block size (the length and width of a square matrix block) is $2^b$, $e$ and $f$ respectively denote the number of bits for the exponent and the fraction. The symbols and corresponding descriptions related to ReFLOAT are listed in Table 5.2.

We use Figure 5.4 to illustrate the idea of ReFLOAT. As shown in Figure 5.4 (a), each scalar is in 64-bit floating-point format. It requires a 32-bit integer for row index and a 32-bit integer for column index to locate each element in the matrix block. Therefore, we will use $8 \times (32 + 32 + 64) = 1024$ bits for storage of the eight scalars. In this example, assuming we use the ReFLOAT($2, 2, 3$) format as depicted in Figure 5.4 (b). Alternatively, (1) each scalar in the block can be indexed by two 2-bit integers; (2) the element value is represented by a $1 + 2 + 3 = 6$-bit floating point; (3) the block is indexed by two 30-bit integers and (4) a 11-bit exponent bias $e_b$ is also recorded. Therefore, we only use $8 \times (2 + 2 + 6) + 2 \times 30 + 11 = 151$ bits to store the entire matrix block, which reduces the memory requirement by approximately $10 \times (151 \text{ vs. } 1024)$. This reduction in bit representation is
also beneficial for reducing the number of ReRAM crossbars for computation in hardware implementation. More specifically, the full precision format consumes 118 crossbars as illustrated in previous work [189], while only 16 crossbars are needed in our design with ReFLOAT reformatting. Hence, within the limits of the same chip area, our design can process more matrix blocks at a time.

### 5.3.2 Conversion to ReFloat

![Diagram of matrix block conversion](image)

Figure 5.5: Comparison of a matrix block (a) in original full precision format and (b) in ReFLOAT format.

In order to configure the original matrix with a ReFLOAT \((b, e, f)\) format, three parameters need to be determined in advance. \(b\) defines how the indices of input data are converted, while the size of \(b\) is determined by the physical size of ReRAM crossbars, i.e., a crossbar with \(2^b\) wordlines and \(2^b\) bitlines. Assuming the crossbar size is \(4 \times 4\), which indicates
As demonstrated in Figure 5.5 (a), the leading 30 bits, i.e., \( b_{31} \) to \( b_2 \) of the index \((iii, jjj)\) for a scalar in the matrix \( A \) are copied to the same bits in the index \((i, j)\) for the block \( A_c \). For each scalar in the block \( A_c \), the index \((ii, jj)\) for that scalar inside the block \( A_c \) are copied from the last two bits of the index \((iii, jjj)\). The block index \((ii, jj)\) are shared by the scalars in the same block, and each scalar uses less bits for index inside that block. Thus, we saved memory space for indexing.

The hyper parameters \( e \) and \( f \) compress the floating-point values. Essentially, a floating-point number consists of three parts, (1) the sign bit, (2) the exponent bits, and (3) the fraction bits. In the conversion to REFLOAT, we first keep the sign bit. For the fraction, we only keep the leading \( f \) bits from the original fraction bits and remove the rest bits in the fraction, as shown in Figure 5.5 (b). For the exponent bits, we need first determine the bias value \( e_b \) for the exponent. As \( e \) means the number of bits for the “swing” range, we need to find an optimal bias value \( e_b \) to fully utilize the \( e \) bits. We formalize the problem as an optimization for find the \( e \) to minimize a loss target \( L \), defined as,

\[
\min_{e_b} L, \quad L = \sum_{a \in A_c} \left( \log_2 \left( \frac{a}{(a)_f \times 2^e} \right) \right)^2 = \sum_{a \in A_c} ((a)_e - e_b)^2. \tag{5.4}
\]

Let \( \frac{\partial L}{\partial e_b} = 0 \), we can get

\[
e_b = \left[ \frac{1}{|A_c|} \sum_{a \in A_c} (a)_e \right]. \tag{5.5}
\]

Thus, in the conversion, we use the original exponent to minus the optimal \( e_b \) to get an \( e \)-bit signed integer. The \( e \)-bit signed integer is the exponent in REFLOAT.

\[
\begin{bmatrix}
(-1) \times 1.1111 \times 2^7 & 1.0101 \times 2^8 \\
(-1) \times 1.0000 \times 2^9 & 1.0001 \times 2^7
\end{bmatrix}
= 
\begin{bmatrix}
-248.0 & 336.0 \\
-512.0 & 136.0
\end{bmatrix}. \tag{5.6}
\]

\[
2^8 \times 
\begin{bmatrix}
(-1) \times 1.11 \times 2^{-1} & 1.01 \times 2^0 \\
(-1) \times 1.00 \times 2^1 & 1.00 \times 2^{-1}
\end{bmatrix}
= 
\begin{bmatrix}
-224.0 & 320.0 \\
-512.0 & 128.0
\end{bmatrix}. \tag{5.7}
\]
We use an example to intuitively illustrate the format conversion. The original floating-point values in Eq. 5.6 are converted to $\text{ReFloat}(x,2,2)$ format in Eq. 5.7, where $e_b = 8$. From this example we can see, $\text{ReFloat}$ incurs conversion loss for the conversion of floating-point values from the original. However, as we have mentioned in Sec. 5.2.1, $\text{ReFloat}$ is application-specific for scientific computing, and the errors in the iterative solver are gradually corrected. Thus, the errors introduced by the conversion will also be corrected in the iteration. More importantly, with $\text{ReFloat}$, we can achieve high performance computation because the low computation cost in $\text{ReFloat}$ format. We will show the performance and convergence of the iterative solver in $\text{ReFloat}$ format in Sec. 5.5.

### 5.3.3 Computation with ReFloat

The whole matrix $A$ is partitioned into blocks. To compute the matrix-vector multiplication $y = Ax$, the input vector $x$ and the output vector $y$ are partitioned correspondingly into vector segments $x_c$ and $y_c$. The size of the vector segments is $(2^b \times 1)$.

For the $p$-th output vector segment $y_c(p)$, the computation will be,

\[
y_c(p) = \sum_i A_c(i, p)x_c(i), \tag{5.8}
\]

where $A_c(i, p)$ is the matrix block indexed by $(i, p)$ and $x_c(i)$ is the input vector segment index by $i$. The matrix blocks at the $p$-th block column are multiplied with the input vector segments for partial sums and then they are accumulated. In the computation for each matrix block, because the original matrix block $A_c(i, p)$ is converted to

\[
A_c(i, p) = 2^{e_b(i,p)}\tilde{A}_c(i, p), \tag{5.9}
\]

and the original vector segment $x_c(i)$ is converted to

\[
x_c(i) = 2^{e_v(i)}\tilde{x}_c(i), \tag{5.10}
\]

111
thus, the matrix-vector multiplication for the matrix block $A_c(i, p)$ and the vector segment $x_c(i)$ is computed as

$$A_c(i, p)x_c(i) = 2^{e_b(i, p) + e_v(i)}\tilde{A}_c(i, p)\tilde{x}_c(i).$$

(5.11)

The matrix-vector multiplication for the $p$-th output vector segment in Eq. 5.8 is then computed as

$$y_c(p) = \sum_i 2^{e_b(i, p) + e_v(i)}\tilde{A}_c(i, p)\tilde{x}_c(i).$$

(5.12)

Here we can see, with ReFLOAT format, the block matrix multiplication in Eq. 5.8 is preserved. The original high-cost multiplication in full precision $A_c x_c$ is replaced by a low-cost multiplication $\tilde{A}_c \tilde{x}_c$. The architecture for high performance scientific computing in ReFLOAT will be presented in Sec. 5.4.

5.4 Accelerator Architecture for ReFLOAT

In this section, we will present an accelerator architecture for efficient processing of scientific computing in ReRAM in ReFLOAT format. We will present the architecture for floating-point SpMV engine, the data streaming of the sparse matrix into the accelerator, the format conversion between ReFLOAT and the 64-bit double-precision floating-point and the mapping of the Krylov subspace solvers to the proposed accelerator.

5.4.1 Accelerator Overview

Figure 5.6 shows the overall architecture of the proposed accelerator for scientific computing in ReRAM with ReFLOAT. The accelerator is organized by bank architecture. Within each bank, ReRAM crossbars are deployed for the processing of floating-point MVM on matrix blocks. The Input Vector (IV) and Output Vector (OV) buffer are used for buffering the input and output vectors respectively. A commit buffer collects the partial SpMV results.
Figure 5.6: The overall accelerator architecture for scientific computing in REFLOAT format.

from the ReRAM crossbars and the Multiply-and-Accumulate (MAC) units are used to update the vectors. The scheduler is responsible for the coordination of the processing. The Input and Output (IO) buffer is used for the matrix data movement from the IO to the accelerator.

5.4.2 Processing Engine

Figure 5.7: The architecture for (a) a processing engine for floating-point MVM on a matrix block and (b) a crossbar cluster for fixed-point MVM.

The most important component in the accelerator is the processing engine for floating-point matrix-vector multiplication in REFLOAT format. A processing engine is a logical unit for the processing of floating-point SpMV in REFLOAT format. The processing
engine consists of a bunch of ReRAM crossbars and the peripheral functional units. The architecture of the processing engine is shown in Figure 5.7. Assuming we are processing the floating-point SpMV on a matrix block with the format ReF10at(b, e, f).

The input to the processing engine are (1) a matrix block in ReF10at(b, e, f) format, (2) an input vector segment in floating-point with e_v exponent bits and f_v fraction bits and the vector length is 2^b, and (3) the exponent bias bits e_b for the matrix block. The output of a processing engine is a vector segment for SpMV on the matrix block, and the output vector segment is in double-precision floating-point.

Before the computation, the matrix block is mapped to the ReRAM crossbars. The mapping is shown in Figure 5.7 (b). The fraction part of the matrix block in ReF10at(b, e, f) represents a number of 1.b_f−1...b_0, then we have (f + 1) bits for mapping. The e-bit exponent of the matrix block contributes to 2^e padding bits for alignment, then we have another 2^e bits for mapping. Thus, we map the total (2^e + f + 1) bits to (2^e + f + 1) ReRAM crossbars, where the i-th bits of the matrix block is mapped to the i-th crossbar. (Here, we assume the cell precision for the ReRAM crossbars is 1-bit. For 2-bit cells, two consecutive bits are mapped to a crossbar.) For the input vector segments with e_v exponent bits and f_v exponent bits, the input to the driver has (2^e_v + f_v + 1) bits.

In the processing, a cluster of crossbars are deployed for the processing of fixed-point MVM for the fraction part of the input vector segment with the fraction part of the matrix block using the shift-and-add method, as the example in Figure 5.1. The input bits are applied to the crossbars by the driver and the output from the crossbar is buffed by a Sample/Hold (S/H) unit and then converted to digital by a shared Analog/Digital Converter (ADC). For each input bit to the driver (we assume an 1-bit DAC), as the crossbar size is 2^b, the bit number for the ADC to convert the multiplication results from crossbars is f_x = b. Then we need to shift-and-add the results from all (2^e_v + f_v + 1) crossbars to get the results for the 1-bit multiplication of the vector with the matrix fraction. Thus, the
number of bits for the $3$ is

$$f_c = 2^e + f + 1 + b. \quad (5.13)$$

Next, we consequentially input the bits in $1$ to the crossbars and shift-and-add the collected $3$ for each of the $(2^e v + f_v + 1)$ bits to get $4$, which is the result for the multiplication of the matrix block with the input $1$. The number of bits for $4$ is

$$f_g = f_c + 2^e v + f_v + 1 + b. \quad (5.14)$$

In a processing engine, as shown in Figure 5.7 (a), there is a sign bit for the matrix block, thus we need two crossbar cluster for the signed multiplication. The elements in the input vector segment also have a sign bits. Thus, we need four $4$ to subtract them to get $5$, which is the results for the multiplication of the matrix block with the vector segment. The number of bits for $5$ is $(f_g + 1)$. We also note that $5$ is a signed number because of the subtraction.

In next step, we convert the $5$ to a double-precision floating-point $6$. $e_b$ $7$ is the exponent bias for the matrix block and $e_v$ $8$ is exponent for the vector segment. Thus we add $7$ and $8$ to the exponent of $9$ to get the $9$, which is the final results for the multiplication of the matrix block with the vector segment. $9$ is in 64-bit double-precision floating-point format.

### 5.4.3 Streaming & Scheduling

The processing engine is designed for the computation of MVM on a matrix block. For the whole SpMV, the computation is partitioned to the block MVM. However, because of the sparsity in the real-world large-scale matrix, the streaming on the matrix and the scheduling for the block matrix MVMs need careful consideration.

Matrix Market File Format [297] (MMFF) is the matrix format used in scientific computing. In MMFF, a line of entry contains the row index, the column index and the value.
for the edge (a non zero) in a sparse matrix, as shown in List 5.1. The order of the entries are usually in row major for the processing on the original matrix or in column major for the processing on the transposed matrix. We show an example for row-major data layout in Figure 5.8 (b) for the sparse matrix in Figure 5.8 (a). In the row-major layout, all the non-zeros in one row are stored continuously.

For the computation in Eq. 5.8, which perform the computations for the same destination vector segment, it is required to continuously access non-zeros from the same block column. For example, the red and yellow non-zeros are accessed. But in the row-major layout, Figure 5.8 (b), we can see the accesses are irregular, which leads to the waste of memory bandwidth. Thus, we need to perform the computation for the same source vector segment for partial results and then accumulate the partial results to the destination vector segments.

However, there is another problem with the row-major layout. Assume the matrix block to be processed is $4 \times 4$ and there are four available processing engines. To avoid bandwidth waste, the red and blue non-zeros in the first 4 rows need to be fetched. They are continuously accessed as shown in Figure 5.8 (b). but the non-zeros are only for two matrix blocks, as shown in Figure 5.8 (a). The number of matrix blocks in the next four rows are unknown before all data are fetched and it is possible to get four matrix blocks in the next fetching. At this time, the non-zeros in the next four rows can not be accessed. Thus, only two out of the four processing engines are busy but the other two are idle.
For full utilization of the processing engines, we propose a block-major layout, as shown in Figure 5.8 (c). Within a matrix block, the non zeros are stored in row-major order, and the data for the blocks are continuously stored. With the block-major layout, we fetch the non zeros to the accelerator until we exhaust all available processing engines. To avoid conflict on writing to the same vector segment, we use a commit buffer, as shown in Figure 5.6, to collect the partial results from the processing engines and then update the output vector with the MAC units.

### 5.4.4 Accelerator Programming

Conjugate Gradient (CG) [294] and Stabilized BiConjugate Gradient (BiCGSTAB) [295] are the two Krylov subspace methods used as the iterative solver in this work. We use CG as an example to illustrate how we program the accelerator for the solver. The pseudo code for the CG solver is listed in Code 5.2. In the code, We use `double` to indicate a variable in 64-bit(double) floating-point precision, and `refloat` to indicate a variable in `RE_FLOAT`. The variable name with a `mat` suffix indicates a matrix, the variable name with a `vec` suffix indicates a vector, and others are scalars.

At the beginning, the exponent number for the matrix block $e$, the fraction number for the matrix block $f$, the exponent number for the vector segment $ev$ and the fraction number for the vector segment $fv$ in `refloat` is set. Within the accelerator, the crossbars in a bank(subbank) will be configured with the parameter as clusters for processing SpMV in `refloat`. Then the sparse matrix $A_{\text{mat}}$ is converted to `refloat $A_{\text{mat}}$` and stored in the block-major layout (Sec. 5.4.3) at Line 2 and 3. The bias vector $b_{\text{vec}}$ for the linear system and the initial solution $x_{\text{vec}}$ is loaded to the accelerator at Line 4 and 5. At Line 6, the SpMV of $A_{\text{mat}}$ and $x_{\text{vec}}$ is processed, where the matrix blocks of $A_{\text{mat}}$ are streamed to the accelerator and partial results are accumulated to a `double` vector $Ax$. At Line 7 the error vector $r_{\text{vec}}$ is computed by the MACs. At Line 8, a precision correction
refloat: e, f, ev, ef

double A_mat
refloat Ar_mat = (refloat) A_mat
double b_vec
double x_vec = x0_vec
double Ax = Ar_mat * x_vec
double r_vec = b_vec - Ax
double p_vec = r_vec
while (not converge) do
  double Ap_vec = Ar_mat * (refloat) p_vec
  double rr = r_vec.T * r_vec
  double pAp = p_vec.T * Ap_vec
  double alpha = rr / pAp
  x_vec = x_vec + alpha * p_vec
  r_vec = r_vec - alpha * Ap_vec
  double rrnew = r_vec.T * r_vec
  double beta = rrnew / rr
  p_vec = r_vec + beta * p_vec
end while

Code 5.2: The pseudo code for the CG solver.

vector p_vec is created. Line 9 to Line 19 are the main body for the CG solver. The SpMV is processed on the matrix and the precision correction vector p_vec which is converted into refloat before processing at Line 10. From Line 11 to Line 13, a coefficient alpha is calculated. The dot product of two vector is performed by the MACs. At Line 14 and Line 15, the coefficient alpha is used to update the solution vector x_vec and the error vector vector r_vec element-wisely. From Line 16 to Line 18, an other coefficient beta is calculated to update the precision correction vector. The main body continue to run until a desired stop condition (such as that the number of the iterations is larger than a threshold or that the residual rr is smaller than a threshold) is reached.
5.5 Evaluation

In this section, we evaluate the speedup of our proposed accelerator in REFLOAT format against the state-of-the-art ReRAM accelerator [189], denoted as ESCMA, for scientific computing. We use a high-end Tesla V100 GPU as the baseline for comparison. We present the error curves of the iterative solvers of REFLOAT to better understand the behavior of the low-cost floating-point computing.

5.5.1 Evaluation Setup

Table 5.3: Platform Configuration.

<table>
<thead>
<tr>
<th></th>
<th>GPU (Tesla V100 SXM2)</th>
<th>ESCMA</th>
<th>ReFloat</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture</td>
<td>Volta</td>
<td>Crossbar Size 128</td>
<td>Crossbar Size 128</td>
</tr>
<tr>
<td>Memory</td>
<td>16GB HBM2</td>
<td>Precision double</td>
<td>Precision refloat</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Comp. ReRAM 17.1Gb</td>
<td>Comp. ReRAM 17.1Gb</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>ADC</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>10-bit pipelined SAR ADC @ 1.5GS/s</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>ReRAM Cells</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1-bit SLC, $T_w = 50.88\text{ns}$, Comp. Latency=107ns @ $(128 \times 128)$</td>
<td></td>
</tr>
</tbody>
</table>

We list the configurations for the baseline GPU platform, the state-of-the-art ReRAM accelerator [189] for scientific computing (ESCMA) and our REFLOAT in Table 5.3. For the baseline GPU, we use an NVIDIA Tesla V100 SXM2 GPU, which has 5012 cuda cores and a 16GB HBM2 memory. We use cuda 10.2 and cuSPARSE routines in the iterative solvers for the processing and computing on sparse matrices. We measure the running time for the solvers on the GPU. For the two ReRAM accelerators, i.e. ESCMA and REFLOAT,
we simulate with the parameters in Table 5.3. Both the two ReRAM accelerators have 128 Banks and the crossbar size is $128 \times 128$. In ESCMA, we configure 64 clusters for each bank, which is slightly larger than that (56) in the original work [189]. There are 128 crossbars in each cluster. The precision in ESCMA is double floating-point. In REFLOAT, we configure 128 banks, 128 subbanks per bank, and 64 crossbars per subbank. The precision in REFLOAT is refloat with a default setting that $e = 3$, $f = 3$, $\epsilon_v = 3$ and $f_v = 8$.

For the two ReRAM accelerators, the equivalent computing ReRAM is 17.1Gb. The ADC and ReRAM cells for the two accelerators are of the same configurations. We use a 1.5GS/s 10-bit pipelined SAR ADC [298] for conversion. The DAC is 1-bit, which is implemented by wordline activation. We use 1-bit SLC [264] and the write latency is 50.88ns. The computing latency for one crossbar including the ADC conversion is 107ns [189]. In the two ReRAM accelerators, each bank has a 16MB memory for buffering and 128 floating-point MACs for vector related processing.

The matrices used in the evaluation are listed in Table 5.4. We evaluate on 12 solvable matrices from the SuiteSparse Matrix Collection (formerly the University of Florida Sparse Matrix Collection) [296]. The size, i.e., the number of rows, of the matrices ranges from 4,875 to 204,316 and the Number of Non-Zero entries (NNZ) of the matrices ranges from 105,339 for 1,660,579. NNZ/Row is a metric for sparsity. A smaller NNZ/Row indicates a sparser matrix. NNZ/Row ranges from 4.0 to 27.7. We also visualize the matrices in Table 5.4.

We apply the iterative solvers CG and BiCGSTAB on the matrices. The convergence criteria for the solvers is that the L-2 norm of the residual vector (we use the term “residual” (denoted by $R^2$) for simplicity to call it in this section) is less than $10^{-8}$. The error ($e_i = x_i - \tilde{x}_i$) of any scalar in the solution vector is bonded by the residual, because $|e_i|^2 \leq \sum_i |e_i|^2 = |x - \tilde{x}|^2 = |r|^2 / |A|^2 = \frac{1}{|A|^2}R^2$, where $|A|$ is a constant.
Table 5.4: Matrices in the evaluation.

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>#Rows</th>
<th>NNZ</th>
<th>NNZ/R</th>
<th>κ</th>
</tr>
</thead>
<tbody>
<tr>
<td>353</td>
<td>crystm01</td>
<td>4875</td>
<td>105339</td>
<td>21.6</td>
<td>4.21e+2</td>
</tr>
<tr>
<td>1313</td>
<td>minsurfo</td>
<td>40806</td>
<td>203622</td>
<td>5.0</td>
<td>8.11e+1</td>
</tr>
<tr>
<td>354</td>
<td>crystm02</td>
<td>13965</td>
<td>322905</td>
<td>23.1</td>
<td>4.49e+2</td>
</tr>
<tr>
<td>2261</td>
<td>shallow_water1</td>
<td>81920</td>
<td>327680</td>
<td>4.0</td>
<td>3.63e+0</td>
</tr>
<tr>
<td>1288</td>
<td>wathen100</td>
<td>30401</td>
<td>471601</td>
<td>15.5</td>
<td>8.24e+3</td>
</tr>
<tr>
<td>1311</td>
<td>gridgena</td>
<td>48962</td>
<td>512084</td>
<td>10.5</td>
<td>5.74e+5</td>
</tr>
<tr>
<td>1289</td>
<td>wathen120</td>
<td>36441</td>
<td>565761</td>
<td>15.5</td>
<td>4.05e+3</td>
</tr>
<tr>
<td>355</td>
<td>crystm03</td>
<td>24696</td>
<td>583770</td>
<td>23.6</td>
<td>4.68e+2</td>
</tr>
<tr>
<td>2257</td>
<td>thermomech_TC</td>
<td>102158</td>
<td>711558</td>
<td>6.9</td>
<td>1.23e+2</td>
</tr>
<tr>
<td>1848</td>
<td>Dubcova2</td>
<td>65025</td>
<td>1030225</td>
<td>15.84</td>
<td>1.04e+4</td>
</tr>
<tr>
<td>2259</td>
<td>thermomech_dM</td>
<td>204316</td>
<td>1423116</td>
<td>6.9</td>
<td>1.24e+2</td>
</tr>
<tr>
<td>845</td>
<td>qa8fm</td>
<td>66127</td>
<td>1660579</td>
<td>25.1</td>
<td>1.10e+2</td>
</tr>
</tbody>
</table>

5.5.2 Performance

CG Solver

We show the performance of GPU in double and single precision floating-point, ESCMA [189] and REFLOAT for CG solver in Figure 5.9. The performance $p$ is defined as $p = t_x / t_{\text{GPU(double)}}$, $x = \text{GPU(single)}, \text{ESCMA}$ or \text{REFLOAT}. $t$ is the processing time for the iterative solver to satisfy that the residual is less than $10^{-8}$. Overall, the geometric-mean(GMN) performance of GPU(single), ESCMA and REFLOAT are 0.78×, 6.07× and 18.71× respectively, resulting in a 3.08× for REFLOAT against ESCMA. GPU(single) benefits from faster SpMV on lower bits, but the iteration number is increased. The highest
performance of GPU(single) $1.22 \times$ for Matrix 2261 and for most matrices the performance gain for switching double to single is less than $1.2 \times$. For Matrix 1311, the GPU(single) performance is $0.015 \times$, because it takes 1468 additional iterations. Less bits, such as FP16 in a straightforward way, will not converge. For most of the matrices, ESCMA and ReFLOAT performs better than the baseline GPU. However, for matrix 2257 and 2259, the performance of ReFLOAT is $3.97 \times$ and $2.68 \times$ respectively. The performance of ESCMA is even lower on the matrix 2257 and 2259, it is $0.72 \times$ and $0.51 \times$ respectively.

![Figure 5.9](image)

**Figure 5.9**: The performance of GPU(in double and single precision), ESCMA and ReFLOAT for CG solver.

### BiCGSTAB Solver

We show the performance of GPU double and single precision floating-point, ESCMA [189] and ReFLOAT for BiCGSTAB solver in Figure 5.10. The geometric-mean(GMN) performance of GPU(single), ESCMA and ReFLOAT are $1.15 \times$, $5.09 \times$ and $21.88 \times$ respectively, resulting in a $4.30 \times$ for ReFLOAT against ESCMA. Notice that matrix 1311 does not converge in GPU(single), and the $1.15 \times$ geomean excludes this matrix. The trend for the three platforms on the evaluated matrices are similar to that for CG solver. In each iteration, for CiCGSTAB solver, there are two SpMV on the whole matrix, while for CG solver, there is one SpMV on the whole matrix. For ESCMA, more time is consumed for the processing of two SpMVs, leading to a slightly lower performance than that of CG solver ($5.09 \times$ v.s. $6.07 \times$) compared with the baseline GPU. From Table 5.6 we can see, the gap (the difference of number of iterations to get converge) in BiCGSTAB solver is smaller.
than the gap in CG solver for most matrices. For matrix 355, 2257, 2259 and 845, the
gap is negative, which means it takes fewer iterations in \textit{refloat} compared with that in
\textit{double}. For \textit{ReFLOAT}, although more time is consumed for the processing of two SpMVs
in BiCGSTAB, which is the same scenarios as that in ESCMA, because the gap is smaller,
the performance of \textit{ReFLOAT} for BiCGSTAB solver is higher compared with CG solver,
i.e., $21.88 \times \text{v.s.} \ 18.71 \times$.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{performance.png}
\caption{The performance of GPU (in double and single precision), ESCMA and \textit{ReFLOAT} for BiCGSTAB solver.}
\end{figure}

5.5.3 Accuracy

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{convergence.png}
\caption{The convergence traces of CG solver of GPU (red line) and \textit{ReFLOAT} (blue line).}
\end{figure}

We show the convergence traces (the residual over each iteration) of the evaluated
matrices in \textit{double} and in \textit{refloat} for CG solver in Figure 5.11 and the convergence traces
for BiCGSTAB solver in Figure 5.12. The iteration number is normalized by the consumed time for the GPU baseline. The configurations of bit number for matrix block and vector segment in refloat are listed in Table 5.5. The absolute (non-normalized) iteration number to reaching convergence is listed in Table 5.6.

For CG solver, from Table 5.6 we can see, refloat leads to more number of iterations to get converged when we do not consider the time consumption for each iteration. From Figure 5.11 we can see, with the low bit representation, the residual curves are almost the same as the residual curves in default double. Most importantly, all the traces in refloat get converged faster than the traces in double. For matrix 1288 and matrix 1848, the bit number for fraction of vector segment is 16 because the default 8 leads to nonconvergence.

For BiCGSTAB solver, from Table 5.6 we can see, while refloat leads to more number of iterations to reaching convergence for 5 matrices, the number of iterations to reaching convergence for 4 matrices are even fewer than those in double. We infer that is because lower bit representation helps to enlarge the changes in the correction term, thus leads to fewer iterations. We also notice there are spikes in the residual curves in refloat more frequently than spikes in double, but they finally reach convergence.
Table 5.5: The bit number for exponent and fraction of matrix block and vector segment in ReFLOAT.

<table>
<thead>
<tr>
<th>ID</th>
<th>CG</th>
<th>BiCGSTAB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>e</td>
<td>f</td>
</tr>
<tr>
<td>353</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>1313</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>354</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>2261</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>1288</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>1311</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>1289</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>355</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>2257</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>1848</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>2259</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>845</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

5.5.4 Memory Overhead

In Figure 5.13, we compare the memory overhead for the matrix in refloat with that in double (used in ESCMA [189]). On average, refloat consumes 0.21 × memory compared with double. For matrices except 2257 and 2259, refloat consumes less than 0.2 × memory compared with double, which is 0.47 × and 0.44 × respectively. For matrix 2257 and matrix 2259, the average density within a matrix is relatively lower, thus more memory is consumed for the matrix block index and the exponent bias as discussed in Sec. 5.3.1.
Table 5.6: The absolute number of iterations to reaching convergence.

<table>
<thead>
<tr>
<th>ID</th>
<th>CG double</th>
<th>refloat</th>
<th>gap</th>
<th>BiCGSTAB double</th>
<th>refloat</th>
<th>gap</th>
</tr>
</thead>
<tbody>
<tr>
<td>353</td>
<td>68</td>
<td>85</td>
<td>+17</td>
<td>48</td>
<td>51</td>
<td>+3</td>
</tr>
<tr>
<td>1313</td>
<td>52</td>
<td>55</td>
<td>+3</td>
<td>30</td>
<td>69</td>
<td>+39</td>
</tr>
<tr>
<td>354</td>
<td>54</td>
<td>95</td>
<td>+41</td>
<td>52</td>
<td>79</td>
<td>+27</td>
</tr>
<tr>
<td>2261</td>
<td>11</td>
<td>11</td>
<td>0</td>
<td>7</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>1288</td>
<td>262</td>
<td>305</td>
<td>+43</td>
<td>195</td>
<td>205</td>
<td>+10</td>
</tr>
<tr>
<td>1311</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1289</td>
<td>294</td>
<td>401</td>
<td>+107</td>
<td>216</td>
<td>317</td>
<td>+101</td>
</tr>
<tr>
<td>355</td>
<td>80</td>
<td>95</td>
<td>+15</td>
<td>55</td>
<td>52</td>
<td>-3</td>
</tr>
<tr>
<td>2257</td>
<td>55</td>
<td>56</td>
<td>+1</td>
<td>43</td>
<td>36</td>
<td>-7</td>
</tr>
<tr>
<td>1848</td>
<td>162</td>
<td>214</td>
<td>+52</td>
<td>117</td>
<td>145</td>
<td>+28</td>
</tr>
<tr>
<td>2259</td>
<td>57</td>
<td>58</td>
<td>+1</td>
<td>43</td>
<td>36</td>
<td>-7</td>
</tr>
<tr>
<td>845</td>
<td>53</td>
<td>54</td>
<td>+1</td>
<td>41</td>
<td>45</td>
<td>-4</td>
</tr>
</tbody>
</table>
Chapter 6

Conclusion

In this dissertation, I study the architecture-level optimizations for efficient acceleration of training in deep learning and efficient acceleration of graph processing. I present ACCPAR, a principled and systematic method to determining the tensor partition among heterogeneous accelerator arrays for achieving optimal performance. ACCPAR considers the complete tensor partition space and reveals a new and previously unknown parallelism configuration. It optimizes performance based on a cost model considering both computation and communication cost of heterogeneous execution environment. I present PIPELAYER, a ReRAM-based PIM accelerator architecture for high-throughput deep learning training acceleration. I propose efficient layer-wise pipeline to exploit intra- and inter-layer parallelism. PIPELAYER enables the highly pipelined execution of both training and testing. I present GRAPHR, the first ReRAM-based graph processing accelerator. The key insight of GRAPHR is that if a vertex program of a graph algorithm can be expressed in sparse matrix vector multiplication (SpMV), it can be efficiently performed by ReRAM crossbar. The core graph computations are performed in sparse matrix format in graph engines (ReRAM crossbars) and edges are partitioned and sequentially fetched to the graph engines. I present REFLOAT, a data format and a corresponding accelerator architecture to support low-cost floating-point SpMV in ReRAM. REFLOAT is tailored for processing on ReRAM crossbars where the number of effective bits is greatly reduced to reduce the crossbar cost and cycle cost for the floating-point multiplication on a matrix block, thus the overall floating-point SpMV processing is boosted.
Bibliography


“Google supercharges machine learning tasks with tpu custom chip.”


131


133


Biography

Linghao Song is a Ph.D. candidate in the Department of Electrical and Computer Engineering at Duke University. His research interests lie in the field of computer architecture, deep learning accelerators, ReRAM-based accelerators and deep learning applications. He received M.S. in Electrical Engineering from University of Pittsburgh in 2017 and B.S.E. in Information Engineering from Shanghai Jiao Tong University in 2014. He worked as a Research Intern at Oak Ridge National Laboratory (ORNL) in Summer 2018 and a Research Intern at Pacific Northwest National Laboratory (PNNL) in Summer 2019.

Linghao’s works on computer architecture and accelerators are published on major computer architecture conferences, including HPCA’20 [174], HPCA’19 [2], HPCA’18 [175] and HPCA’17 [92], and a journal CCF THPC [299]. His leading work on a survey of deep learning accelerators is published on Elsevier Engineering [300]. His work on high-energy physics data analysis is published on ICASSP’19 [15]. He leads an invited work on parallelism in deep learning accelerators on ASP-DAC’20 [301]. He has mentored several junior or undergraduate students and their works are published on DAC’18 [93], DATE’18 [94], SafeAI@AAAI’19 [5] and DATE’20 [302]. He also has collaborative works published on ISCA’20 [303], ASP-DAC’17(Best Paper Award) [304], ASP-DAC’18(Best Paper Nomination) [95], IEEE TCAD [305], DAC’20 [306], ASP-DAC’20 [307], DAC’19 [99], ICCAD’19(Invited) [308], ESWEK’19(Invited) [309], GLVLSI’19(Invited) [310], DATE’18 (Invited) [96], ICCAD’17 [76], DAC’16 [311] and DAC’15 [312].