COMPASS: A Compiler Framework for Resource-Constrained Crossbar-Array Based In-Memory Deep Learning Accelerators

Jihoon Park*, Jeongin Choe*, Dohyun Kim, Jae-Joon Kim
Seoul National University, Seoul, South Korea

***Abstract*—Recently, crossbar array based in-memory accelerators have been gaining interest due to their high throughput and energy efficiency. While software and compiler support for the in-memory accelerators has also been introduced, they are currently limited to the case where all weights are assumed to be on-chip. This limitation becomes apparent with the significantly increasing network sizes compared to the in-memory footprint. Weight replacement schemes are essential to address this issue. We propose *COMPASS*, a compiler framework for resource-constrained crossbar-based processing-in-memory (PIM) deep neural network (DNN) accelerators. *COMPASS* is specially targeted for networks that exceed the capacity of PIM crossbar arrays, necessitating access to external memories. We propose an algorithm to determine the optimal partitioning that divides the layers so that each partition can be accelerated on chip. Our scheme takes into account the data dependence between layers, core utilization, and the number of write instructions to minimize latency, memory accesses, and improve energy efficiency. Simulation results demonstrate that *COMPASS* can accommodate much more networks using a minimal memory footprint, while improving throughput by 1.78X and providing 1.28X savings in energy-delay product (EDP) over baseline partitioning methods.**

***Index Terms*—PIM, Accelerator, Neural Network, CNN, In-memory Computing, Compiler, Layer pipelining**

I. Introduction

Rapid evolution of Deep Neural Networks (DNNs) has fueled the demand for advanced computing architectures and accelerators to meet growing computational requirements. As DNNs continue to scale in complexity and size, conventional computing architectures, primarily Von Neumann, face significant challenges in coping with the escalating demand for computation and memory bandwidth. The traditional separation of processing and memory, inherent in Von Neumann architectures, introduces substantial inefficiencies, particularly in data movement and energy consumption. Recognizing these challenges, attention to Processing-In-Memory (PIM) architectures has been rapidly increasing as an alternative to traditional architecture [1–4]. Along with the architectural interest, IMC are being implemented in various technologies such as SRAM [5, 6], ReRAM [7–9], MRAM [10–12], etc. Recent prototype chips integrate multiple macros on chip to verify the effectiveness of PIM architectures. However, these prototypes still suffer from limited IMC memory capacity, regardless of which technology it uses, varying from a few hundred KBs to a few MBs and do not support networks that exceeds the size of the IMC footprint at all. Some includes tailored weight mapping and execution schemes but these approaches are not general with lack of compiler support. Existing compilers for digital neural processing units (NPUs) are not suitable for PIMs as they do not consider the PIM's inherent parallel MVM operation capability. PIM-aware compilers like PUMA [13] and PIMCOMP [3] have their primary focus on mapping all the weights on chip, but it is not possible to map large networks on chip when PIM memory footprint is constrained to tens of MBs at most.

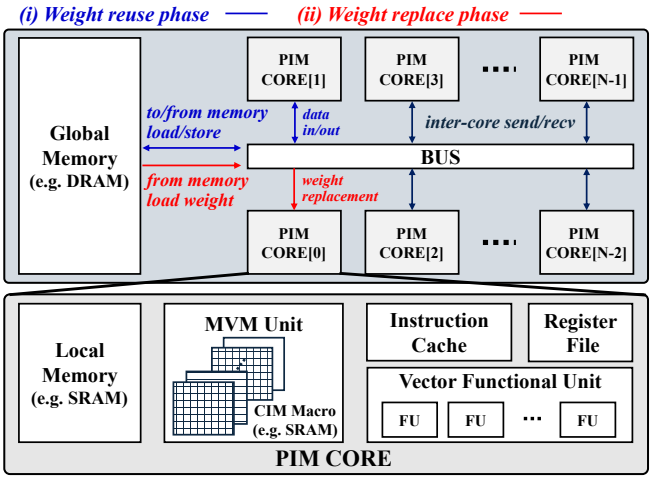

Fig. 1. In-memory DNN accelerator architecture with weight replacement

To address the issue, we introduce a novel compiler framework, *COMPASS*, designed for resource-constrained crossbar-based in-memory DNN accelerators. Our framework intelligently partitions the network into smaller and manageable units, optimizing on-chip resource utilization while maintaining balance across layers and partitions. Through this approach, we provide a pragmatic solution to the challenges posed by mapping large DNNs on PIM design with limited resources. Our contributions can be summarized as follows:

- We introduce a compiler framework for in-memory computing, which can support large DNN models when the required weight memory exceeds the CIM memory footprint, requiring communication with external memory.
- We propose a network partitioning method for general crossbar-array based in-memory accelerators by supporting weight reloading and multi-endpoint dependency checks between each partition.
- We propose a genetic algorithm (GA) to find the optimal

*These authors contributed equally to this work

978-3-9826741-0-0/DATE25/© 2025 EDAA

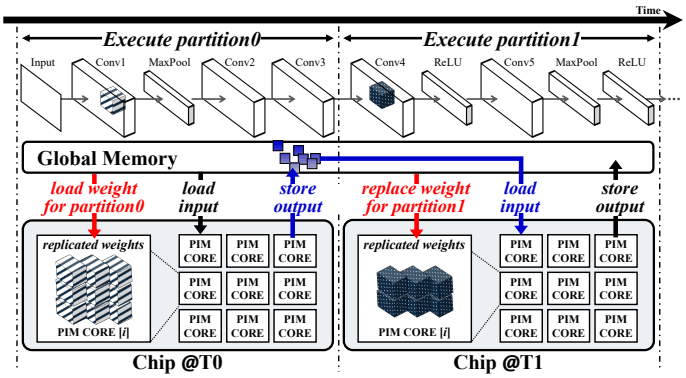

Fig. 2. Partitioned model execution. At T0, the first partition runs with weights loaded into PIM memory, inputs processed, and outputs stored in the global memory. The stored output becomes the input for the next partition at T1.

partitioning and a novel fitness function that optimizes for high throughput and low energy-delay product (EDP).

II. PIM Accelerator with Model Partitioning

A. Weight Replacement

We choose to use the abstract in-memory DNN accelerator architecture described in [3] as a PIM architecture template. Similar to previous works [1, 2, 13], the architecture defines a Macro-Core-Chip hierarchy as shown in Fig. 1. The chip consists of multiple PIM cores and a global memory, where each PIM core is connected via on-chip interconnect. Each core contains a matrix unit, a vector processing unit, an instruction memory for core control, and local registers and memory for storing intermediate results and activation features. The matrix unit consists of multiple crossbar-based CIM macros which perform matrix-vector multiplications in an efficient manner.

To address cases where a large DNN model does not fit within the CIM memory footprint, we introduce a weight replacement capability, as depicted in color in Fig. 1. During the weight reuse phase, a partition of the model is mapped and executed on chip, loading input and storing output in the process. The core then transitions to a weight replace phase, where weights are loaded from global memory and broadcast to the crossbars for writing. When the core enters a computation phase again, new inputs are loaded from global memory, and the generated results are stored back upon exiting the state.

B. Partitioned Model Execution

Fig. 2 shows the abstract view of the partitioned model execution. The execution within the partition is conducted in a pipelined manner, treating each partition as a model fully mapped on chip. The execution of different partitions is performed sequentially, with weights being replaced between each execution. Cores mapped to earlier layers complete execution and start weight replacement faster, enhancing the effective utilization of global memory bandwidth.

Another critical factor we consider is the weight replication. Weight replication is a widely known scheme to replicate the weights of the layers prior to pooling or striding to balance the throughput between pipelined layers. However, previous works [2, 3] assume that a DNN model fits entirely on a chip and they rather do a simple optimization to determine how to allocate the remaining in-memory computing cells for additional replication of layers. Considering weight replacement together with weight replication becomes a more complex joint optimization problem in which partitioning the model and replicating layers within the partition should both be considered for optimal performance.

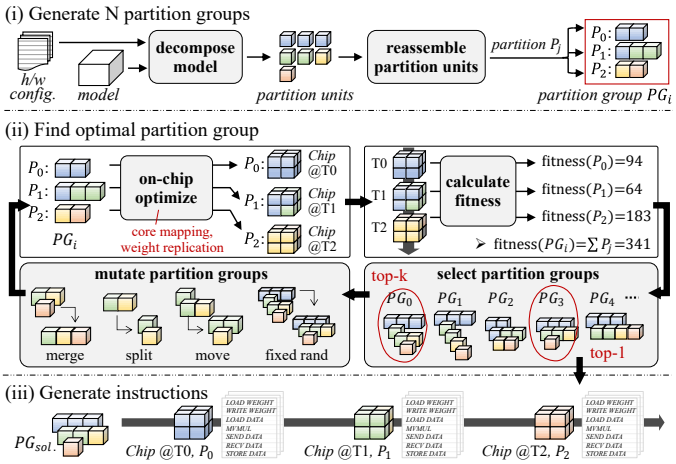

Fig. 3. *COMPASS* compiler framework overview

Additionally, we propose to execute each partition in a batched manner, where weight parameters of each partition are loaded and reused until a batch of input features are processed and saved to DRAM. Then, the weights are replaced for the next partition execution. Increasing the batch size improves the throughput and per-inference energy consumption. On the other hand, each sample has to wait for other samples in the same batch to finish before starting execution of its next partition, thereby increasing the end-to-end latency of the total execution. Therefore, the batch size should be kept relatively small to balance the throughput and the end-to-end inference latency.

III. Compiler Framework

A. Overview of COMPASS Framework

Fig. 3 shows the proposed *COMPASS* compiler framework which consists of three major components: partition generator, partition optimizer, and a scheduler. A user-specified hardware configuration (e.g. crossbar attributes, number of crossbars, core size, interconnect specification) and the network model is provided to the partition generator. Subsequently, the partition generator divides the model into the smallest units for core mapping (partition unit) and then recombines these units to generate initial model partitions (partition group). The partition optimizer uses the initial partition groups to estimate their performance considering the weight replication and core mapping and iteratively optimizes the partition groups. We introduce the *COMPASS* algorithm which is a GA algorithm to select and mutate the partition groups. After multiple generations, it identifies the optimal partition. Finally, the scheduler generates necessary instructions for model execution on each core, including the weight writes and activation load/store instructions between partitions as described in Sec. II-A.

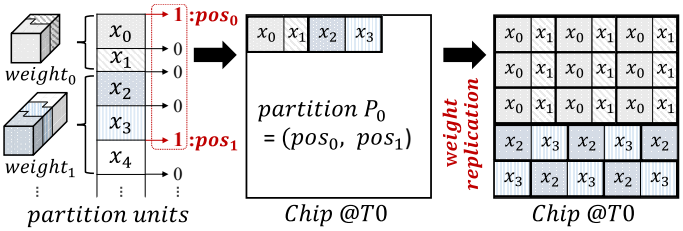

Fig. 4. Model decomposition and partition generation

B. Partition Generation

Fig. 4 illustrates the process of model decomposition and partition generation. First, weight matrices are divided along the output dimension into partition units x_i's. Each unit is sized to fit within the IMC memory footprint of a single core, serving as a minimum granularity for partitioning. Then, a span of consecutive partition units represented by a pair of position, $\{x_i \mid pos_0 \leq i < pos_1\}$, are grouped into the same partition P_0. As the last step, weights are replicated in each partition as necessary. The following conditions should be met for valid partitioning and replicating:

1. A partition unit cannot be of size bigger than the in-memory footprint of a single core.
2. The partition units that originate from a single kernel share their replication counts.
3. The total size of the replicated units cannot exceed the chip memory constraint.

Through this procedure, the Conv/Linear layers can be flexibly assigned to their respective partition according to the model and hardware constraint. Early layers with small kernel sizes can be mapped together inside a single partition with replication, while later layers with bigger kernel size can be split into multiple partitions. A validity map is constructed for efficient partition selection. The layers that cannot be mapped on crossbar arrays are dealt after the partition is generated.

1) Validity Map: If partition positions are selected randomly, the likelihood of producing a valid result becomes low and multiple iterations are required to find a valid solution, especially with a large model size and a small in-memory computing cell capacity. Therefore, instead of randomly selecting positions to generate partitions, we pre-calculate a validity map which marks the possible end position when a starting position for a partition is given. Using the validity map, we can ensure that every partition is generated within the chip's constraints. Fig. 5 shows the validity map for different model and chip sizes, indicating whether a partition defined by two positions (x_i, x_j) is valid. M represents the number of partition units after model decomposition in each case. We iteratively select partition positions within the valid range, taking into account the positions previously selected. Note that with a bigger number of weight parameters and a smaller in-memory computing cell capacity (towards the lower right in Fig. 5), the invalid portion of the validity map becomes larger.

2) Non-crossbar-mapped Layers: After the partitions for layers that are to be mapped on crossbar are determined, other layers are taken into account. We construct a directed acyclic graph (DAG) for the decomposed model, with partition units

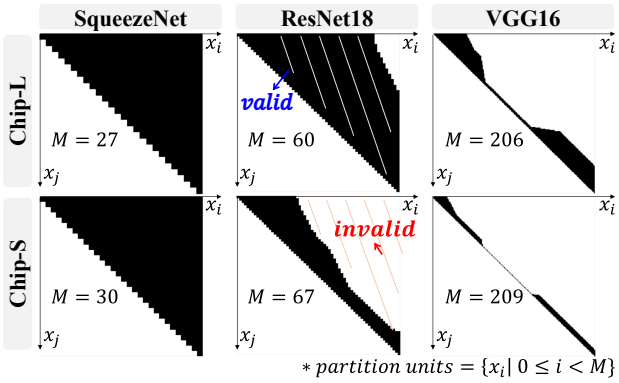

Fig. 5. Partition validity map. Chip-S and Chip-L represent the small and large chip configurations, as detailed in Table I. The models increase in size from SqueezeNet to VGG16, with details provided in Table II.

serving as nodes. As the remaining non-crossbar-mappable layers such as batch normalization and activation layers are closely coupled with the prior Conv/Linear layer, we traverse back the layer dependence graph to place the layers in the same partition.

3) Memory Access Management: During an execution of a partitioned graph, the entry nodes require load from the global memory, and the exit nodes have to store the intermediate features back to the global memory for next partition execution. Thus these nodes are marked with load/store attributes along with the respective data transfer sizes. Unlike a fully on-chip model, which has a single entry and exit node, each partition can have multiple entry and exit nodes. For example, a ResNet model with a residual connection that is not fully contained within a partition would have multiple exit nodes. The load/store attributes are used to consider DRAM access latency in performance estimation, memory allocation during instruction scheduling.

C. COMPASS Algorithm

Algorithm 1 outlines the steps of the *COMPASS* algorithm. Each gene represents a partition and each chromosome represents a partition group. The partition generator generates a predefined number of initial partition groups, denoted as Π_0. In each generation g, a predefined ratio of population is kept (n_{sel}) and then mutated to generate the remaining population (n_{mut}) based on their fitness (Sec. III-C1). For partition groups selected for mutation, score for each partition is calculated (Sec. III-C2) and the group is mutated by one of the four mutation schemes (Sec. III-C3). At the end of the generation ($g = G$), the final partition group ϕ is selected based on its fitness.

1) Partition Group Fitness: The model is optimized by its fitness (power or throughput) as specified by the user. The partition group fitness (PGF) is calculated by summing up all of its partition's fitness. Since each partition is a sub-model which is mapped fully on chip, we can use previous optimization methods [3, 13] to optimize a partition. We modify PIMCOMP [3]'s scheme for partition optimization by considering layer dependence and memory accesses as described in Sec. III-B.

2) Partition Score and Selection: We define a partition score to evaluate the performance of a partition relative to the overall

Algorithm 1 *COMPASS* algorithm(G, N, n_{sel}, n_{mut})

$\Pi_0 \leftarrow generatePGs(N)$
for $g \leftarrow 0, G$ **do**
  for all $PG_i \in \Pi_g$ **do**
    $PGF_i \leftarrow$ CALCULATEPGFITNESS(PG_i)
  end for
  $\Pi_g \leftarrow sort_{asc}(\Pi_g; \text{PGF})$
  $\Pi_{sel} \leftarrow takeFirstN(\Pi_g, n_{sel})$
  $\Pi_{mut} \leftarrow takeRandomN(\Pi_{sel}, n_{mut})$
  for all $PG_j \in \Pi_{mut}$ **do**
    for all $P_k \in PG_j$ **do**
      $R_k \leftarrow$ CALCULATEPARTITIONSCORE(P_k)
    end for
    $PG_j \leftarrow sort_{asc}(PG_j; \text{R})$
    $P_{mut} \leftarrow takeLastN(PG_j, 1)$
    $PG_j \leftarrow mutatePG(PG_j, P_{mut})$
  end for
  $\Pi_{g+1} \leftarrow \Pi_{sel} \cup \Pi_{mut}$
end for
$\Pi_G \leftarrow sort_{asc}(\Pi_G; \text{PGF})$
$\phi \leftarrow takeFirstN(\Pi_G, 1)$
return ϕ

population. Worse performing partition or a partition pair is selected for mutation. For a partition $P = \{x_i \mid a \leq i < b\}$, the partition score R is defined as follows.

$$m(x_i) = \frac{f(P)}{|P|} \quad \text{where } x_i \in P,$$

$$\overline{F}[p,q] = \mathbb{E}\left[\sum_{i=p}^{q-1} m(x_i)\right], \; R = \frac{f(P)}{\overline{F}[a,b]}$$

First, the partition unit fitness, $m(x)$, is defined as its residing partition's fitness ($f(P)$) divided by the number of partition units in the partition ($|P|$). This is further used to describe the expected fitness of the partition units' span, $\overline{F}[p,q]$. For each partition unit in the span $[p,q]$, their fitness is summed up and the expectation over the population Π. The partition score R is given as the ratio between partition's fitness and the expected fitness over the same span.

The score effectively captures whether the selected partition of an individual is performing well or not compared to other individuals in the pool. If there exists a partition which could potentially perform better if partition units are better partitioned, the $m(x)$'s would be relatively larger than other individuals where the same partition units reside in better performing partitions. Therefore, optimizing against the defined partition score would provide pressure to partition better.

3) Mutation: For n_{mut} partition groups in each generation, single or a pair of partitions are selected according to the partition score and are mutated with one of the four mutation schemes. *Merge* selects two neighboring partitions and merges them into a single partition. We evaluate the relative partition score of consecutive partition pairs to select the worst-performing pair to perform the merge. This method effectively removes small partitions that are inefficient. *Split* transforms a selected partition into two randomly split partitions. This method removes ill-performing partitions holding many layers, suffering from low replication value. *Move* moves a partition unit between two neighboring partitions. This method adjusts the total fitness in a meticulous way, by searching for an optimal partitioning position for neighboring partitions. *FixedRandom* fixes a partition with best fitness and all other partitions are randomly generated. This guarantees that new individuals for the next generation are highly random and do not fall into a local optimum.

TABLE I
HARDWARE CONFIGURATION

Component	Parameters	Specification	Power(mW)
VFU	# per core	12	22.8
Local Memory	# per core	64kB	18.0
Control Unit	# per core	-	8.0
DRAM	config.	LPDDR3 8GB	trace-based

Chip	# Cores	# Crossbar/Core	Capacity(MB)	Power(W)
S	16	9	1.125	1.57
M	16	16	2.0	2.80
L	36	16	4.5	6.30

TABLE II
NETWORK MODEL AND COMPILER SUPPORT

Network	Linear.(MB)	Conv.(MB)	Total(MB)	Prev.	**Ours**
VGG16	58.95	7.02	65.97	✗	✔
ResNet18	0.244	5.324	5.569	✗	✔
SqueezeNet	0.0	0.58725	0.58725	✓	✔

IV. EVALUATION

A. Experiment Setup

1) Hardware Details: We adopt the hardware architecture of PIMCOMP [3] and PUMA [13] but with tighter resource constraints. We adopt the parameters used in PIMCOMP for core design and scale them into 16nm technology, including the power information of VFUs, control units, data and instruction memory. We use a bus interconnect to connect the PIM cores. We adopt a 256 x 256 crossbar array and calculate the power consumption using the energy breakdown from the 16nm IMC-SRAM prototype by Jia et al. [5]. The write power is directly taken from the prototype. The inference power is estimated by adding the ADC power and the power of remaining components which are scaled with respect to the number of wordlines. We model the DRAM energy by generating a memory trace from the scheduled instruction and feeding it into DRAMsim3 [14]. To demonstrate the generality of our approach, we evaluate on three chip configurations, "S," "M," and "L," with varying memory footprint sizes. These sizes are chosen based on the fact that existing chip prototypes across various technologies [3, 5, 7–9, 11], typically exhibit an in-memory footprint up to a few megabytes or less.The hardware configurations are summarized in Table I.

2) Benchmark and Baselines: We evaluate three representative CNN networks with varying model sizes: VGG16, ResNet18, and SqueezeNet. The parameter sizes of the networks are given in Table II. We assume 4b weight and activation precision to faithfully model power consumption based on a recent CIM array which incorporates 4b quantization [5]. We evaluate the networks across various chip configurations and

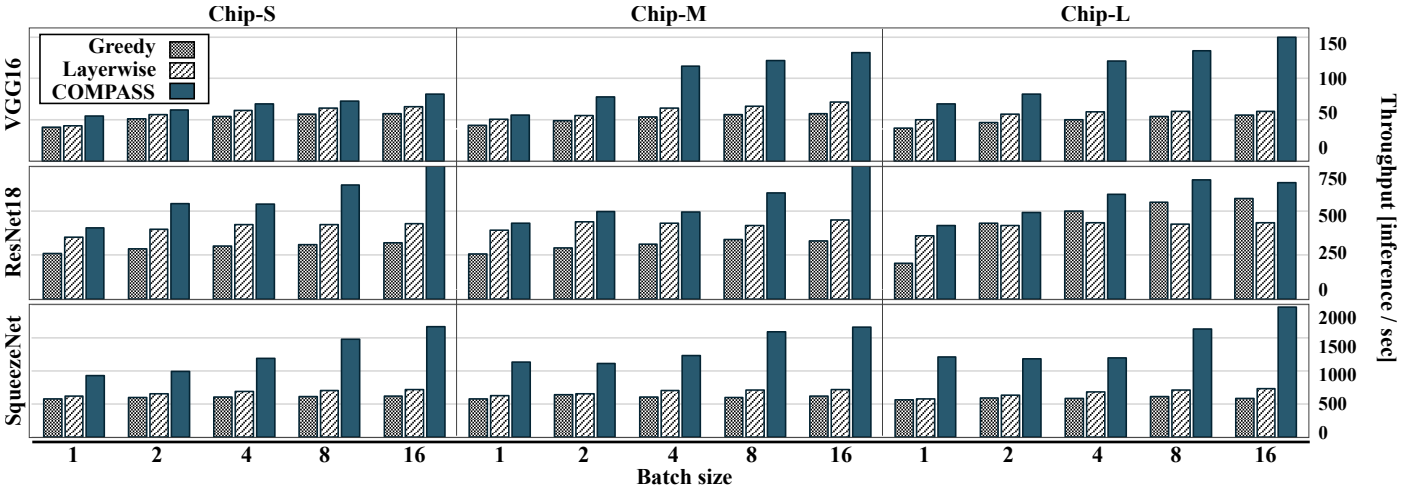

Fig. 6. Throughput comparison

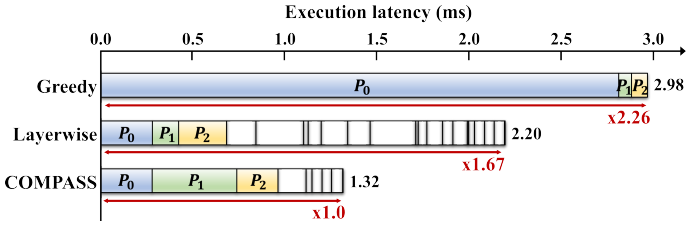

Fig. 7. Latency breakdown for each partition. The per-partition latencies for each schemes are shown, with the first three partitions highlighted for easier comparison.

batch sizes, labeling each evaluation as *"Network-ChipConfig-BatchSize"* (e.g. *"ResNet18-S-4"*). Note that existing compiler methods can only map SqueezeNet in resource-constrained chips, while *COMPASS* allows all three models.

We compare *COMPASS* results with two baseline partitioning schemes: *greedy*, and *layerwise*. The *greedy* scheme attempts to pack in as many consecutive layers as possible, by iterating the nodes and tracking the remaining memory footprint. The *layerwise* scheme maps a single Conv/Linear layer at a time, then maps the trailing non-Conv/Linear nodes together with its producer Conv/Linear nodes. All partitioning schemes, including ours, are implemented by extending the open-source PIMCOMP framework [3]. We implement code for the *COMPASS* algorithm with an enhancement to PIMCOMP's latency estimator, as the original estimator was designed for non-partitioned design and did not consider weight load, intermediate data load/store.

3) COMPASS GA algorithm Parameters: We maintain a population of 100 for 30 generations, using a selection size (n_{sel}) of 20 and a mutation size (n_{mut}) of 80. We also adopt an early stopping mechanism. The mutation schemes are selected with the same probability.

B. Experimental Results

1) Latency and Throughput: Fig. 6 shows the result of inference throughput of *COMPASS* under different workloads and memory constraints. *COMPASS* achieves 1.78X higher throughput than the baseline methods. Specifically, *COMPASS* outperforms the *greedy* scheme by 1.80X, 1.71X, and 2.24X and *layerwise* scheme by 1.56X, 1.31X, and 1.98X in VGG16, ResNet, and SqueezeNet respectively.

We can observe different trends in performance gain with differing chip sizes and networks. Performance gain in VGG execution for a large chip is much higher than the one for a smaller chip. This is due to the fact the VGG network has large channel dimensions, making it hard to fit in many layers together in a small chip. In such a case, possible pipeline depth and replication numbers reduce, thereby making both greedy and layerwise method perform similar. The optimal balance between pipeline and replication does not differ much either. On the other hand, a larger chip size can have bigger chances of tweaking replication numbers and the pipeline depth of each partition. ResNet18 and SqueezeNet models have relatively smaller layers and a small chip configuration is enough to exploit *COMPASS*'s optimization ability. In case of "ResNet18-L", we see that all methods suffer a certain amount of performance degradation compared to smaller chip configuration. As we increase the batch size, the weights are written once and reused over multiple samples, effectively making throughput higher. Our methods outperform the baselines across all typical batch sizes.

Fig. 7 shows the latency results for each partition during execution of *"ResNet18-M-16"*. Different colored portion of the graph indicates different partition's execution time. While *COMPASS* achieves 2.26X and 1.67X speed-up compared to greedy and layerwise partitioning respectively.

Greedy partitioning maps many layers in the earlier partition and does not exploit weight replication favorably, resulting in high latency. Its first partition, P_0, occupies over 95% of the total execution time. *Layerwise* partitioning maps a single Conv/Linear layer on a partition and exploits more weight replications. However, this increases DRAM access, as more partitions require intermediate features to be moved in and out of DRAM between each partition. In contrast, *COMPASS* can map multiple layers within a single partition, reducing this overhead.

2) Energy-Delay-Product: Fig. 8 shows the inference energy and energy-delay product (EDP) per sample of *COMPASS* for different batch sizes in a *"ResNet18-S"* configuration. Since

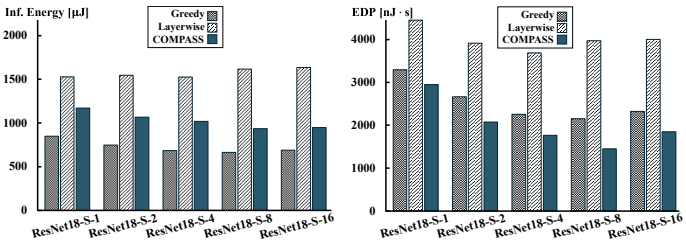

Fig. 8. Inference energy and energy-delay product

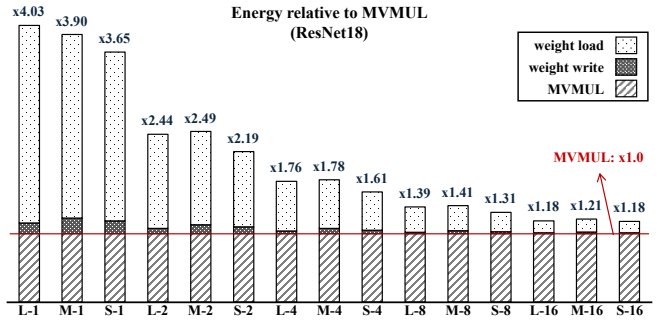

Fig. 9. Energy consumption of weight writes and loads relative to MVMUL for different chip and batch sizes. For example, "M-4" denotes chip "M" with a batch size of 4.

we optimize for latency, we utilize more replication in each partition, thereby limiting the pipeline depth compared to *greedy* partition scheme. This requires more data communication with DRAM, increasing the energy consumption required per inference as shown in Fig. 8. However, we observe that *COMPASS* is more efficient in terms of joint optimization of power and latency. We outperform *greedy* and *layerwise* schemes by 1.28X and 2.08X in EDP results on average. This is mainly because limited replication in *greedy* scheme incurs many stalls during the execution making it less optimized for latency.

3) Effect of different batch sizes: As discussed in Sec II-B, an appropriate number of batch size is important to amortize weight replacement overhead. Fig. 9 plots the energy consumption of the weight writes and loads on different batch sizes normalized to matrix vector multiplication energy consumption. With a batch size of 1, the weight load energy dominates over compute energy. With a batch size of 16, replacement overhead is sufficiently amortized.

4) GA Fitness Convergence: Fig. 10 shows the evolution of the population's fitness across generations under a *COMPASS "ResNet18-M-16"* optimization. For clarity, a random one-third of the population is selected for visualization in each generation. The population selected for the next generation (Π_{sel}) is represented by "O" markers, while the mutated population (Π_{mut}) is represented by "+" markers. We can observe that GA algorithm makes the population steadily evolve into the selected one, finding an optimal fit.

The different partition groups are represented by various colors based on the number of partitions. Initially, the population tends to start with fewer partitions, and by the 9th or 10th generation, an optimal number of partitions is typically reached. From this point, most partition groups continue to be refined within the same partition count, yielding improved fitness as

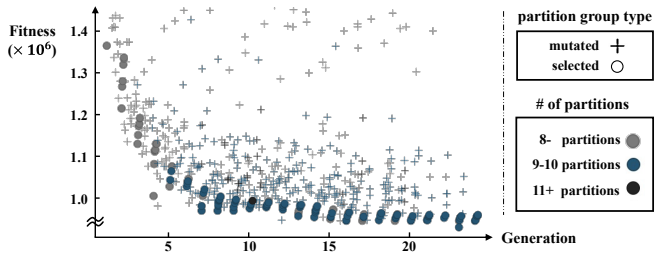

Fig. 10. Evolution of partition groups and their number of partitions over generations

the GA algorithm explores the design space.

V. Discussion

A. Challenges of Adopting Traditional Compiling Methods for Digital GPU/NPUs to PIM Architectures

While both Processing-In-Memory (PIM) cores and digital processing engines, such as GPU SM cores and other NPUs, serve as fundamental processing units for AI accelerations, their operational paradigms differ significantly. PIM cores perform computations directly where the data is stored, reducing data movement but incurring higher write costs. Also, data movement of PIM cores are done in crossbar granularity. In contrast, GPU SM cores and digital accelerators rely on fast data movement between the cores and the local buffer. These differences create challenges when applying traditional compiling methods to PIM architectures. This involves managing high memory write costs and ensuring efficient partitioning and scheduling within PIM's limited resources. As a result, new compilation techniques are needed to fully exploit PIM's unique capabilities while overcoming these constraints.

B. Applicability to Different PIM Technologies

While the current work is evaluated on an SRAM-based architecture, this choice is due to the maturity of in-memory SRAM technology, making it an easier target for system-level evaluation. However, our approach can also be extended to emerging non-volatile memory (eNVM) technologies such as ReRAM and MRAM. Although ReRAM is limited by its write endurance, our method aligns well with this constraint by minimizing the number of weight rewrites. In the case of MRAM, which has higher write latency and energy consumption, we can parameterize the crossbar properties as part of the hardware configuration and optimize weight replacement accordingly.

VI. Conclusion

This work develops *COMPASS*, a compiler framework for PIM-based CNN accelerator in which a given model does not fit entirely on chip. *COMPASS* generates an optimal model partition where each partition fits on chip, thereby enabling automatic execution of larger neural networks without the need for manual model decomposition. To the best of our knowledge, *COMPASS* is the first compiler framework to consider communication with external memory for analog in-memory computing hardware. Compared to naive partitioning schemes, *COMPASS*'s partitioning scheme achieves higher throughput and better EDP in diverse workload settings.

ACKNOWLEDGEMENT

This work was supported in part by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No.RS-2023-00208606, NeuroHub+: Scheduler and Simulator for General In-Memory Neural Network Accelerators, IITP-2023-RS-2023-00256081: artificial intelligence semiconductor support program to nurture the best talents), BK21 FOUR program at Seoul National University, and ISRC at Seoul National University. The EDA tool was supported by the IC Design Education Center(IDEC). (Corresponding Author: Jae-Joon Kim).

REFERENCES

[1] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar, "Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 14–26, 2016.

[2] L. Song, X. Qian, H. Li, and Y. Chen, "Pipelayer: A pipelined reram-based accelerator for deep learning," in *2017 IEEE international symposium on high performance computer architecture (HPCA)*. IEEE, 2017, pp. 541–552.

[3] X. Sun, X. Wang, W. Li, L. Wang, Y. Han, and X. Chen, "Pimcomp: A universal compilation framework for crossbar-based pim dnn accelerators," in *2023 60th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2023, pp. 1–6.

[4] X. Qiao, X. Cao, H. Yang, L. Song, and H. Li, "Atomlayer: A universal reram-based cnn accelerator with atomic layer computation," in *Proceedings of the 55th Annual Design Automation Conference*, 2018, pp. 1–6.

[5] H. Jia, M. Ozatay, Y. Tang, H. Valavi, R. Pathak, J. Lee, and N. Verma, "15.1 a programmable neural-network inference accelerator based on scalable in-memory computing," in *2021 IEEE International Solid-State Circuits Conference (ISSCC)*, vol. 64. IEEE, 2021, pp. 236–238.

[6] S. Yin, B. Zhang, M. Kim, J. Saikia, S. Kwon, S. Myung, H. Kim, S. J. Kim, M. Seok, and J.-s. Seo, "Pimca: A 3.4-mb programmable in-memory computing accelerator in 28nm for on-chip dnn inference," in *2021 Symposium on VLSI Technology*. IEEE, 2021, pp. 1–2.

[7] C.-X. Xue, J.-M. Hung, H.-Y. Kao, Y.-H. Huang, S.-P. Huang, F.-C. Chang, P. Chen, T.-W. Liu, C.-J. Jhang, C.-I. Su *et al.*, "16.1 a 22nm 4mb 8b-precision reram computing-in-memory macro with 11.91 to 195.7 tops/w for tiny ai edge devices," in *2021 IEEE International Solid-State Circuits Conference (ISSCC)*, vol. 64. IEEE, 2021, pp. 245–247.

[8] W.-H. Huang, T.-H. Wen, J.-M. Hung, W.-S. Khwa, Y.-C. Lo, C.-J. Jhang, H.-H. Hsu, Y.-H. Chin, Y.-C. Chen, C.-C. Lo *et al.*, "A nonvolatile al-edge processor with 4mb slc-mlc hybrid-mode reram compute-in-memory macro and 51.4-251tops/w," in *2023 IEEE International Solid-State Circuits Conference (ISSCC)*. IEEE, 2023, pp. 15–17.

[9] J.-M. Hung, Y.-H. Huang, S.-P. Huang, F.-C. Chang, T.-H. Wen, C.-I. Su, W.-S. Khwa, C.-C. Lo, R.-S. Liu, C.-C. Hsieh *et al.*, "An 8-mb dc-current-free binary-to-8b precision reram nonvolatile computing-in-memory macro using time-space-readout with 1286.4-21.6 tops/w for edge-ai devices," in *2022 IEEE International Solid-State Circuits Conference (ISSCC)*, vol. 65. IEEE, 2022, pp. 1–3.

[10] S. K. Roy, H.-M. Ou, M. G. Ahmed, P. Deaville, B. Zhang, N. Verma, P. K. Hanumolu, and N. R. Shanbhag, "Compute sndr-boosted 22-nm mram-based in-memory computing macro using statistical error compensation," *IEEE Journal of Solid-State Circuits*, 2024.

[11] H. Cai, Z. Bian, Y. Hou, Y. Zhou, Y. Guo, X. Tian, B. Liu, X. Si, Z. Wang, J. Yang *et al.*, "33.4 a 28nm 2mb stt-mram computing-in-memory macro with a refined bit-cell and 22.4-41.5 tops/w for ai inference," in *2023 IEEE International Solid-State Circuits Conference (ISSCC)*. IEEE, 2023, pp. 500–502.

[12] P. Deaville, B. Zhang, and N. Verma, "A fully row/column-parallel in-memory computing macro in foundry mram with differential readout for noise rejection," *IEEE Journal of Solid-State Circuits*, 2024.

[13] A. Ankit, I. E. Hajj, S. R. Chalamalasetti, G. Ndu, M. Foltin, R. S. Williams, P. Faraboschi, W.-m. W. Hwu, J. P. Strachan, K. Roy *et al.*, "Puma: A programmable ultra-efficient memristor-based accelerator for machine learning inference," in *Proceedings of the twenty-fourth international conference on architectural support for programming languages and operating systems*, 2019, pp. 715–731.

[14] S. Li, Z. Yang, D. Reddy, A. Srivastava, and B. Jacob, "Dramsim3: A cycle-accurate, thermal-capable dram simulator," *IEEE Computer Architecture Letters*, vol. 19, no. 2, pp. 106–109, 2020.