This is a summary of the original paper, entitled "CHOPPER: A Compiler Infrastructure for Programmable Bit-serial SIMD Processing Using Memory in DRAM" which appears in HPCA 2023 [9]

Towards Programmable and Efficient In-Memory Accelerators[*]

Xiangjun Peng[δ] Yaohua Wang[$] Ming-Chang Yang[δ]
[δ]*The Chinese University of Hong Kong* [$]*National University of Defense Technology*

ABSTRACT

The programmability and efficiency of In-Memory Accelerators (or Processing-In-Memory) are still major obstacles for adoption, though there are diverse forms of In-Memory Accelerators (including Processing-Using-Memory and Processing-Near-Memory). By focusing on Bit-serial SIMD Processing Using DRAM (PUD), our HPCA-29 paper outlines and aims to address (1) the programmability issue, by enabling automatic transformation from naturally-expressive codes to "SIMD-Within-A-Register"-style codes; and (2) the efficiency issue, by resolving the granularity mismatch between processing and storing data operands; and the under-utilization of Memory-Level Parallelism, caused by such a mismatch.

Our HPCA-29 paper introduces CHOPPER, the first compiler infrastructure to make Bit-serial SIMD PUD architectures more progammable and efficient. For the programmability, CHOPPER delivers the first dataflow programming interface on PIM architectures, by automating *bit-slicing* compilation to generate "SIMD-Within-A-Register"-style codes. For the efficiency, CHOPPER introduces new optimizations to minimize data movements on Bit-serial SIMD PUD. Along with the above designs, CHOPPER also reveals the potential to exploit Memory-Level Parallelism in PIM architectures. Our HPCA 2023 paper also covers how CHOPPER design can be potentially beneficial for other types of Processing-Using-Memory, as well as Processing-Near-Memory accelerators. Our evaluations suggest that CHOPPER can significantly improve the performance with a huge reduction of Lines-of-Codes, compared with codes using the state-of-the-art hands-tuned methodology on three state-of-the-art Bit-serial SIMD PUD architectures.

1. SUMMARY

1.1 Problem: Programmability & Efficiency on Bit-serial SIMD PUD

Processing-In-Memory (PIM), as recently-bloomed practices on commercial DRAM chips [1, 5, 7], receives a considerable amount of attention due to its potentials to mitigate data movement bottlenecks. However, Processing-Near-DRAM (PND) architectures fail to exploit the maximum level of internal DRAM bandwidth. A new approach is to leverage in-DRAM analog computation for bitwise (or complex) operations [3, 4, 10, 11]. Such architectures, denoted as Bit-serial SIMD Processing-Using-DRAM (PUD) architectures, (1) take each DRAM bank as an ultra-wide SIMD processing unit (i.e. 65,536); (2) transpose individual data operands vertically in each SIMD lane (i.e. DRAM bitline in Bit-serial SIMD PUD architectures) [4]; and (3) support basic logic (and complex) logic operations in a SIMD manner.

[*]The full paper is: "CHOPPER: A Compiler Infrastructure for Programmable Bit-serial SIMD Processing Using Memory in DRAM" [9], X. Peng and et al, HPCA 2023, 24 March 2023. The full paper can be downloaded from `https://ieeexplore.ieee.org/document/10071070`.

Different from PND architectures, (Bit-serial SIMD) PUD architectures are more difficult to be programmable and efficient. Our HPCA 2023 paper details three outstanding challenges on existing Bit-serial SIMD PUD architectures from the perspectives of programmability and efficiency. These challenges include: (1) assembly programming interfaces for the poor programmability; (2) extra intra-subarray data movements for the low efficiency, when data can fit within DRAM subarrays; and (3) (potential) data spilling for huge overheads (or even slowdowns).

❶ The first challenge is Assembly Programming Interfaces for Poor Programmability. Existing Bit-serial SIMD PUD architectures (e.g. [4, 10, 11]) only support assembly-like interfaces for programmers to use, by directly exposing their instructions as intrinsic functions. The state-of-the-art programming interface in SIMDRAM [4]) is in assembly; and SIMDRAM [4] synthesizes data preparation and computation on Bit-serial SIMD PUD architectures in multi-bits. This enforces programmers to (1) handle memory allocation explicitly; (2) rewrite all codes in the "SIMD-Within-A-Register" style; and (3) consider how to exploit parallelism of different DRAM components.

❷ The second challenge is Extra Intra-subarray Data Movements for Low Efficiency. Existing Bit-serial SIMD PUD architectures can not minimize intra-subarray data movements [3, 10, 11]: this refers to data movements between a pre-assigned region for analog computation and the rest. In the state-of-the-art programming abstractions [4], there is a granularity mismatch between full-size storing (e.g. word-size, or byte-size) and bitwise (i.e., 1-bit) processing on these architectures. Therefore, this prevents the minimization of intra-subarray data movements, as well as redundant buffering for (parts of) data operands.

❸ The third challenge is (Potential) Data Spilling for Huge Overheads. No existing work (e.g. [4, 10, 11]) considers such an issue, and assume all workloads only require a constant amount of DRAM rows to buffer either input data or intermediate data; also, existing work also assumes all data can always fit within a DRAM subarray. Such an issue can be particularly serious on Bit-serial SIMD PUD architectures, because (A) Bit-serial SIMD PUD architectures have a ultra-wide SIMD width, and the vectorization for it amplifies the issue; and (B) inter-bitline communications, namely DRAM bitlines, are forbidden since the storage functionality of DRAM needs to be preserved. Hence, it is rationale to consider the limit of a DRAM subarray can be hit.

The goal of our HPCA-29 paper is to address the above issues on Bit-seiral SIMD PUD architectures. To achieve this, our paper describes a new compiler infrastructure, CHOPPER, to make Bit-serial SIMD PUD architectures more programmable and efficient. CHOPPER ❶ provides a synchronous dataflow programming interface; ❷ automatically transforms synchronous dataflow codes into heavily-optimized codes; and ❸ proposes a new set of optimizations.

1.2 Solution: CHOPPER Compiler

Our CHOPPER compiler has following three key features.

1. Synchronous Dataflow Programming. The CHOPPER programming interface is a synchronous dataflow programming interface, which (1) automates explicit memory allocation via the whole-program analysis; and (2) generates "SIMD-Within-A-Register"-style codes, as demanded in Bit-serial SIMD PUD architectures.

2. A Compilation Abstraction for Exploiting Memory-Level Parallelism. CHOPPER compiler introduce a new compilation abstraction called "VIRtual COde Emitter", to improve the exploitation of Bank-Level Parallelism on Bit-serial SIMD PUD architectures.

3. New Optimizations to Minimize Data Movements. CHOPPER propose new Optimizations for Bit-Sliced codes (OBS), and implement them in the CHOPPER compiler to improve the efficiency, for less data movements on Bit-serial SIMD PUD architectures.

1.2.1 Programming Interface and Compiler

An overview of CHOPPER is explained in detail in our HPCA-29 paper [9]. Here, we briefly describe them.

Programming Interface. The CHOPPER programming interface inherits from the Usuba Programming Language [8], with extra engineering efforts for Bit-serial SIMD PUD architectures. Figure 1a and 1b gives a comparison.

Compiler. The CHOPPER compiler consists of the front-end and the back-end. The front-end of the CHOPPER compiler translates naturally-expressive codes into C/C++ codes for a single subarray of Bit-serial SIMD PUD architectures. This automates explicit memory allocation and generates *bit-sliced* codes. The back-end of the CHOPPER compiler translates *bit-sliced* codes in C/C++ into assembly instructions for Bit-serial SIMD PUD architectures. An iterative use of the back-end broadcasts C/C++ codes to different subarrays.

1.2.2 VIRtual COde Emitter (VIRCOE)

CHOPPER provides a new middleware called VIRCOE, to exploit the Bank/Subarray-Level Parallelism. This is because code permutation affects the exploitation significantly. The key idea is to aggressively overlap data transfer (i.e. the activation before READ/WRITE) with computation (i.e. the Triple-Row Activation) on Bit-serial SIMD PUD architectures. Note that CHOPPER also supports VIRCOE to be reconfigured for subarray-level parallelism [6].

1.2.3 Optimizations for Bit-Sliced codes (OBS)

CHOPPER incorporates three new optimizations. ❶ The first optimization reorders and aggregates *bit-sliced* operations based on the data dependency, to minimize the number of rows for buffering intermediate operands. ❷ The second optimization aggressively supports bit-level data reuse rather than operand-level. ❸ The third optimization builds upon the first two: for any "one-shot" bitslices following the "Store-Copy-Compute" pattern, eliminates "copy" instructions can be eliminated by directing data to the B-group.

1.2.4 Applicability to Other PIM

The design philosophy of CHOPPER is expected to be useful for other PIM architectures, because (1) CHOPPER inherits a generalization from *bit-slicing* to *operand-slicing*, which can be suitable for other PUM architectures; (2) *operand-slicing* can improve the throughput of PND architectures.

1.2.5 Experimental Results

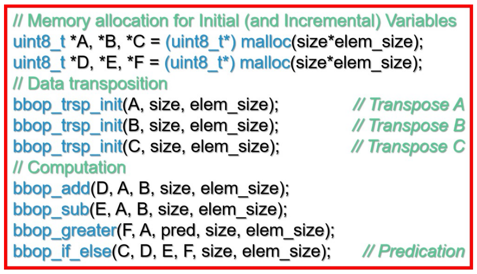
We evaluate CHOPPER by hosting it on three state-of-the-art Bit-serial SIMD PUD architectures. We compare CHOPPER-generated codes against the state-of-the-art hands-tuned codes. Figure 2 summarizes the results: averaged across 16 real-world workloads, CHOPPER achieves considerable performance gains on Ambit [10], ELP2IM [11] and SIMDRAM [4], compared with hands-tuned codes using the state-of-the-art methodology [4] for Bit-serial SIMD PUD architectures. These performance benefits also accompany wit a great reduction of Lines-of-Codes (LoC) in CHOPPER (i.e. by 4.3X less LoCs for one subarray, and $>10^3$X less for all subarrays in a rank). (See Table III in the paper).

We provide additional insights into *source benefits* of different optimizations in CHOPPER via a breakdown analysis, analyze tradeoffs when reconfiguring different sizes of a DRAM subarray, evaluate the impacts of subarray-level parallelism on Bit-serial SIMD PUD architectures. Our results shows that CHOPPER is very effective with different configurations of DRAM chips.

2. NOVELTY AND SIGNIFICANCE

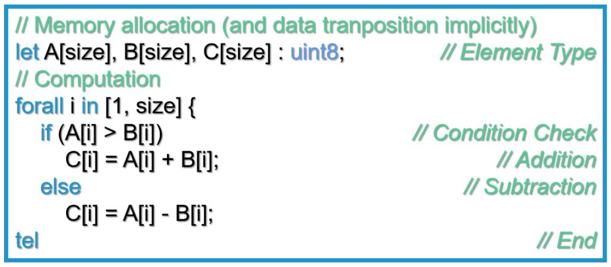
Recent industrial practices already attempt PND for their products. However, the roadmap to improve the programmability and the efficiency of these architectures (and PUD architectures) is still unclear. They are mostly likely to remain as huge obstacles for the wide adoption of PIM. By taking Bit-serial SIMD PUD architectures as a motivating example, Our HPCA-29 paper introduces a new compiler infrastructure for the better programmability and efficiency.

2.1 Contributions on Future Research



```
// Memory allocation for Initial (and Incremental) Variables
uint8_t *A, *B, *C = (uint8_t*) malloc(size*elem_size);
uint8_t *D, *E, *F = (uint8_t*) malloc(size*elem_size);
// Data transposition
bbop_trsp_init(A, size, elem_size);          // Transpose A
bbop_trsp_init(B, size, elem_size);          // Transpose B
bbop_trsp_init(C, size, elem_size);          // Transpose C
// Computation
bbop_add(D, A, B, size, elem_size);
bbop_sub(E, A, B, size, elem_size);
bbop_greater(F, A, pred, size, elem_size);
bbop_if_else(C, D, E, F, size, elem_size);    // Predication
```

(a) SIMDRAM Programming Interface [4].



```
// Memory allocation (and data tranposition implicitly)
let A[size], B[size], C[size] : uint8;       // Element Type
// Computation
forall i in [1, size] {
  if (A[i] > B[i])                           // Condition Check
    C[i] = A[i] + B[i];                      // Addition
  else                                       // Subtraction
    C[i] = A[i] - B[i];
tel                                          // End
```

(b) CHOPPER Programming Interface.

Figure 1: A comparative example of written codes in (A) SIMDRAM Programming Interface [4]; and (B) CHOPPER Programming Interface to perform packed addition and subtraction in a subarray of Bit-serial SIMD PUD architectures.

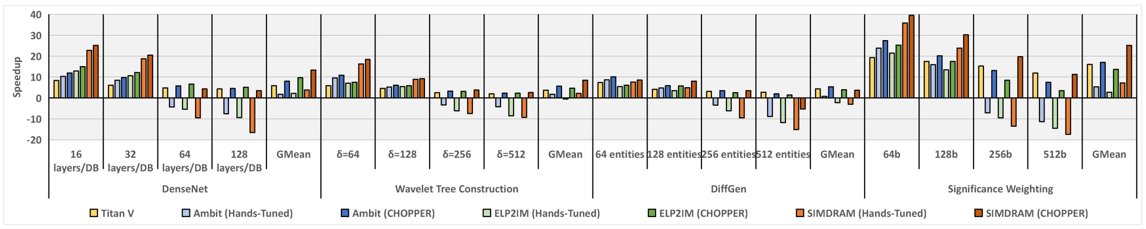


Figure 2: Speedup of all workloads over Intel multi-core for CHOPPER and the hands-tuned codes.

Our paper makes three major contributions which will likely have long-term impact on both industry and academia:

1. In-Memory Accelerators Can Be Programmable: our paper introduces the first example to support a (synchronous) dataflow programming interface on PIM architectures. Such an idea can be exploited for other PIM architectures with small migration. Our paper also clears the roadmap on how to make PIM architectures benefit from high-level programming: by leveraging another level of the indirection, it is expected to be feasible to incorporate PIM architectures in an accelerator manner. This is because the switches between the processor and the PIM accelerator can be easily achieved, and such a design philosophy can ignite more sophisticated designs, to provide portable compilers for PIM-integrated systems.

2. A New Solution Direction to Improve the Efficiency of In-Memory Accelerators: our paper demonstrates the first software-level automation, to enhance the exploitation of memory-level parallelism (MLP) on PIM architectures. Our paper also reveals the importance of code permutations for MLP exploitation, and experimentally demonstrates the potential benefits. Moreover, our paper demonstrates the applicability (and benefits) to adopt such an exploitation onto more aggressive MLP. This can motivate future work on architectural (and compilation) designs for this goal.

3. Granularity Mismatch between In-Memory Accelerators and Memory Chips: our paper is the first to identify the issue of *granularity mismatch between processing and storing operands* on PIM architectures. Our paper clears the performance issues from the granularity mismatch, which includes (A) data movements within PIM architectures; and (B) data spilling on PIM architectures. With the emergence of PIM architectures in practice, these two issues can motivate ad-hoc solutions (e.g., tight integration between PIM logic and memory chips) or unified solutions (e.g., new hardware-software co-designs to address these issues) as future work.

2.2 Long-Term Impacts on Industry

The importance of both the problem (programmabilty and efficiency) and the solutions presented in this work will increase in future systems. The importance of "high-level programming on PIM architectures" is likely to increase, as the emerging trends of PIM architectures in practice. The importance of "exploitation of MLP on PIM" and "bit-slicing on PIM architectures" would also increase, as the growing amount of industrial practices on improving the efficiency of In-Memory Accelerators. Moreover, the importance of "resolving the granularity mismatch" will increase, as the limited area budget for In-Memory Accelerators.

As a result, co-designs throughout architectural designs and software abstractions will likely be considered as one of suitable alternatives, to some throughput-demanded techniques under area-insufficient scenarios (e.g., DRAM chips) in future PIM-integrated systems: the need of *bit-slicing automation* on Bit-serial SIMD PUD architectures architectures (and other PND architectures) is such an example. To enhance the generality of PIM architectures in diverse set of workloads, the desired techniques are expected to involve supports of high-level programming on PIM architectures.

3. CONCLUSIONS

This paper proof-of-concepts that both programmability and efficiency of PIM architectures can be improved simultaneously, by rethinking the software stack for PIM-integrated systems. The compiler infrastructure proposed in this paper can be used/extended to benefit other PIM architectures, as well as new optimizations proposed in this work (for MLP exploitation and the minimization of data movements). Moreover, the design philosophy of our paper can yield future work on portable compilers for different PIM architectures.

Our HPCA paper presents a new set of problems and practical ideas for improving the programmability and efficiency of PIM architectures. Though there are a diverse set of PIM architectures, the authors assume that (Bit-serial) Processing Using DRAM shall be considered as the central role in Memory-Centric Computing. With the most recent breakthrough on (Bit-serial) Processing Using DRAM [2] (i.e., demonstrates the feasibility to simultaneously activate up to 32 DRAM rows in existing commercial DRAM chips), the future is bright.

4. REFERENCES

[1] F. Devaux. The True Processing In Memory Accelerator. In *IEEE Hot Chips*, 2019.

[2] Y. I. Emir and et al. PULSAR: Simultaneous Many-Row Activation for Reliable and High-Performance Computing in Off-the-Shelf DRAM Chips. 2024.

[3] F. Gao and et al. ComputeDRAM: In-Memory Compute Using Off-the-Shelf DRAMs. In *MICRO*, 2019.

[4] N. Hajinazar and et al. SIMDRAM: A Framework for Bit-serial SIMD Processing Using DRAM. In *ASPLOS*, 2021.

[5] M. He and et al. Newton: A DRAM-maker's Accelerator-in-Memory (AiM) Architecture for Machine Learning. In *MICRO*, 2020.

[6] Y. Kim and et al. A Case for Exploiting Subarray-Level Parallelism (SALP) in DRAM. In *ISCA*, 2012.

[7] S. Lee and et al. Hardware Architecture and Software Stack for PIM Based on Commercial DRAM Technology : Industrial Product. In *ISCA*, 2021.

[8] D. Mercadier and et al. Usuba: High-Throughput and Constant-Time Ciphers, by Construction. In *PLDI*, 2019.

[9] X. Peng and et al. CHOPPER: A Compiler Infrastructure for Programmable Bit-serial SIMD Processing Using Memory in DRAM. In *HPCA*, 2023.

[10] V. Seshadri and et al. Ambit: In-Memory Accelerator for Bulk Bitwise Operations Using Commodity DRAM Technology. In *MICRO*, 2017.

[11] X. Xin and et al. ELP2IM: Efficient and Low Power Bitwise Operation Processing in DRAM. In *HPCA*, 2020.