CHOPPER: A Compiler Infrastructure for Programmable Bit-serial SIMD Processing Using Memory in DRAM

Xiangjun Peng† Yaohua Wang‡ Ming-Chang Yang†

†*The Chinese University of Hong Kong* ‡*National University of Defense Technology*
{*xjpeng, mcyang*}*@cse.cuhk.edu.hk* {*yhwang*}*@nudt.edu.cn*

***Abstract*—Increasing interests in Bit-serial SIMD Processing-Using-DRAM (PUD) architectures amplify the needs for a compiler to automate code generation, credited to their ultra-wide SIMD width and reduction of data movements. The state-of-the-art Bit-serial SIMD PUD architectures (1) only provide assembly SIMD programming interfaces, which heavily saddles with programmers to exploit the ultra-wide SIMD width on these architectures; and (2) encapsulate 1-bit operations into multi-bit abstractions, which incurs a granularity mismatch and restricts the optimization space to minimize data movements.**

We present CHOPPER, a new compiler infrastructure to make Bit-serial SIMD PUD more programmable and efficient. For the better programmability, the design of CHOPPER (1) exploits *bit-slicing compilers* to enable automatic memory allocation and code generation, from naturally-expressive codes (i.e. similar to Parallel Haskell) into the "SIMD-Within-A-Register"-style codes; and (2) introduces a new abstraction called "Virtual Code Emitter", to make Bit-serial SIMD PUD architecture exploit Memory-Level Parallelism (i.e. Bank or Subarray) more effectively. For the better efficiency, we propose three novel optimizations for CHOPPER to better exploit the potentials of Bit-serial SIMD PUD architectures, which (1) minimize the amount of intra-subarray data movements; and (2) mitigate the overheads of spilling data outside Bit-serial SIMD PUD architectures. These optimizations can greatly improve the overall efficiency of Bit-serial SIMD PUD architectures. We also discuss (1) the limitations of the current CHOPPER; and (2) the potentials of CHOPPER for other types of Processing-In-Memory architectures.

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 for Bit-serial SIMD PUD architectures. We highlight that, averaged across 16 real-world workloads from 4 PUD-friendly application domains, CHOPPER achieves (A) 1.20X, 1.29X and 1.26X speedup when data can fit within DRAM subarrays; and (B) 12.61X, 9.05X and 9.81X speedup when data need to spill to the secondary storage, on Ambit [50], ELP2IM [56] and SIMDRAM [22], compared with hands-tuned codes using the state-of-the-art methodology [22] for Bit-serial SIMD PUD architectures. These performance benefits also accompany with a great reduction of Lines-of-Codes (LoC) in CHOPPER (i.e. by 4.3X less LoCs for hands-tuning a single subarray, and $>10^3$X less for hands-tuning all subarrays in a rank). We also perform breakdown and sensitivity studies of CHOPPER, to better understand its source benefits and examine its robustness under various architectural features.

I. Introduction

Near-DRAM Processing, as recently-bloomed practices of Processing-In-Memory (PIM), receives a considerable amount of attention due to its potentials to mitigate data movement bottlenecks [12], [33]. However, recent practices fail to exploit the maximum level of internal DRAM bandwidth. This substantially motivates recent efforts on in-DRAM analog computation for bitwise operations [20], [22], [50], [56], which are denoted as Bit-serial SIMD Processing-Using-DRAM (PUD) architectures. These 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) [22]; and (3) synthesize basic logic operations (e.g. AND/OR/NOT) to support complex arithmetic computations.

While Bit-serial SIMD PUD architectures deliver a cost-effective direction to enable massively parallel computation within DRAM chips, both the programmability and efficiency remain as huge obstacles. ❶ The programmability of existing Bit-serial SIMD PUD architectures is restricted by its assembly programming interfaces: all prior works [22], [50], [56] deliver hands-tuned codes, which incur significant overheads for employing these architectures. ❷ The efficiency of existing Bit-serial SIMD PUD architectures is limited by unnecessary data movements: all prior works fail to minimize the amount of intra-subarray data movements, and overlook the overheads from (potential) data spilling. (Detailed in Section III)

We introduce CHOPPER, a new compiler infrastructure to make Bit-serial SIMD PUD architectures more programmable and efficient. Compared with the state-of-the-art hands-tuned methodology for Bit-serial SIMD PUD architectures (i.e. SIMDRAM approach [22]), CHOPPER ❶ provides a synchronous dataflow programming interface for the better expressiveness; ❷ automatically transforms synchronous dataflow codes into heavily-optimized codes for Bit-serial SIMD PUD architectures; and ❸ incorporates new optimizations, to (A) minimize data movements caused by data granularity mismatch, and (B) mitigate potential overheads from data spilling from Bit-serial SIMD PUD architectures.

CHOPPER improves the programmability from the following three aspects. ❶ CHOPPER delivers a synchronous dataflow programming interface, which automates explicit memory allocation via the whole-program analysis; ❷ CHOPPER inherits *bit-slicing compilers* [37], [38], to generate "SIMD-Within-A-Register"-style codes, as demanded in Bit-serial SIMD PUD architectures (denoted as Bit-Sliced Coded throughout the rest of this paper); and ❸ CHOPPER introduces a new abstraction called "VIRtual COde Emitter", to enable CHOPPER-generated codes to exploit Memory-Level Parallelism (Bank or Subarray) more effectively. (Section IV)

CHOPPER improves the overall efficiency via Optimizations for Bit-Sliced codes (OBS), to minimize extra data movements. OBS consists of ❶ a *bit-sliced* code scheduling mechanism by synthesizing multiple operations comprehensively, to improve the utilization of the limited subarray space; ❷ a *bit-sliced* instruction selection scheme, which exploits bit patterns (e.g. bit-level sparsity) on Bit-serial SIMD PUD architectures to improve data reuses of intra-subarray constant values; and ❸ a *bit-sliced* instruction renaming approach to eliminate extra intra-subarray data movements, to reduce both the space consumption and instruction sequence (for the better performance). Our newly-proposed OBS broadens the scope of the state-of-the-art Bit-serial SIMD PUD architectures, and can be fully automated in CHOPPER. (Section V)

We quantitatively examine the benefits of CHOPPER, by hosting it on three Bit-serial SIMD PUD architectures (including Ambit [50], ELP2IM [56] and SIMDRAM [22]), and compare CHOPPER-generated codes against hands-tuned codes on these architectures under the state-of-the-art hands-tuned methodology (i.e. the SIMDRAM approach [22]). We also provide comparisons with state-of-the-art implementations on an Intel Skylake multi-core CPU and a NVIDIA TITAN V GPU. Our evaluations use 16 workloads from 4 PUD-friendly application domains, including: DenseNet from Deep Neural Networks, Wavelet Tree Construction from Compressed Suffix Arrays, DiffGen from Differential Privacy, and a widely-used approach from Significance Weighting (Section VII).

Results Overview. Compared with the state-of-the-art hands-tuned optimizations on Ambit [50], ELP2IM [56] and SIMDRAM [22], CHOPPER-generated codes improve the average performance of all selected workloads by (A) 1.20X, 1.29X and 1.26X speedup when data can fit within DRAM subarrays; and (B) 12.61X, 9.05X and 9.81X speedup when data needs to spill to the secondary storage. The benefits of CHOPPER are credited to the reduction of intra-subarray data movements and (potential) data spilling. These performance benefits also accompany with a great reduction of Lines-of-Codes (LOC) in CHOPPER: averaged across all selected workloads, CHOPPER reduces LOCs by 4.3X for hands-tuning a single subarray, and $>10^3$X less for hands-tuning all subarrays in a rank. Our results also suggest that: CHOPPER can preserve the acceleration benefits of Bit-serial SIMD PUD architectures against the CPU and GPU, when the problem settings grow more complex.

We further provide sensitivity studies by (1) breaking down individual optimizations and examining their impacts separately to better understand the benefits of optimizations in CHOPPER; (2) reconfiguring the DRAM organization (i.e. the size of a subarray) to demonstrate the robustness; and (3) exploiting Subarray-Level Parallelism [30] to examine how the benefits of CHOPPER can be further amplified, with novel architectural features in DRAM chips. Our results suggest that our optimizations in CHOPPER are synergistic, and the benefits of CHOPPER are robust with novel architectural features (Section VIII).

We make the following major contributions in this paper:

- We outline three outstanding issues in existing Bit-serial SIMD PUD architectures, including (1) assembly programming interfaces for restricted programmability; (2) extra intra-subarray data movements for limited efficiency; and (3) (potential) data spilling for huge overheads.
- We propose CHOPPER design to improve the programmability of Bit-serial SIMD PUD architectures. To the best of our knowledge, CHOPPER is the first compiler infrastructure to automatically generate efficient codes for Bit-serial SIMD PUD architectures.
- We propose three synergistic optimizations in CHOPPER, to further improve the efficiency of CHOPPER-generated codes. These optimizations address the issues on (1) extra intra-subarray data movements; and (2) (potential) data spilling. These optimizations broaden the scope of code optimizations for Bit-serial SIMD PUD architectures.
- Our quantitative evaluation results show that CHOPPER can (1) bring significant benefits over the state-of-the-art hands-tuned implementations; (2) provide robust benefits across a variety of architectural configurations; and (3) amplify the benefits with novel architectural features of DRAM chips.

II. Background

A. DRAM Organization and Operations

A modern main memory subsystem consists of one or more memory *channels*, where each channel contains a *memory controller* that manages a dedicated subset of DRAM modules. The modules in a single channel share an off-chip bus that is used to issue commands and transfer data between DRAM modules and memory controller, which typically resides in the processor. Each module is made up of multiple DRAM chips, which are grouped into one or more *ranks*. A chip is divided into multiple *banks*, which can serve memory requests (i.e., loads or stores) independently. Each bank typically consists of 64–128 two-dimensional arrays of DRAM cells called *subarrays* [11], [30], [49], [50], which can serve memory requests (i.e., loads or stores) in parallel and independently of each other. For each subarray, there are 512–2048 rows in commodity DRAM design. The memory controller issues four commands to access and update data within DRAM [28]. In a typical memory access process, (1) The memory controller *activates* the DRAM row containing the data via the ACTIVATE command, which latches the selected DRAM row into the Local Row Buffer (LRB) of the subarray that contains the row;

(2) once the activation finishes, the memory controller issues a READ or WRITE command, which operates on a column of data, on a READ, one column of the LRB is selected using the column decoder and is sent to the Global Row Buffer (GRB) via global bitlines; (3) GRB then drives the data to the chip I/O logic, which sends the data out of the DRAM chip to the memory controller. While a row is activated, the memory controller can issue subsequent READ/WRITE commands to access other columns of data from the LRB if there are other memory requests to the same row, which is called a *row buffer hit*; (4) the controller *precharges* the LRB and the subarray by issuing a PRECHARGE command to prepare all bitlines for a subsequent ACTIVATE command to a different row.

B. Processing-Using-DRAM Architectures

A Processing-Using-DRAM (PUD) subarray is organized in a similar manner with Ambit [50]. PUD DRAM engines uses Triple-Row Activation (TRA) to analogly perform bitwise AND/OR operations, and leverage dual-contact cells to implement NOT operations. Hereby, we provide an architectural overview of a PUD DRAM subarray. Figure 1 shows the organization of a PUD DRAM subarray. Within such a subarray, there are three groups: the **D**ata group (D-group), the **C**ontrol group (C-group) and the **B**itwise group (B-group).

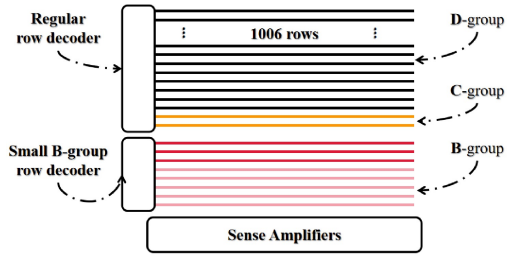


Fig. 1: A Processing-Using-DRAM Subarray.

Ambit [50] implements *bulk bitwise operations* with Triple-Row activation, where three rows(i.e., row *A*, *B*, and *C*) are activated simultaneously, resulting in a bitwise majority function across the cells in these three rows (i.e., $R = AB + BC + AC$). By setting the initial value of row *C* (i.e., control row), to all "0"s (or all "1"s), such majority function turns into a bitwise AND (or OR) of the bits in row A and B (i.e., operand rows). *Ambit* [50] also implements a row of modified dual-connected cells to support bitwise NOT of an entire DRAM row, this makes the PUD logic complete.

The D-group contains regular rows that store data. The C-group consists of two constant rows, called $C0$ and $C1$, that contain all "0"s and all "1"s values, respectively. These rows are used to control which operations (i.e. AND/OR) to perform via a TRA. The D-group and the C-group are connected to the regular row decoder, which selects a single row at a time. The B-group contains six regular rows, called $T0$, $T1$, $T2$, and $T3$; and two rows of dual-contact cells, called ($DCC0$, $\overline{DCC0}$), and ($DCC1$, $\overline{DCC1}$). The B-group rows, called compute rows, are designated to perform bitwise operations. They are all connected to a special row decoder that can simultaneously activate three rows using a single address (i.e., perform a TRA). For a typical subarray size of 1024 rows [2], this design splits the rows into 1006 D-group rows, 2 C-group rows, and 16 B-group rows.

C. Bit-Serial SIMD Processing-Using-DRAM

Several works extend Ambit [50] for architectural improvements. ELP2IM modifies the precharge units (in local row buffer) within bitlines to improve the energy efficiency of bulk bitwise operations [56]. *ComputeDRAM* [20] is a proof-of-concept of Ambit [50], which shows that, it's feasible to implement bulk bitwise row copy, logical AND and OR in unmodified commercial DRAM chips, by violating the nominal timing specification properly to activate multiple rows in rapid succession. Though the above bulk bitwise copy and logical operations show great potential, the inner structure of DRAM prohibits propagating carry bits across different bitlines. So that traditional bit-parallel computation patterns can not perform complex operations (e.g. addition), which require carry propagation across bitlines. This makes the bit-serial computation promising for PUD computation, where the input operands of a computation operation are bitline-aligned and have the same length and width, making each bitline becomes a computation unit, we denote such data layout as bit-serial (vertical) layout. Figure 2 gives an example to compare horizontal data layout and vertical data layout.

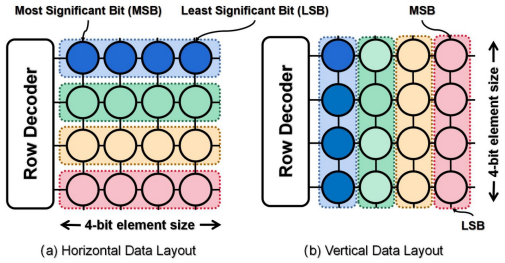


Fig. 2: Data layout: horizontal vs. vertical.

By employing the bit-serial (vertical) layout, PUD engines operate in a Single-Instruction-Multiple-Data (SIMD) manner. This is because the bulk bitwise operations are applied to the entire DRAM row, so all of the results from bitlines are generated in parallel. Such a method makes PUD engines support more complex operations in a cost-effective manner. In modern DRAM chips, enabling inter-bitline operations can destroy the storage capability (i.e. as identified in DRISA [35]). Following the pioneering efforts of *ComputeDRAM* on bit-serial addition, recent efforts show that the Majority functions can be used for complex operations (e.g. SIMDRAM [22] and other works [6]), which can be more efficient than emulating operations using AND/OR/NOT logic. SIMDRAM [22] uses an identical subarray architecture as Ambit [50], exposes multi-bit (i.e. multi-rows in bit-serial layout) operations to the host processor. SIMDRAM [22] also presents the state-of-the-art hands-tuned methodology for a single subarray of Bit-serial SIMD PUD architectures, by reusing the Linear Scan Register Allocation algorithm [45] for the reduction of intra-subarray data movements (for multi-bit operations).

III. Motivation

We outline 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 (Section III-A); (2) extra intra-subarray data movements for the low efficiency, when data can fit within DRAM subarrays (Section III-B); and (3) (potential) data spilling for huge overheads (or even slowdowns) (Section III-C).

A. Challenge 1: Assembly Programming Interfaces for Poor Programmability

Existing Bit-serial SIMD PUD architectures (e.g. [22], [50], [56]) directly expose their instructions to the host processors, and provide assembly-like interfaces for programmers to use. Hence, existing Bit-serial SIMD PUD architectures only deliver assembly programming interfaces, which are considered as great obstacles for programmers.

The State-of-the-art. The programming interface (i.e. SIMDRAM [22]) is in assembly; Moreover, SIMDRAM [22] synthesizes data preparation and computation on Bit-serial SIMD PUD architectures in multi-bits.

Limitation. There are three major limitations, and we detail each of them below.

❶ The assembly programming interface requires programmers to handle memory allocation explicitly, which is a huge burden. The issue can become more serious when the size of the data within a subarray increments.

❷ All Bit-serial SIMD PUD architectures require codes to be written in the "SIMD-Within-A-Register" style, which is notoriously infamous for the programming ease [15]–[17]. Moreover, the synthesis of multi-bit operations further increases the burdens of *bit-sliced* code transformation.

❸ All existing programming interfaces only target one subarray on Bit-serial SIMD PUD architectures, and there are no considerations on how to exploit parallelism of different DRAM components (e.g. Bank-Level or Subarray-Level Parallelism). Note that naively broadcasting the written programs to all subarrays in all banks cannot effectively exploit the Bank-Level (or Subarray-Level) Parallelism.

B. Challenge 2: Extra Intra-subarray Data Movements for Low Efficiency

Early efforts on Bit-serial SIMD PUD architectures overlook the potentials to reduce intra-subarray data movements [20], [50], [56], which uses a pre-assigned region (i.e. **B**itwise group (B-group)) for the computation only. This is because DRAM operations are destructive. Therefore, Bit-serial SIMD PUD demands intra-subarray data movements, to move data between the storage region (i.e. D-Group) and the computation region (B-Group).

The State-of-the-art. The programming abstractions in Bit-serial SIMD PUD architectures [22] incur a granularity mismatch between full-size storing (e.g. word-size, or byte-size) and bitwise (i.e., 1-bit) processing on Bit-serial SIMD PUD architectures, which prevents the minimization of intra-subarray data movements.

Limitation. The above granularity mismatch limits the optimization space for the reduction of intra-subarray data movements: all input operands are (A) transposed and stored at full size, but (B) compute bit-by-bit. This results in redundant buffering for (parts of) data operands.

C. Challenge 3: (Potential) Data Spilling for Huge Overheads

All prior works [20], [22], [50], [56] overlook the potential issues of spilling data outside DRAM (i.e. to the secondary storage), which can incur significant performance overheads (or even slowdowns) of these architectures.

The State-of-the-art. Prior works (e.g. [22], [50], [56]) overlook such an issue, and assume all workloads only require a constant amount of DRAM rows to buffer either input data or intermediate data. Prior works also assume all data can always fit within a DRAM subarray.

Limitation. In fact, the size of intermediate data, can be substantially amplified by the vectorization for the ultra-wide SIMD width, which gradually consume more space in this manner due to the forbidden communications between SIMD lanes. This can eventually hit the limits of DRAM subarrays, which finally leads to data spilling to the secondary storage. The spilling can cause significant overheads (or even slowdowns) of Bit-serial SIMD PUD architectures.

```
// 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 [22].

```
// 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.

Fig. 3: A comparative example of written codes in (A) SIMDRAM Programming Interface [22]; and (B) CHOPPER Programming Interface to perform packed addition and subtraction in a subarray of Bit-serial SIMD PUD architectures.

IV. CHOPPER Design for the Better Programmability

CHOPPER is a new compiler infrastructure to explicitly address the above three challenges in Bit-serial SIMD PUD architectures. To improve the programmability of Bit-serial SIMD PUD architectures, the expected programming interface and compiler should (1) automate memory allocation; (2) automate *bit-sliced* (i.e., "SIMD-Within-A-Register") code generation; and (3) exploit the parallelism of different DRAM components effectively (e.g. Bank-Level Parallelism). In this section, we give an overview of the CHOPPER programming interface and the CHOPPER compiler. We follow the assumption of the platform model for using Bit-serial PUD architectures: viewing all these architectures as the standalone accelerators (e.g. [22], [50], [56]), and offloading coarse-grained functions/computation kernels on Bit-serial SIMD PUD architectures[1].

A. The CHOPPER Programming Interface & Compiler

❶ **Programming Interface.** The CHOPPER programming interface is based on a synchronous dataflow programming language, the Usuba Programming Language [37][2]. The extensions ensures the delivery of all elementary basic types (boolean, integer and real) and basic operators (arithmetic, boolean, relational and conditional). We choose to deliver a dataflow programming interface, to enable the whole-program analysis for (1) automatic memory allocation; and (2) *bit-slicing compilation*. Figure 3 delivers a comparative example of written codes in (3A) SIMDRAM Programming Interface [22]; and (3B) CHOPPER Programming Interface, to perform packed addition and subtraction. The example accounts for a single subarray for the clarity, and CHOPPER delivers a much more elegant implementation than the SIMDRAM programming interface. This is because there are no needs to explicitly handle (1) memory allocation (especially incremental memory allocation); and (2) data transposition and preparation for Bit-serial SIMD PUD architectures.

❷ **Compiler.** The CHOPPER compiler consists of the front-end and the back-end (as shown in Figure 4), and we elaborate each component in detail separately.

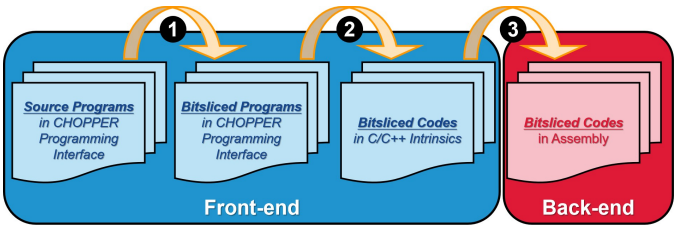


Fig. 4: The workflow of the CHOPPER compiler.

• Front-End. The front-end of the CHOPPER compiler distills the programs written in its programming interface, and translates them into C/C++ codes for a single subarray of Bit-serial SIMD PUD architectures. The front-end is based on the Usuba compiler [37], [38], which is the state-of-the-art *bit-slicing* compiler. A *bit-slicing compiler* can (1) automate the transformation from arithmetic operations to equivalent logic operations; and (2) enable the corresponding transformation by combining data transposition and (logic-synthesized) arithmetic operations. The main difference between Usuba [37], [38] and CHOPPER front-end is that: Usuba compiles transposition and computation codes for processors, but CHOPPER compiles transposition codes for processors and computation codes for Bit-serial SIMD PUD architectures. Hence, we implement this separation at Step-❷ of Figure 4, which consists of two parts: (1) we separate the target of this step to translate transposition codes for the processor; and computation codes for Bit-serial SIMD PUD architectures; and (2) we add an instruction inserter for write operations (after transposition codes), so the processor can write transposed data into Bit-serial SIMD PUD architectures.

• Back-End. The back-end of the CHOPPER compiler translates *bit-sliced* codes in C/C++ into assembly instructions for Bit-serial SIMD PUD architectures. The back-end is based on the LLVM compiler [32], a modular compiler to be portable to different hardware architectures. An iterative use of the back-end broadcasts C/C++ codes, generated from the front-end, to different subarrays one by one.

In summary, the current design choice in CHOPPER can address the first two challenges of the programmability issue (described in Section III-A-❶ and -❷). However, this design focuses on code compilation for a single subarray, which fails to effectively exploit the parallelism of different DRAM components (e.g. Bank-Level Parallelism).

B. The VIRtual COde Emitter: A New Compilation Abstraction for Bit-serial SIMD PUD Compilers

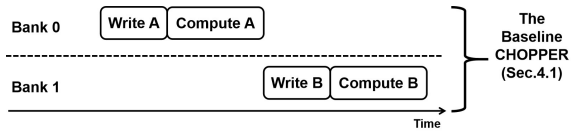
As described in Section IV-A, the baseline CHOPPER design does not aware of the parallelism of different DRAM components (e.g. Bank-Level Parallelism), and therefore the generated codes fail to exploit them. This is because the baseline CHOPPER design naively broadcasts codes for one subarray (in a bank) to multiple subarrays (in all banks). Therefore, the design space of managing multiple subarrays on Bit-serial SIMD PUD architectures remains unexplored.

❶ **Motivation.** Figure 5A shows an example using two banks, where data preparation and computation are executed serially. Therefore, the baseline design (i.e. in both the state-of-the-art methodology and the baseline CHOPPER in Section IV-A) cannot effectively exploit the Bank-Level Parallelism in modern DRAM chips (and Bit-serial SIMD PUD architectures).

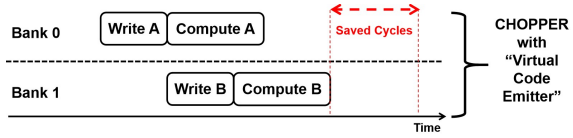
❷ **The "Virtual Code Emitter".** We introduce a new compilation abstraction called "VIRtual COde Emitter", to improve the exploitation of Bank-Level Parallelism on Bit-serial SIMD PUD architectures. The key idea of VIRCOE is to exploit the Bank-Level Parallelism by overlapping data transfer (i.e. the activation before READ/WRITE) and computation (i.e. the Triple-Row Activation) on Bit-serial SIMD PUD architectures,

[1]We do not consider dual-use of these architectures in this work, though intra-subarray data movements can be feasible (e.g. [11], [49], [54]).

[2]The Usuba Programming Language is a synchronous dataflow language, since it follows the usual compilation pipeline of synchronous dataflow languages, namely normalization and scheduling, with transformations. We have extensively engineered it for the applicability, to support our evaluation setups (see the implemented workloads in Section VII).



(A) The example under the baseline CHOPPER design.



(B) The example under the CHOPPER design with VIRCOE.

Fig. 5: A comparative example on emitting codes to two different banks on Bit-serial SIMD PUD, under (A) the baseline CHOPPER design (described in Section IV-A); and (B) the CHOPPER with "Virtual Code Emitter".

in a more effective manner. Figure 5B gives out a pictorial example to showcase the benefits of VIRCOE.

• Design. The "Virtual Code Emitter" is architected as a middleware between the front-end and the back-end of the CHOPPER compiler. The goal of such middleware is to exploit the Bank-Level Parallelism by making code emission aware of the DRAM hierarchy (namely emitting the codes to multiple subarrays (in all banks) rather than only a single subarray (in a bank)). The design of VIRCOE initializes virtual program counters to direct the first code for every subarray individually: all subarrays are flagged based on different banks, and a virtual program counter is maintained for each subarray. The counter directs the next code, after emitting the current code.

• Workflow. VIRCOE performs the code emission subarray by subarray, and triggers BLP-aware optimizations whenever a data transfer code is emitted in the current subarray. If a data transfer code is emitted, VIRCOE checks all the subarrays in other banks, to examine whether there are any computation code can be emitted (by evaluating the corresponding virtual program counter). The check is terminated when one of the two conditions is satisfied: (1) the total timing of the to-be-emitted computation codes equals/exceeds the data transfer; or (2) all subarrays are iterated and no more codes can be emitted. We implement this middleware as a separate stack, hence there is no need for modifications in either the front-end or the back-end.

• Applicability. Note that this design can also be used when Subarray-Level Parallelism (SALP) is enabled [30], by expanding the search range for the conditional emission within the triggered optimization: under SALP, the activation can be parallelized at subarray level. Hence, the search range of VIRCOE can include all other subarrays within the same bank as well. (i.e. evaluated in Section VIII-D).

C. Opportunities for Optimizing CHOPPER

Though the above CHOPPER design greatly improves the programmability and BLP exploitation of Bit-serial SIMD PUD architectures. The overall efficiency still needs to be improved to substantially reduce data movements and spilling. We outline three new opportunities of *bit-sliced* codes for Bit-serial SIMD PUD architectures, generated by CHOPPER.

❶ **Reducing Redundant Buffering**. As covered in Section III, buffering redundant bits on Bit-serial SIMD PUD architectures may cause extra data movements (e.g. spilling). Therefore, it's desired to store as less intermediate bits as possible on these architectures. Credited to CHOPPER, it's feasible to perform the whole-program analysis, and automatically schedule dependent operations together whenever possible. Therefore, every dependent operation on the same bitslice can be grouped, and thus this bitslice does not need to be buffered (in D-group, as shown in Figure 1).

❷ **Increasing Data Reuse**. Similar with ❶, data operands, which are the same across all SIMD lanes (i.e. DRAM bitlines), can be considered as redundant as well. In existing Bit-serial SIMD PUD architectures [22], [50], there are constant bits within each DRAM subarray. However, they are only used for controlling the Triple-Row Activation. In fact, these constant bits can be exploited to construct constant operands in a bitwise manner, which can greatly improve data reuse. Prior works [22], [50] do not use them for data reuse because: the programming abstractions for data operands are at full size, and it's challenging to exploit bit-level operand reuse. However, CHOPPER generates *bit-sliced* codes, which paves the way for data reuse at bit level. Hence, it's promising to exploit bit-level patterns (e.g. bit-level sparsity) opportunistically so as to make full use of these constant values (namely C-group in Figure 1) for data reuse.

❸ **Reducing "Store-Copy-Compute" Patterns**. All existing Bit-serial SIMD PUD architectures follow the "Store-Copy-Compute" pattern (i.e. store the operands, copy them into computation region, and then compute), since DRAM operations are destructive. CHOPPER reduces the granularity of storing data operands to one bit by providing *bit-sliced* codes. Hence, if the bitslice only require "one-shot" computation (i.e. no need to store in D-Group as shown in Figure 1), it's feasible to opportunistically direct the bitslice on the computation region (namely B-group in Figure 1), eliminating unnecessary "Copy" in the "Store-Copy-Compute" Pattern.

V. Optimizing CHOPPER for Better Efficiency

We examine how to further reduce data movements on Bit-serial SIMD PUD architectures. Though *bit-slicing* can partially resolve the granularity mismatch between storing (i.e. full-size) and processing (i.e. 1-bit) operands on Bit-serial SIMD PUD architectures, it's still demanded to exploit the architectural features of Bit-serial SIMD PUD, to further address the granularity mismatch. To this end, we 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.

A. Optimization 1: Bit-sliced Code Scheduling to Aggregate Dependent Operations

To exploit the opportunity outlined in Section IV-C-❶, CHOPPER needs to exclusively manipulates distilled codes from its programming interface, to reduce redundant data buffering for dependent operations. This is because *bit-sliced* codes still inherit the execution order as these codes on the full-size

granularity of data operands. Therefore, it's still demanded to minimize the number of rows for buffering intermediate data operands. To this end, we propose a new *bit-sliced* code scheduling, to aggregate dependent operations.

❶ **Design**. This optimization reorders and aggregates *bit-sliced* operations based on the data dependency, to minimize the number of rows for buffering intermediate operands. This is achieved by a newly-proposed scheduling of *bit-sliced* codes. The design consists of two parts. First, the compiler demands the knowledge of the dependency of all *bit-sliced* operations. Second, the compiler demands the statistics for the occurrences of the dependant operands in *bit-sliced* codes, to maximize the benefits of this optimization.

❷ **Implementation**. This optimization is completely implemented in the front-end (within Step-❷ in Figure 4), which consists of two modifications. First, we add a program analysis component to extract the dependency of all operations. This can be directly derived from the dataflow graph when *bit-slicing* automation is carried out. Second, we add a *bit-sliced* code scheduling component to aggregate dependent operations under a greedy manner. The algorithm first ranks all variables based on their number of occurrences in different operations; and then performs the aggregation, according to the ranking.

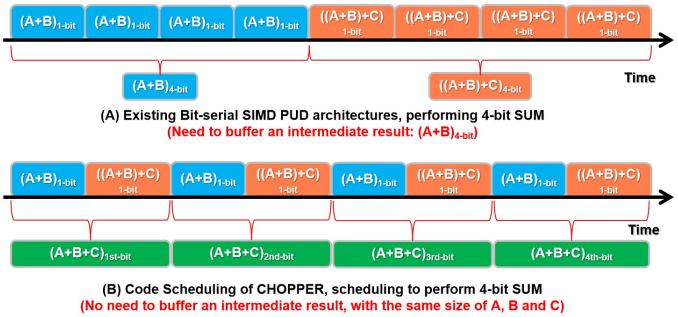


Fig. 6: A comparative example for 4-bit in-DRAM bit-serial summation between (A) existing Bit-serial SIMD PUD architectures, which need to buffer intermediate results (compiled by CHOPPER without Code Scheduling); and (B) CHOPPER with Code Scheduling, which doesnot need to buffer intermediate results.

❸ **Example**. Figure 6 shows a comparative example about how scheduling of *bit-sliced* codes improves the utilization of in-DRAM data streaming. We compare our scheduling proposal with existing Bit-serial SIMD PUD architectures (even compiled by CHOPPER without Optimization 1), using two consecutive 4-bit summations as an example. As shown in Figure 6-(A), the first 4-bit summation generates a 4-bit intermediate result, and uses this result to perform the other 4-bit summation for the final result. However, if the proposed scheduling is used to aggregate dependent operations (as shown in Figure 6-(B)), every summation consumes all operands, without the need to buffer the intermediate bits.

B. Optimization 2: Bit-sliced Instruction Selection to Maximize Data Reuses

As identified in Section IV-C-❷, CHOPPER can actively examine the opportunities to use constant values for data reuses. Existing Bit-serial SIMD PUD architectures maintain two rows to store constant values (i.e. all "0" or "1"), which are used to control bitwise AND/OR operations [22], [49], [50]. Such in-DRAM data initialization can avoid significant data movements by performing intra-subarray row copy. Credited to *bit-slicing* in CHOPPER, the opportunities for bit-level data reuses are significantly amplified.

❶ **Design**. To maximize the extent of data reuse, we observe that we can aggressively perform bit-level data reuse in Bit-serial SIMD PUD architectures. The conventional compilers only expose constant values at the operand level, whereas bit-level reuse is feasible on Bit-serial SIMD PUD architectures (e.g. bit-level sparsity). Also, since bit-level data reuse is non-trivial in conventional compilers, it's also expected to allow programmers to transparently decide whether this optimization shall be enforced based on their own specifications, if they deem necessary.

❷ **Implementation**. This bit-level reuse optimization is an addition to the operand-level data reuse, which is built with two parts ranging from the front-end and back-end. First, we provide an extra parser in front-end of CHOPPER (within Step-❷ from Figure 4), to extract bit-level value broadcast statements within the source program. This is achieved by exposing an annotation for programmers, so that they can trigger this optimization and provide the specifications (e.g. which set of bits and the constant values). Second, we add an additional pass in back-end of CHOPPER (within Step-❸ from Figure 4), to bridge the C/C++ intrinsic functions for constant values and assembly codes, using LLVM Constant Classes. Note that our addition do not affect the operand-level data reuse. We provide an example to pictorially showcase the benefits of this optimization as follow.

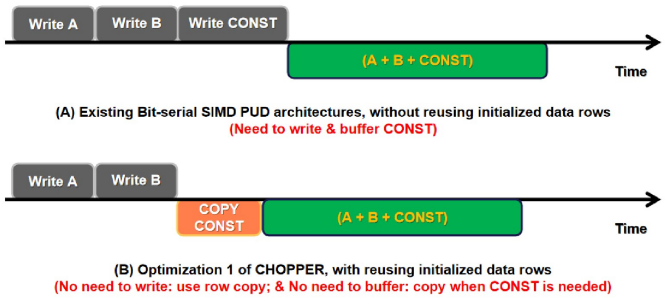


Fig. 7: A comparative example for in-DRAM bit-serial addition $A + B + CONST$ (where $CONST$ refers to CONSTANT values), between (A) existing Bit-serial SIMD PUD architectures which need to write constant values by the CPU; and (B) Data Reuse of our proposal by taking advantage of reusing initialized data rows.

❸ **Example**. Figure 7 shows a comparative example about how this optimization improves both the performance and space efficiency. We consider an addition $A+B+CONST$, where $CONST$ refers to CONSTANT values (i.e. the same across all lanes of Bit-serial SIMD PUD architectures). As shown in Figure 7-(A), the CONST needs to be written by the CPU and buffered within the Bit-serial SIMD PUD architectures.

This causes two inefficiencies: (1) data movements increase since the CPU needs to write the CONST values; and (2) the written CONST values need to be buffered within Bit-serial SIMD PUD architectures, and the space efficiency decreases. However, if we can reuse the initialized data rows (as shown in Figure 7-(B), both the performance and space efficiency can be improved significantly, because (1) data movements are reduced for performance; and (2) CONST values would be generated only when it's needed, without storing them in the D-Group of the subarray.

C. Optimization 3: Renaming Bit-Sliced Instructions to Eliminate Redundant Data Buffering

As for the opportunity in IV-C-❸, CHOPPER can opportunistically shorten the instruction sequence by directly storing the bitslice on the computation region, if the bitslice can be overwritten. A typical operation in Bit-serial SIMD PUD architectures follows the "Store-Copy-Compute" pattern, which refers to "store the data, copy them into B-group (i.e. Rows for Multi-row Activation) and then compute" [22], [50], [56]. This is due to the fact that operands are destructed during the multi-row activation procedure in DRAM, so operands have to be copied to a specific region (i.e., B-group) to maintain their value. Obviously, such copy operations waste extra space for buffering data and also incur extra intra-subarray data movements. Credited to CHOPPER, we observe that there exits many "one-shot" bitslices, which is only computed once, and will never be used again. For these "one-shot" bitslices, the typical "Store-Copy-Compute" pattern can be shortened as "Store-Compute".

❶ **Design**. The design builds upon the benefits from the first two optimizations: for any "one-shot" bitslices following the "Store-Copy-Compute" pattern, the compiler first eliminates "copy" instructions and then renames the addresses of "store" instructions to store directly in the B-group.

❷ **Implementation**. We implement a specialized pass on the back-end of CHOPPER (within Step-❸ in Figure 4). This specialized pass is organized into the following three steps. ❶ iterating all glued regions of codes after Instruction Selection phase, and identifying "one-shot" bitslices with the patterns of "write-copy-compute"; ❷ within the glued regions of codes, deleting the copy operations to move written data into B-group; and ❸ replacing the destination address of "write" operations", with the destination address of the deleted "copy" operations.

❸ **Example**. Figure 8 shows a comparative example of how this optimization improves both the performance and space efficiency. We consider an addition $A + B$. As shown in Figure 8-(A), the command sequence follows "Store-Copy-Compute" pattern, which causes extra overheads including: (1) data movements increase since in-subarray data copy is needed, resulting in performance overheads; and (2) the written values need to be buffered within Bit-serial SIMD PUD architectures, and the space efficiency decreases. However, if we can rename and eliminate copy operations to redirect the data (as shown in Figure 8-(B)), both the performance and space efficiency can be improved significantly, because (1) data movements are reduced for better performance; and (2) to-be-compute values are directly used without buffering.

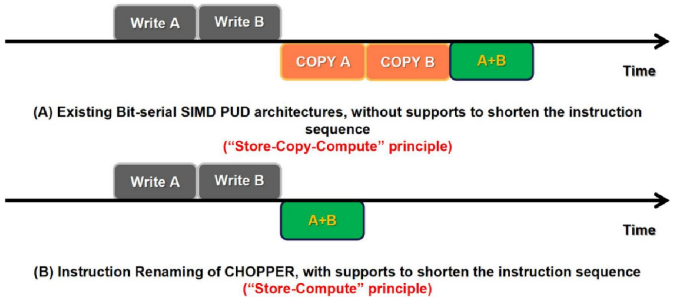


Fig. 8: A comparative example for in-DRAM bit-serial addition $A + B$ between (A) existing Bit-serial SIMD PUD architectures without supports of Instruction Renaming; and (B) Instruction Renaming of our proposal by redirecting written data directly on computation regions.

D. Putting Them Together

All optimizations in CHOPPER are synergistic (as evidenced by our breakdown analysis in Section VIII-B). Overall, CHOPPER broadens the scope of existing hands-tuned methodology for Bit-serial SIMD PUD architectures, by optimizing intra-subarray data movements and mitigating (potential) data spilling issues. Moreover, CHOPPER delivers a fully-functional compiler to automate these optimizations, which greatly improves both the programmability and efficiency.

❶ **Intra-subarray Data Movments:** CHOPPER can further reduce the amount of intra-subarray data movements, compared with SIMDRAM [22]. First, the programming interface of SIMDRAM [22] prohibits efficient analysis of the whole program, to deliver similar variants of optimization 2 from CHOPPER. Second, SIMDRAM [22] stores all data operands as full size, and therefore it's infeasible to fully exploit optimization 1 and 3 from CHOPPER.

❷ **(Potential) Data Spilling:** CHOPPER first identify the the impacts of (potential) data spilling, whereas SIMDRAM [22] doesnot address. First, SIMDRAM [22] doesnot resolve the granularity mismatch between storing and processing operands, which causes redundant data buffering. Second, the methodology, used in SIMDRAM [22], solely focuses on intra-subarray optimizations, which overlooks the potential issues of data spilling.

We also identify the virtues of the automation. Though all optimizations can be hands-tuned, the costs can be considered highly significant due to two reasons. First, all codes on Bit-serial SIMD PUD architectures require to be written in the "SIMD-Within-A-Register" style, which is notoriously infamous for both the programming difficulty and the code size. Second, all optimizations, proposed in CHOPPER, requires the whole-program analysis. Given the fact that the "SIMD-Within-A-Register" codes are significantly larger than the original program (normally 100X more LOCs), the automation of these newly-proposed optimizations are essential.

VI. Discussions

We first discuss the limitations of CHOPPER, then we discuss the extendability of CHOPPER and its potentials to be combined with dataflow graph scheduling.

A. Limitations of CHOPPER

The current CHOPPER prototype follows the assumption of existing Bit-serial SIMD PUD architectures [22], [50], [56], which directly uses physical addresses for memory allocation and computation. This assumption is not practical, given the fact that (1) modern computer systems apply virtual-to-physical address mapping; and (2) modern DRAM chips apply internal address remapping. Though the above issues do not affect the proof-of-concept of our work, we envision that incorporating the above features into CHOPPER and its follow-up works are essential, to enhance its practicality.

B. Extendability of CHOPPER to other PIM architectures

CHOPPER is expected to be generally extendable to other Processing-In-Memory architectures, including both Processing-Using-Memory and Processing-Near-Memory architectures. By inheriting Usuba [37], [38], CHOPPER front-end also yields a systematic generalization of *bit-slicing* compilation, which supports (1) the horizontal (or bit-parallel) and vertical (or bit-serial) layout of data operands; and (2) different sizes of data operands (e.g. bytes in Usuba [37], [38], since modern SIMD architectures use bytes as its elementary granularity for processing) for operand slicing. Therefore, the extendability of CHOPPER is more promising than the current shape, and we discuss the potentials below.

❶ For Processing-Using-Memory (PUM) architectures, CHOPPER can be extended to (1) facilitate with different forms of data layouts (e.g. bit-parallel and bit-serial); and (2) support different granularity of data operands (e.g. bytes and bits). Such an unique capability is credited to the inheritance of the *bit-slicing* generalization in CHOPPER front-end, and we envision CHOPPER is applicable to benefit different Processing-Using-DRAM architectures with marginal costs (e.g. pLUTo [14], Fulcrum [34] and etc.). Additionally, CHOPPER can also benefit other types of PUM architectures (e.g. SRAM [13], PCM [36], ReRAM [53], RTM [40], NAND Flash [19], [42] and etc.).

❷ For Processing-Near-Memory (PNM) architectures, CHOPPER can be applicable by combining their C/C++ compilers as the back-end. This methodology can allow PNM architectures benefit from *bit-slicing* for the better throughput, under different elementary sizes of data operands (e.g. IRAM [44], FlexRAM [18], Active Pages [41], DIVA [23], UPMEM DPU [12], SAMSUNG FIM-DRAM [33], SK Hynix AiM [24] and etc.). Though no prior works examine the benefits of *bit-slicing* on PNM architectures, we assume *bit-slicing* can play a significant role in PNM architectures because the area constraints, when integrating processing logic near memory, demand software tricks to improve the performance. *Bit-slicing* is particularly promising because the simplification of operation complexity has already been evidenced by recent trends of PNM architectures (e.g. [5], [7]).

C. Potential Combinations between CHOPPER with Dataflow Graph Scheduling

CHOPPER proof-of-concepts that PIM computation kernels can be exposed as dataflow graphs, and this can potentially allow more sophisticated strategies in dataflow scheduling for triggering PIM acceleration. Recent advances demonstrate that dataflow graphs can be used to model large-scale computations (e.g. TensorFlow [4]), and such systems can deliver optimizations at the level of the dataflow graph (e.g. [29]). One can view the dataflow graph for PIM architectures (i.e. generated by CHOPPER or its variants) as the sub-graph, and perform graph optimizations/substitutions in such systems.

VII. Experimental Methodology

A. Experimental Infrastructure

We use gem5 simulator [10] to implement Bit-serial SIMD PUD architectures, where we integrated Ramulator for DRAM [31]. We also model a SSD using MQSim [52], to account for potential data spilling. For Bit-serial SIMD PUD architectures, we consider Ambit [50], ELP2IM [56] and SIMDRAM [22], by rigorously comparing our implementations against them to ensure the correctness. All architectures exploit the Bank-Level Parallelism, unless specified otherwise. We compare them real machines, including a Intel Skylake multi-core CPU [1] and an NVIDIA TITAN V GPU [3]. Detailed configurations are shown in Table I.

TABLE I: Evaluated system configurations.

Intel Skylake CPU	x86 [1], 8-cores, out-of-order, 4GHz; *L1 D+I. Private Cache:* 32kB, 8-way, 64 B line; *L2 Private Cache:* 256kB, 4-way, 64 B line; *L3 Shared Cache:* 8MB, 16-way, 64 B line; *Main Memory:* 32GB DDR4-2400, 4 channels & ranks
NVIDIA TITAN V GPU	6 graphics processing clusters, 5120 CUDA Cores; 80 streaming multiprocessors, 1.2 GHz base clock; L2 Cache: 4.5MB; Main Memory: 12GB HBM
Ambit [50] ELP2IM [56] SIMDRAM [22]	gem5 emulation; x86 [1], 1-core, OoO, 4 GHz; *L1 D+I. Cache:* 32kB, 8-way, 64 B line;; *L2 Cache:* 256kB, 4-way, 64 B line; *Mem. Controller:* 8kB row size, FR-FCFS [48], [55] scheduling; *Main Memory:* DDR4-2400, 1 channel, 1 rank, 16 banks; *Solid-State Drive:* 60GB, 1 channel, 1 chip/channel, 1 die/chip;

B. Workload Specifications

We elaborate how the workloads from four representative application domains can be mapped to Bit-serial SIMD PUD architectures, and describe how we vary the detailed configurations (as shown in Table II).

TABLE II: An overview of detailed configurations regarding derived real-world workloads, from four application domains.

Types	**Workload Configurations**			
DenseNet	layers within a Dense Block (DB) =			
	16	32	64	128
Wavelet Tree Construction	σ =			
	64	128	256	512
DiffGen	the number of attributes (attr.) =			
	64	128	256	512
Significance Weighting	element size within user-item matrix =			
	64-bit	128-bit	256-bit	512-bit

❶ **Deep Neural Networks**. We select DenseNet [8], [27] as DNN networks. Different from other DNN architectures, DenseNet leverages the concept of "feature reuse" to enhance performance. DenseNet divides multiple layers into Dense Blocks. Within each Dense Block, each layer takes all input feature maps from previous layers. Therefore, the design of overwriting the input results layer by layer, as existing Bit-serial SIMD PUD architectures, cannot be applied for DenseNet. We apply 5 Dense Blocks for our experiments, and we vary the size of each Dense Block.

❷ **Compressed Suffix Arrays**. We select Wavelet Tree Construction [21], a fast query-able data structure where no decompression is required. Wavelet Tree encodes the document based on the partitions of the alphabet/subsets, and recursively repeat until the subset can be distinguished using 0 and 1 (i.e. when the subset only contains one character). Existing designs perform bit-serial subtraction between characters and medians of partitions/subsets, and use the sign bits as the encoding. Therefore, the design requires to buffer all encoding from previous layers, so that Bit-serial SIMD PUD architectures can reorganize such medians layer-by-layer, to ensure the correctness when encoding. We use a fixed size for the input document as 2GB, apply the size of characters as 8-bit, and we vary the size of the alphabet.

❸ **Differential Privacy**. We select DiffGen algorithm [39], a pioneering algorithm on data release through differential privacy. DiffGen partitions the taxonomy tree of attributes, to recursively encode raw data based on different levels of partitions. We use a fixed 4GB size for the input document, set each attribute as 16-bit, and vary the number of attributes.

❹ **Significance Weighting**. We select Significance Weighting for collaborative filtering [25], [26], an industrial standard for normalizing user data in recommender systems. Significance Weighting normalizes the user statistics based on an empirical information: if the user rates less then 50 items, the statistics need to be normalized. We use addition for such a normalization. The input user-item matrix is fixed as 4GB, and we vary the size of data elements within user-item matrix. Note that each element has a 864-bit identifier.

C. Benchmark Configurations

We use representative frameworks to evaluate respective workloads on multi-core CPUs and GPU separately. For DenseNet inference, we use PyTorch [43] to perform the inference; for Wavelet Tree Construction, we use LevelWT [51] for CPUs and implement a variant for GPUs; for Differential Privacy, we leverage LevelWT [51] to extend DiffGen for CPUs and GPUs; as for Significance Weighting, we implement the algorithm [25], [26] for CPUs and GPUs. We rigorously tune these implementations for multiple attempts, in order to obtain the best performance.

VIII. Experimental Results

We first compare CHOPPER against the state-of-the-art hands-tuned implementations (i.e. the SIMDRAM methodology [22]) in Section VIII-A. Then we perform a breakdown analysis of CHOPPER to understand the benefits of each optimization in Section VIII-B. Next, we examine potential impacts of different subarray sizes in DRAM in Section VIII-C. Finally, we examine potential impacts when enabling subarray-level parallelism in DRAM in Section VIII-D.

A. CHOPPER versus Hands-Tuned Codes

In Figure 9 shows the speedup of Titan V, three Bit-serial SIMD PUD architectures (i.e. Ambit [50], ELP2IM [56] and SIMDRAM [22]) with hands-tuned methodology and CHOPPER respectively, over the Intel Skylake multi-core CPU. Note that the first two configurations of each workload do not require data spilling while the rest configurations do. We make two observations from Figure 9.

First, CHOPPER improves the performance of Bit-serial SIMD PUD architectures over the state-of-the-art hands-tuned methodology, *when data completely fit within DRAM subarrays (i.e., the first two configurations)*. Compared with hands-tuned codes on Ambit [50], ELP2IM [56] and SIMDRAM, CHOPPER codes deliver an average speedup by 1.25X, 1.16X and 1.11X for DenseNet inference; 1.16X, 1.08X and 1.03X for Wavelet Tree Construction; 1.23X, 1.59X, and 1.64X for DiffGen; and 1.26X, 1.29X, and 1.27X for Significance Weighting. This is because (1) the current hands-tuned methodology cannot minimize intra-subarray data movements due to the data granularity mismatch, and CHOPPER effectively overcomes this problem and thus minimizing redundant intra-subarray data movements; and (2) the emitted codes from CHOPPER effectively exploit the Bank-Level parallelism.

Second, CHOPPER *significantly* improves the performance of Bit-serial SIMD PUD architectures over the state-of-the-art hands-tuned methodology, *when data need to be spilled to the secondary storage*. Compared with hands-tuned codes on Ambit [50], ELP2IM [56] and SIMDRAM, CHOPPER codes

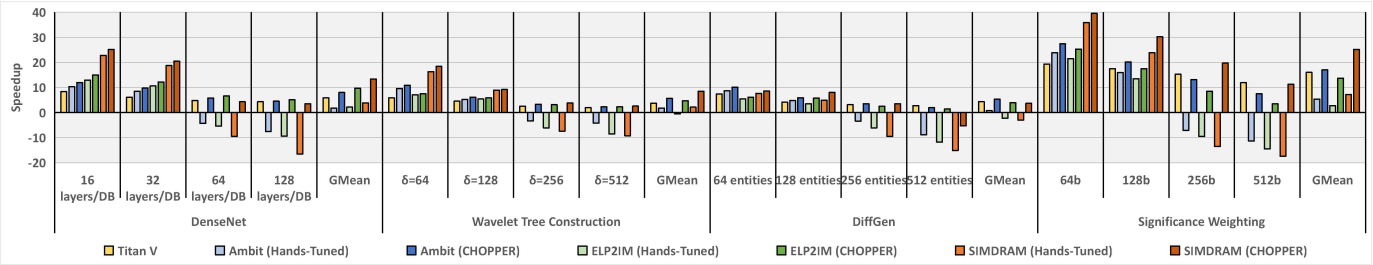


Fig. 9: Speedup of all workloads over Intel multi-core for CHOPPER and the hands-tuned codes.

deliver an average speedup by 7.96X, 7.71X, and 5.82X for DenseNet inference; 5.05X, 3.82X, and 6.89X for Wavelet Tree Construction; 32.29X, 15.99X, and 20.79X for DiffGen; and 5.12X, 8.68X, and 5.73X for Significance Weighting. This is because current hands-tuned methodology overlook the potential issues of data spilling, and CHOPPER can greatly mitigate the overheads from data spilling on Bit-serial SIMD PUD architectures, by effectively exploiting the limited space of DRAM subarrays. This enables these architectures to accelerate a broader range of problem settings.

TABLE III: A comparison of Lines-Of-Codes in the state-of-the-art hands-tuned methodology and CHOPPER for selected workloads. "single" refers to hands-tuned codes for one subarray, and "all" refers to hands-tuned codes for all subarrays. Note that WTC stands for "Wavelet Tree Construction", and SW stands for Significance Weighting".

Methods	Workloads			
	DenseNet	WTC	DiffGen	SW
Hands-Tuned (single)	$\approx$ 2.1K	$\approx$ 1.5K	$\approx$ 1.7K	$\approx$ 0.9K
Hands-Tuned (all)	$\approx$ 4.3K^2	$\approx$ 3.1K^2	$\approx$ 3.5K^2	$\approx$ 1.8K^2
CHOPPER	$\approx$ 0.7K	$\approx$ 0.3K	$\approx$ 0.4K	$\approx$ 0.2K

We then compare Lines-Of-Codes (LOCs) of CHOPPER-generated codes with LOCs of the hands-tuned codes for Bit-serial SIMD PUD architectures, following the methodology in SIMDRAM [22]. We rigorously examine our implementations against SIMDRAM [22], to ensure the correctness of our implementations.

Table III reports the LOCs of hands-tuned codes and CHOPPER codes, for the selected workloads in our experiments. We make the observation that, CHOPPER can significantly reduce the LOCs of Bit-serial SIMD PUD workloads. Compared with hands-tuned codes for one subarray, CHOPPER can reduce LOCs by 3.2X, 5.1X, 4.3X and 4.4X, for DenseNet, Wavelet Tree Construction, DiffGen and Significance Weighting. This is because CHOPPER delivers a dataflow programming interface, and eases the development costs by automating (1) memory allocation and managements of DRAM subarrays; and (2) *bit-sliced* code generation. These automation allows CHOPPER programming interface to solely focus on high-level implementation for these algorithms, which substaintially reduce LOCs significantly. Compared with hands-tuned codes for all subarrays, the LOC of CHOPPER-generated codes can be 10^3X less. This is because CHOPPER automates the code emission to effectively manage all subarrays. On the contrary, SIMDRAM requires explicitly to implement the algorithms subarray-by-subarray, which greatly increases the programming burdens.

B. Breakdown Analysis of CHOPPER

As suggested in Section VIII-A, Ambit [50] can deliver better coverage in our evaluation workloads than both SIMDRAM [22] and ELP2IM [56], where no slowdowns are incurred under CHOPPER over the multi-core CPU. Hence, we perform a breakdown analysis based on Ambit [50]. We break down CHOPPER to examine the performance gains of each optimization. Table IV lists CHOPPER variants with different optimizations. Figure 10 shows the performance gains of CHOPPER variants over Intel multi-core CPU.

TABLE IV: Denotations for CHOPPER breakdown.

abbr.	Optimizations		
	schedule	reuse	rename
bitslice			
schedule	✔		
reuse	✔	✔	
rename	✔	✔	✔

We observe that, CHOPPER variants, with optimizations enabled, *always* improve performance over CHOPPER-bitslice. CHOPPER variants provides an average speedup over CHOPPER-bitslice by 2.54X (up to 5.31X) for DenseNet inference, 3.85X (up to 11.35X) for Wavelet Tree Construction, 6.69X (up to 22.63X) for DiffGen, and 1.68X (up to 3.19X) for Significance Weighting. Across all 16 real-world workloads, the average performance improvement of CHOPPER-full is 10.33X, 1.38X and 5.11X, compared with Intel Skylake multi-core CPU, TITAN V GPU and CHOPPER-bitslice. We observe CHOPPER-bitslice incurs slowdowns (compread with TITAN V GPU), when the problem settings become more complex. This is because CHOPPER-bitslice cannot efficiently take advantage of the DRAM capacity, leading to the problem of data spilling.

C. Impacts of DRAM Subarray Size

We vary the size of subarrays within each bank (i.e. with a fixed total capacity), from 512 rows/subarray, 1024 rows/subarray (default) to 2048 rows/subarray. Figure 11 reports the results of this experiment. We observe that the performance improvements of CHOPPER are *robust* with different sizes of subarrays within each bank. CHOPPER *always* improves the overall performance when either shrinking or

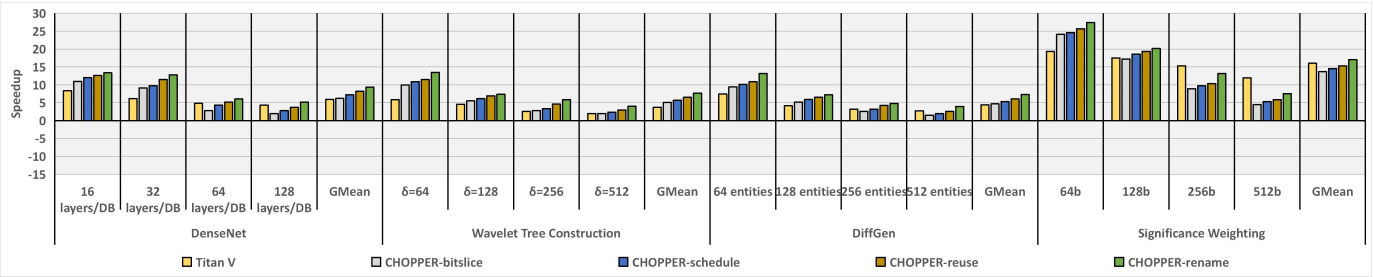


Fig. 10: Speedup of all workloads over Intel Skylake multi-core for CHOPPER breakdown analysis.

enlarging the subarray size. This is because CHOPPER can both exploit the memory capacity efficiently and accelerate the computation of Bit-serial SIMD PUD architectures.

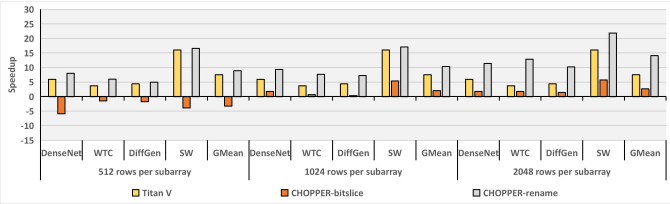


Fig. 11: Speedup of Bit-serial SIMD PUD architectures with various subarray sizes over Intel multi-core.

D. Impacts of Subarray-Level Parallelism

We enable Subarray-Level Parallelism (SALP) [30] to understand how CHOPPER performs under such a setting. To demystify the trade-offs of our newly-proposed VIRCOE code emitter, we reconfigure VIRCOE to make it aware of Bank-Level Parallelism (BLP) and SALP respectively, and examine the impacts with or without SALP enabled. Figure 12 reports the results of this experiment.

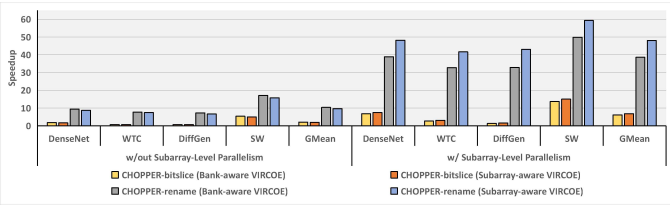


Fig. 12: Speedup of Bit-serial SIMD PUD architectures w/ or w/out Subarray-Level Parallelism over Intel multi-core.

We make two observations. First, the performance improvements of CHOPPER (w/ Subarray-aware VIRCOE) *degrade* when only BLP is enabled. When SALP is not enabled, compared with CHOPPER-bitslice and CHOPPER-rename (w/ Bank-aware VIRCOE), CHOPPER-bitslice and CHOPPER-rename (w/ Subarray-aware VIRCOE) degrades the performance benefits by 7.9% and 7.5% averaged across all workloads. This is because codes are emitted from the improper configuration of VIRCOE, which exaggerates the bank conflicts. Second, the performance improvements of CHOPPER (w/ Subarray-aware VIRCOE) *amplify* when SALP is enabled. When SALP is enabled, compared with CHOPPER-bitslice and CHOPPER-rename (w/ Bank-aware VIRCOE), CHOPPER-bitslice and CHOPPER-rename (w/ Subarray-aware VIRCOE) amplifies the performance benefits by 11.1% and 24.5% averaged across all workloads. This is because the Subarray-aware VIRCOE makes CHOPPER-generated codes effectively exploit SALP, whereas Bank-aware VIRCOE can not.

IX. Related Works

We discuss related works of CHOPPER in a broad context of Processing-In-Memory (PIM) architectures, including both Processing-Using-Memory and Processing-Near-Memory. To the best of our knowledge, CHOPPER is the first work to (1) introduce dataflow programming interface in PIM; (2) introduce *bit-slicing* and its automation into PIM; and (3) systematically address the data movement overheads, caused by the granularity mismatch between storing and processing data operands on PIM.

CHOPPER delivers the first proof-of-concept that dataflow programming can be used to program computation kernels on PIM architectures. Early works (e.g. IRAM [44], FlexRAM [18], Active Pages [41], DIVA [23] and etc.) and recent works (e.g. UPMEM DPU [12], SAMSUNG FIM-DRAM [33], SK Hynix AiM [24] and etc.) only use programming interfaces in C/C++ or assembly. The modular design of CHOPPER allows these architectures to benefit from dataflow programming. If their C/C++ compilers are available, they can be integrated with CHOPPER as the back-end to make these architectures benefit from dataflow programming (as we discussed in Section VI).

CHOPPER is the first work to introduce *bit-slicing* (and its automation) into PIM architectures. Pioneered by Biham et al [9], *bit-slicing* is considered as hand-tuned optimizations to effectively maximize the parallelism, which is notoriously infamous in terms of the programmer burdens [15]–[17]. There are only a limited amount of efforts on *bit-slicing* automation [46], [47], and Usuba is the most recent advance for the automation of *bit-slicing* [37], [38]. CHOPPER is fundamentally different from Usuba regarding both design goals and implementation choices: (A) CHOPPER targets Bit-serial SIMD PUD architectures, and proposes new optimizations on *bit-sliced* codes on these architectures; (B) Usuba solely focuses on automating *bit-slicing* for modern SIMD architectures. Moreover, CHOPPER can potentially allow other PIM to further improve the parallelism, via *bit-slicing* tricks in an automated manner.

CHOPPER is the first work to systematically address the granularity mismatch between storing and processing operands on PIM architectures. The closet work to the OBS in CHOPPER is the reuse of "Linear Scan Register Allocation" [45] in SIMDRAM [22]. However, this design overlooks the granularity mismatch, which precludes minimizing the data movements on Bit-serial SIMD PUD architectures. On the contrary, CHOPPER (1) utilizes *bit-slicing* to break the granularity restriction in the state-of-the-art assumption (i.e. SIMDRAM [22]); and (2) introduces OBS, which consists of three new optimizations, to minimize data movements on Bit-serial SIMD PUD architectures. Such a methodology can also be beneficial for other PIM, to reduce data movements for the better efficiency.

X. Conclusions

We present CHOPPER, a new compiler infrastructure to improve both the programmability and efficiency of Bit-serial SIMD PUD architectures. CHOPPER ❶ provides a synchronous dataflow programming interface to improve the programmability; and ❷ incorporates new abstractions and optimizations, to improve the overall efficiency of Bit-serial SIMD PUD architectures. We quantitatively evaluate the performance of CHOPPER, and the results suggest great potentials of CHOPPER.

ACKNOWLEDGMENTS

We thank the anonymous reviewers from ASPLOS 2022, MICRO 2022 and HPCA 2023 for the feedback. Yaohua Wang and Ming-Chang Yang are corresponding authors of this paper. This work is supported in part by The Research Grants Council of Hong Kong SAR (Project Nos. CUHK14210320 and CUHK14208521), NSF of Hunan Province (Project No.2022JJ10066), and NSFC (Project No. 62272477).

REFERENCES

[1] "6th Generation Intel® Processor Families for S-Platforms," https://www.intel.com/content/www/us/en/processors/core/desktop-6th-gen-core-family-datasheet-vol-1.html.

[2] "JEDEC DDR4 SDRAM Standard," https://www.jedec.org/standards-documents/docs/module42027d.

[3] "NVIDIA TITAN V GPU Specifications," https://www.nvidia.com/en-us/titan/titan-v/.

[4] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. A. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: A System for Large-Scale Machine Learning," in *OSDI*, 2016.

[5] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A Scalable Processing-In-Memory Accelerator for Parallel Graph Processing," in *ISCA*, 2015.

[6] M. F. Ali, A. Jaiswal, and K. Roy, "In-Memory Low-Cost Bit-Serial Addition Using Commodity DRAM Technology," *IEEE TCSI*, 2020.

[7] M. Besta, R. Kanakagiri, G. Kwasniewski, R. Ausavarungnirun, J. Beránek, K. Kanellopoulos, K. Janda, Z. Vonarburg-Shmaria, L. Gianinazzi, I. Stefan, J. Gómez-Luna, J. Golinowski, M. Copik, L. Kapp-Schwoerer, S. D. Girolamo, N. Blach, M. Konieczny, O. Mutlu, and T. Hoefler, "SISA: Set-Centric Instruction Set Architecture for Graph Mining on Processing-in-Memory Systems," in *MICRO*, 2021.

[8] J. Bethge, H. Yang, M. Bornstein, and C. Meinel, "BinaryDenseNet: Developing an Architecture for Binary Neural Networks," in *ICCV Workshops*, 2019.

[9] E. Biham, "A Fast New DES Implementation in Software," in *FSE*, 1997.

[10] N. L. Binkert, B. M. Beckmann, G. Black, S. K. Reinhardt, A. G. Saidi, A. Basu, J. Hestness, D. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. S. B. Altaf, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, 2011.

[11] K. K. Chang, P. J. Nair, D. Lee, S. Ghose, M. K. Qureshi, and O. Mutlu, "Low-Cost Inter-Linked Subarrays (LISA): Enabling Fast Inter-Subarray Data Movement in DRAM," in *HPCA*, 2016.

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

[13] C. Eckert, X. Wang, J. Wang, A. Subramaniyan, R. R. Iyer, D. Sylvester, D. T. Blaauw, and R. Das, "Neural Cache: Bit-Serial In-Cache Acceleration of Deep Neural Networks," in *ISCA*, 2018.

[14] J. D. Ferreira, G. Falcão, J. Gómez-Luna, M. Alser, L. Orosa, M. Sadrosadati, J. S. Kim, G. F. Oliveira, T. Shahroodi, A. Nori, and O. Mutlu, "pLUTo: Enabling Massively Parallel Computation in DRAM via Lookup Tables," in *MICRO*, 2022.

[15] R. J. Fisher and H. G. Dietz, "Compiling for SIMD Within a Register," in *LCPC*, 1998.

[16] R. J. Fisher and H. G. Dietz, "The Scc Compiler: SWARing at MMX 3DNow!" in *LCPC*, L. Carter and J. Ferrante, Eds., 1999.

[17] R. J. Fisher, "General-Purpose SIMD Within a Register: Parallel Processing on Consumer Microprocessors," *PhD Thesis in Purdue University*, 2003.

[18] B. B. Fraguela, J. Renau, P. Feautrier, D. A. Padua, and J. Torrellas, "Programming the FlexRAM Parallel Intelligent Memory System," in *PPoPP*, 2003.

[19] C. Gao, X. Xin, Y. Lu, Y. Zhang, J. Yang, and J. Shu, "ParaBit: Processing Parallel Bitwise Operations in NAND Flash Memory based SSDs," in *MICRO*, 2021.

[20] F. Gao, G. Tziantzioulis, and D. Wentzlaff, "ComputeDRAM: In-Memory Compute Using Off-the-Shelf DRAMs," in *MICRO*, 2019.

[21] R. Grossi, A. Gupta, and J. S. Vitter, "High-Order Entropy-Compressed Text Indexes," in *SODA*, 2003.

[22] N. Hajinazar, G. F. Oliveira, S. Gregorio, J. D. Ferreira, N. Mansouri-Ghiasi, M. Patel, M. Alser, S. Ghose, J. Gómez-Luna, and O. Mutlu, "SIMDRAM: A Framework for Bit-serial SIMD Processing Using DRAM," in *ASPLOS*, 2021.

[23] M. W. Hall, P. M. Kogge, J. G. Koller, P. C. Diniz, J. Chame, J. Draper, J. LaCoss, J. J. Granacki, J. B. Brockman, A. Srivastava, W. C. Athas, V. W. Freeh, J. Shin, and J. Park, "Mapping Irregular Applications to DIVA, a PIM-based Data-Intensive Architecture," in *SC*, 1999.

[24] M. He, C. Song, I. Kim, C. Jeong, S. Kim, I. Park, M. Thottethodi, and T. N. Vijaykumar, "Newton: A DRAM-maker's Accelerator-in-Memory (AiM) Architecture for Machine Learning," in *MICRO*, 2020.

[25] J. L. Herlocker, J. A. Konstan, A. Borchers, and J. Riedl, "An Algorithmic Framework for Performing Collaborative Filtering," in *SIGIR*, 1999.

[26] J. L. Herlocker, J. A. Konstan, A. Borchers, and J. Riedl, "An Algorithmic Framework for Performing Collaborative Filtering," *SIGIR Forum*, 2017.

[27] G. Huang, Z. Liu, L. van der Maaten, and K. Q. Weinberger, "Densely Connected Convolutional Networks," in *CVPR*, 2017.

[28] B. Jacob, D. Wang, and S. Ng, *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann, 2010.

[29] Z. Jia, O. Padon, J. Thomas, T. Warszawski, M. Zaharia, and A. Aiken, "TASO: Optimizing Deep Learning Computation with Automatic Generation of Graph Substitutions," in *SOSP*, 2019.

[30] Y. Kim, V. Seshadri, D. Lee, J. Liu, and O. Mutlu, "A Case for Exploiting Subarray-Level Parallelism (SALP) in DRAM," in *ISCA*, 2012.

[31] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A Fast and Extensible DRAM Simulator," *CAL*, 2016.

[32] C. Lattner and V. S. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *CGO*, 2004.

[33] S. H. Lee, S. Kang, J. Lee, H. Kim, E. Lee, S. Seo, H. Yoon, S. Lee, K. Lim, H. Shin, J. Kim, S. O, A. Iyer, D. Wang, K. Sohn, and N. S. Kim, "Hardware Architecture and Software Stack for PIM Based on Commercial DRAM Technology : Industrial Product," in *ISCA*, 2021.

[34] M. Lenjani, P. Gonzalez-Guerrero, E. Sadredini, S. Li, Y. Xie, A. Akel, S. Eilert, M. R. Stan, and K. Skadron, "Fulcrum: A Simplified Control and Access Mechanism Toward Flexible and Practical In-Situ Accelerators," in *HPCA*, 2020.

[35] S. Li, D. Niu, K. T. Malladi, H. Zheng, B. Brennan, and Y. Xie, "DRISA: A DRAM-Based Reconfigurable In-Situ Accelerator," in *MICRO*, 2017.

[36] S. Li, C. Xu, Q. Zou, J. Zhao, Y. Lu, and Y. Xie, "Pinatubo: A Processing-In-Memory Architecture for Bulk Bitwise Operations in Emerging Non-Volatile Memories," in *DAC*, 2016.

[37] D. Mercadier and P. Dagand, "Usuba: High-Throughput and Constant-Time Ciphers, by Construction," in *PLDI*, 2019.

[38] D. Mercadier, P. Dagand, L. Lacassagne, and G. Muller, "Usuba: Optimizing & Trustworthy Bitslicing Compiler," in *WPMVP@PPoPP*, 2018.

[39] N. Mohammed, R. Chen, B. C. M. Fung, and P. S. Yu, "Differentially Private Data Release for Data Mining," in *SIGKDD*, 2011.

[40] S. Ollivier, S. Longofono, P. Dutta, J. Hu, S. Bhanja, and A. K. Jones, "CORUSCANT: Fast Efficient Processing-in-Racetrack Memories," in *MICRO*, 2022.

[41] M. Oskin, F. T. Chong, and T. Sherwood, "Active Pages: A Computation Model for Intelligent Memory," in *ISCA*, 1998.

[42] J. Park, R. Azizi, G. F. Oliveira, M. Sadrosadati, R. Nadig, D. Novo, J. Gómez-Luna, M. Kim, and O. Mutlu, "Flash-Cosmos: In-Flash Bulk Bitwise Operations Using Inherent Computation Capability of NAND Flash Memory," in *MICRO*, 2022.

[43] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "PyTorch: An Imperative Style, High-Performance Deep Learning Library," in *NIPS*, 2019.

[44] D. A. Patterson, T. E. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. E. Kozyrakis, R. Thomas, and K. A. Yelick, "A Case for Intelligent RAM," *IEEE Micro*, 1997.

[45] M. Poletto and V. Sarkar, "Linear Scan Register Allocation," *ACM TOPLAS*, 1999.

[46] T. Pornin, "Automatic Software Optimization of Block Ciphers using Bitslicing Techniques," *Ecole Normale Superieure*, 1999.

[47] T. Pornin, "Implantation Et Optimisation DES Primitives Cryptographiques," http://www.bolet.org/~pornin/2001-phd-pornin.pdf, 2001.
[48] S. Rixner, W. J. Dally, U. J. Kapasi, P. R. Mattson, and J. D. Owens, "Memory Access scheduling," in *ISCA*, 2000.
[49] V. Seshadri, Y. Kim, C. Fallin, D. Lee, R. Ausavarungnirun, G. Pekhimenko, Y. Luo, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "RowClone: Fast and Energy-Efficient In-DRAM Bulk Data Copy and Initialization," in *MICRO*, 2013.
[50] V. Seshadri, D. Lee, T. Mullins, H. Hassan, A. Boroumand, J. Kim, M. A. Kozuch, O. Mutlu, P. B. Gibbons, and T. C. Mowry, "Ambit: In-Memory Accelerator for Bulk Bitwise Operations Using Commodity DRAM Technology," in *MICRO*, 2017.
[51] J. Shun, "Parallel Wavelet Tree Construction," in *DCC*, 2015.
[52] A. Tavakkol, J. Gómez-Luna, M. Sadrosadati, S. Ghose, and O. Mutlu, "MQSim: A Framework for Enabling Realistic Studies of Modern Multi-Queue SSD Devices," in *FAST*, 2018.
[53] M. S. Q. Truong, E. Chen, D. Su, L. Shen, A. Glass, L. R. Carley, J. A. Bain, and S. Ghose, "RACER: Bit-Pipelined Processing Using Resistive Memory," in *MICRO*, 2021.
[54] Y. Wang, L. Orosa, X. Peng, Y. Guo, S. Ghose, M. Patel, J. S. Kim, J. Gómez-Luna, M. Sadrosadati, N. Mansouri-Ghiasi, and O. Mutlu, "FIGARO: Improving System Performance via Fine-Grained In-DRAM Data Relocation and Caching," in *MICRO*, 2020.
[55] William K. Zuravleff and Timothy Robinson, "Controller for A Synchronous DRAM that Maximizes Throughput by Allowing Memory Requests and Commands to be Issued Out of Order," 1997, US Patent 5,630,096.
[56] X. Xin, Y. Zhang, and J. Yang, "ELP2IM: Efficient and Low Power Bitwise Operation Processing in DRAM," in *HPCA*, 2020.