TECHNISCHE
UNIVERSITÄT
DRESDEN

CHAIR FOR
COMPILER
CONSTRUCTION

Faculty of Computer Science
Center for Advancing Electronics Dresden
Chair for Compiler Construction

Diploma Thesis

Implementing an MLIR-based Compiler for In-DRAM Computing

Tianrui Zheng
Born on: 18th February 1998 in Shaanxi, China
Matriculation number: 4999374

to achieve the academic degree

Diplomingenieur (Dipl.-Ing.)

First referee
Prof. Dr.-Ing. Jeronimo Castrillon

Second referee
Dr.-Ing. Asif Ali Khan

Supervisors
Dr. João Paulo Cardoso de Lima
Clément Fournier

Submitted on: 13th October 2025

TU Dresden Fakultät Informatik, Institut für Technische Informatik, Professur für Compilerbau

Task Description for Final Thesis (Diplomarbeit)

For:	**Tianrui Zheng**
Degree program:	Diplom Informationssystemtechnik
Matriculation number:	4999374
E-mail:	tianrui.zheng@mailbox.tu-dresden.de
Topic:	**Implementing an MLIR-based compiler for in-DRAM computing**

In DRAM computing refers to a set of techniques that exploit architectural details of DRAM cells to perform computation directly in memory. This can offer great improvement in throughput, as it conceptually uses a DRAM module as an extremely large SIMD processing unit. Memory traffic is also greatly reduced, as data can stay in memory. On the other hand, DRAM cells only support very simple operations on bits, like majority gate, which can be used to build other simple logic gates. Larger instructions, like an adder, have to be built using many of these primitives, which increases latency.

Existing compilers for in DRAM like Chopper (Peng et al., 2023) use specific DSLs as input, and do not support automatic transformation of higher-level programs to compute in DRAM. The goal of this project is to create a new compiler that can take an arbitrary program as input, and convert parts of it to use in DRAM computing techniques where beneficial. The infrastructure for this compiler will be MLIR, as it will allow this compiler to be integrated in the wider MLIR ecosystem. The compiler IR will be generic and able to handle core arithmetic operations like those from the `arith` dialect. As a more specific frontend flow, integration within the CINM framework (Khan et al., 2022) should be considered as part of this project. An additional context element is the ongoing work on a C++-based in-DRAM compiler by Niklas Jungnitz at the Chair for Compiler Construction. This work could be used as a backend to generate memory controller instructions, and enjoy integration with the RTSim hardware simulation framework (Khan et al., 2019) for evaluation purposes.

For this project, the student should design an MLIR dialect or set of dialects to model in DRAM compute primitives. This should include conversions from higher-level operations (eg from the `arith` dialect) to those primitives. The dialects should be carefully designed to take advantage of SIMD-like capabilities of DRAM hardware. To evaluate this project, the student should gather quantitative data about the performance of their implementation for some applications (like GEMV, GEMM, or others), either through simulation or running on hardware. The thesis should compare these numbers to a CPU implementation, and if time allows, against a comparable framework like Chopper. An optional extension to this task would be to integrate the new dialects within the CINM framework. The student should also study select related work, and compare them qualitatively with their work.

Start:	12.05.2025
End:	13.10.2025
1st referee:	Prof. Dr.-Ing. Jerónimo Castrillón
2nd referee:	Dr.-Ing. Asif Ali Khan
1st supervisor:	João Paulo Cardoso de Lima
2nd supervisor:	Clément Fournier

Jeronimo Castrillon Mazo
Digitally signed by Jeronimo Castrillon Mazo
Date: 2025.05.08 06:00:49 +02'00'

Prof. Dr.-Ing. Jerónimo Castrillón
(Professor in charge)

Martin Wollschlaeger
Digital unterschrieben von Martin Wollschlaeger
Datum: 2025.05.12 13:47:57 +02'00'

Prof. Dr.-Ing. habil. Martin Wollschlaeger
(Chairman of the Examination Board)

Statement of authorship

I hereby certify that I have authored this document entitled *Implementing an MLIR-based Compiler for In-DRAM Computing* independently and without undue assistance from third parties. No other than the resources and references indicated in this document have been used. I have marked both literal and accordingly adopted quotations as such. There were no additional persons involved in the intellectual preparation of the present document. I am aware that violations of this declaration may lead to subsequent withdrawal of the academic degree.

Dresden, 13th October 2025

Tianrui Zheng

Abstract

Recent advances in deep learning, especially large language models (LLMs), have intensified the data movement bottleneck inherent in traditional Von Neumann architectures, where frequent transfers between compute units and memory dominate energy consumption and limit performance. Near- and in-memory computing address this challenge by relocating computation closer to or even inside memory, thereby reducing data movement and improving efficiency.

This thesis focuses on bit-serial in-DRAM computing. While prior frameworks have demonstrated its potential, they often rely on domain-specific languages or fixed programming patterns, limiting generality. To overcome this, this work develops a general-purpose MLIR-based compiler that lowers arithmetic and logical operations from the generic `arith` dialect into a new `pud` dialect, which represents bulkwise in-DRAM computation primitives. The backend targets computation-capable DRAM controllers to generate executable memory instruction sequences. Evaluation across diverse workloads shows that our compiler achieves comparable performance and energy efficiency to prior work. These results demonstrate the flexibility and extensibility of our MLIR-based framework as a general-purpose compilation flow for in-DRAM computing. Our code is publicly available at [Zhe25].

Contents

Abstract . . . . **VII**

Symbols and Acronyms . . . . **XI**

1 Introduction **1**
- 1.1 Motivation 1
- 1.2 Goal 3
- 1.3 Structure of the Work 3

2 Background **5**
- 2.1 DRAM Basics 5
  - 2.1.1 DRAM Organization 5
  - 2.1.2 Basic DRAM Operations 6
  - 2.1.3 DRAM Performance Characteristics 7
- 2.2 In-DRAM Computing 7
  - 2.2.1 Ambit's Computing Substrates 8
  - 2.2.2 Row Address Organization 8
  - 2.2.3 AP and AAP Command Primitives 9
  - 2.2.4 Vertical Data Layout 9
- 2.3 Toolchain for In-DRAM Compilation 10
  - 2.3.1 Mockturtle 10
  - 2.3.2 Lime 11
- 2.4 MLIR 11
  - 2.4.1 Dialects 11
  - 2.4.2 Operations, Types and Attributes 12
  - 2.4.3 Lowering 12

3 Related Work **15**
- 3.1 SIMDRAM 15
- 3.2 MIMDRAM 16
- 3.3 CHOPPER 17
- 3.4 Discussion 18

4 Design and Implementation of the Compiler **21**
4.1 Overview of the Compilation Flow 21
4.2 Bits Dialect 22
4.2.1 Slice Type 23
4.2.2 Operations 24
4.3 PuD Dialect 26
4.3.1 Row Type 26
4.3.2 Operations 27
4.4 From Bits to PuD 27
4.4.1 Logic Synthesis 27
4.4.2 Address Allocation 29
4.4.3 Computation Process 31
4.5 Operand Tracking 36

5 Evaluation **39**
5.1 Computation Correctness Verification 39
5.1.1 Verification Method 39
5.1.2 Accuracy of Floating-point Multiplication Approximation 40
5.2 Experimental Setup 40
5.2.1 Baseline Selection 40
5.2.2 Workloads Specification 41
5.2.3 NVMain-PIM Simulator 41
5.2.4 Trace Emitting 42
5.3 Results Analysis 43

6 Conclusion and Further Work **47**
6.1 Future Work 48
6.1.1 Flexible Data Layout and Configuration 48
6.1.2 Bank-Level Parallelism 49
6.1.3 Advanced Data Mapping 49
6.1.4 Additional In-DRAM Computing Back-ends 49
6.1.5 Summary 50

7 Acknowledgements **51**

Symbols and Acronyms

AAP	ACTIVATE; ACTIVATE; PRECHARGE
AI	Artificial Intelligence
AIG	AND-Inverter Graph
AP	ACTIVATE; PRECHARGE
API	Application Program Interface
ASIC	Application-Specific Integrated Circuit
CNN	Convolutional Neural Network
CPU	Central Processing Unit
DCC	Dual-Contact Cell
DNN	Deep Neural Network
DRAM	Dynamic Random-Access Memory
DSA	Domain-Specific Accelerator
DSP	Digital Signal Processor
DSL	Domain-Specific Language
FCFS	First-Come, First-Serve
FRFCFS	First-Ready, First-Come, First-Serve
FP	Floating-Point
GPU	Graphics Processing Unit
IR	Intermediate Representation
LLM	Large Language Model
LSB	Least Significant Bit
MIMD	Multiple Instructions, Multiple Data
MSB	Most Significant Bit
MIG	Majority-Inverter Graph
MLIR	Multi-Level Intermediate Representation
NVM	Non-Volatile Memory
OS	Operating System
PUD	Processing-Using-DRAM
RRAM	Resistive Random-Access Memory
SIMD	Single Instruction, Multiple Data
SRAM	Static Random-Access Memory
SSA	Static Single-Assignment
TPU	Tensor Processing Unit
TRA	Triple-Row Activation
XIG	XOR-Inverter Graph

1 Introduction

1.1 Motivation

For many years, Moore's Law [Moo65] - the idea that the number of transistors on a chip doubles roughly every two years - drove the steady growth of computing performance. However, as transistors approach their physical limits, the benefits of Moore's Law have been fading, marking the transition to what is often called the *post-Moore era* [ITR15; Wal16]. Even more critically, Dennard scaling [Den+74] - the principle that power density stays constant as transistors shrink - broke down in the early 2000s. As a result, further increases in clock frequency caused excessive power use and heat generation, bringing the steady rise in single-core performance to a halt [Esm+11].

To address these limitations, the semiconductor industry has increasingly shifted toward **Domain-Specific Accelerators (DSAs)**, which achieve superior performance and energy efficiency by tailoring hardware to specific application domains. Examples include:

- **Graphics Processing Units (GPUs)**, initially designed for rendering graphics but later exploited for highly parallel workloads such as deep learning training and inference
- **Digital Signal Processors (DSPs)**, optimized for embedded and mobile markets, often consuming only 10% of the power of a typical x86 Central Processing Unit (CPU) while providing significantly better floating-point performance per watt [Ham+10]
- **Tensor Processing Units (TPUs)**, custom Application-Specific Integrated Circuits (ASICs) by Google, which deliver substantial speedups for neural network inference by exploiting matrix-multiplication workloads [Jou+17]

These examples reflect a broader trend: as scaling of general-purpose CPUs slows, specialized hardware has become the main path to sustaining gains in performance and energy efficiency. Industry reports identify DSAs as a key driver of innovation in the post-Moore era [McK20; Kha+24], and academic research likewise highlights their growing role in areas ranging from high-performance computing to machine learning [Nie+24; Hu+25].

In recent years, advances in machine learning, particularly Convolutional Neural Networks (CNNs) and more recently Large Language Models (LLMs), have driven remarkable breakthroughs across Artificial Intelligence (AI). These models have achieved state-of-the-art results in domains ranging from image classification and object detection to natural language understanding and generation [KSH17] [Dev+19]. At the same time, the training phase of such models has grown increasingly demanding. Training a modern LLM, for instance, may require hundreds of billions of parameters and terabytes of training data, translating into exaflop-scale computations sustained over weeks on large GPU clusters [Nar+21].

These characteristics impose new and stringent requirements on hardware. On the one hand, the enormous volume of parallel computation has fueled the adoption of massively parallel accelerators such as GPUs and TPUs, which can deliver the necessary throughput for matrix-multiplication-dominated workloads. On the other hand, the **data-centric** nature of machine learning exposes severe memory bottlenecks: moving large tensors between compute cores and off-chip Dynamic Random-Access Memory (DRAM) consumes substantial energy and limits overall system efficiency. It has been reported that data movement, rather than computation, accounts for 40–60% of total system energy consumption in many machine learning workloads [Hor14]. As model sizes continue to grow, these bottlenecks increasingly dominate both performance scaling and energy consumption, motivating new architectural paradigms such as near-memory and in-memory computing.

The central idea behind near-memory and in-memory computing is to reduce the costly data movement between processing units and memory, which has become the dominant factor in both performance and energy consumption. The former integrates compute units close to the memory banks or within the memory controller. By reducing the distance data must travel, near-memory computing relieves the memory bandwidth wall and lowers energy overhead. In-memory computing goes one step further by embedding computation directly inside the memory arrays themselves. In this paradigm, memory cells can be repurposed to perform simple arithmetic or bit-wise operations, thereby eliminating most data transfers [Lim+24; Lim+25].

A particularly promising branch of in-memory computing is in-DRAM computing, which leverages the intrinsic analog behavior of DRAM circuitry to perform operations directly within the memory arrays. Unlike near-memory computing, which integrates logic units into the memory controller or 3D-stacked memory logic layers, in-DRAM techniques exploit existing DRAM structures such as sense amplifiers and row buffers to realize data movement and bit-wise operations with minimal changes to commodity DRAM chips.

Representative works demonstrate the efficiency of this approach. [Ses+13] performs bulk data copy and initialization entirely inside DRAM through row-buffer transfers, achieving up to 11× lower latency and 74× lower energy than copying using CPU. [Ses+16] extends this idea to bulk bit-wise operations by exploiting Triple-Row Activation (TRA), yielding 10–25× higher throughput and 25–59× lower energy compared to conventional execution. [Ses+17] further generalizes in-DRAM bit-wise acceleration with modest hardware overhead (<1% DRAM area), improving throughput and energy efficiency by 32× and 35× respectively. These results highlight the main advantage of in-DRAM computing: it directly eliminates costly data transfers over the memory channel while utilizing the full internal bandwidth of DRAM subarrays.

Beyond early primitives above, a number of works have sought to extend this in-DRAM computing paradigm toward more general or domain-specific acceleration. For example, [Haj+21] and [Oli+24] propose a general-purpose Single Instruction, Multiple Data (SIMD) and Multiple Instructions, Multiple Data (MIMD) execution model. By packaging sequences of memory instructions such as TRAs and Row-Clones into micro-programs (μPrograms), it supports a broad range of bit-wise and arithmetic kernels in a programmable fashion. Another representative effort is [PWY23], which emphasizes flexible in-DRAM acceleration for machine learning workloads. It proposes a high-level programming model that maps neural network operators onto in-DRAM bit-serial execution.

Nevertheless, they share a common limitation: their programmability is tightly coupled to specific domains, operators, or Domain-Specific Languages (DSLs). [Haj+21] and [Oli+24] expose programmability only through predefined bit-serial kernels (C macros), while [PWY23] (its compiler is built upon a DSL named `Usuba` [MD19]) and related frameworks are restricted to neural networks. As such, they lack the generality and flexibility required for serving as a compiler backend across diverse application domains. This observation motivates this work: to design a general-purpose compilation flow, based on Multi-Level Intermediate Representation (MLIR) framework, that bridges high-level arithmetic operations and low-level in-DRAM primitives in a systematic and extensible manner.

1.2 Goal

The goal of this work is to develop an MLIR-based compiler targeting bit-serial bulkwise in-DRAM computing.

To ensure generality and flexibility, our compiler adopts the `arith` dialect as the front-end, since within the MLIR ecosystem it serves as a generic representation of arithmetic and logical operations. For instance, operations defined in machine learning frameworks such as **PyTorch** can be lowered via `torch-mlir` into the `linalg` dialect, from which fundamental operators are further lowered into `arith` representation, augmented with the `scf` or `affine` dialects to capture control flow and loop information.

This work introduces two new dialects to describe bulkwise in-DRAM computing primitives, thereby exploiting the bit-serial, SIMD-like massive parallelism inherent in this computing model. On the back-end, the compiler targets DRAM controllers augmented with computation capabilities, ultimately generating memory instruction sequences for computation, exemplified by primitives such as `ACTIVATE; PRECHARGE (AP)` and `ACTIVATE; ACTIVATE; PRECHARGE (AAP)`.

For evaluation, this work compares against prior efforts such as Chopper [PWY23] and MIMDRAM [Oli+24]. Specifically, we assess the generated memory instruction sequences for equivalent workloads, and analyze performance and energy efficiency metrics obtained through simulation.

1.3 Structure of the Work

The remainder of this thesis is organized as follows.

Chapter 2 provides the necessary background for understanding the work. It first introduces the organization, basic operations, and performance characteristics of DRAM, and then reviews in-DRAM computing with a focus on Ambit's computing substrates and related primitives. The chapter also presents the tool-chain employed in this thesis, including the `mockturtle` logic synthesis framework and the Lime code generator, before concluding with an overview of MLIR and its dialect-based compilation infrastructure.

Chapter 3 surveys related work. We discuss SIMDRAM [Haj+21] and MIMDRAM [Oli+24] as two representative programmable frameworks for in-DRAM computing, and CHOPPER [PWY23] as the most closely related compiler-based approach. The chapter concludes with a summary highlighting the differences between prior work and our MLIR-based compiler.

Chapter 4 presents the design and implementation of our compiler. It outlines the overall compilation flow and introduces the newly designed `bits` and `pud` dialects. Furthermore, it details the progressive lowering from high-level arithmetic operations to in-DRAM instruction sequences, covering logic synthesis, address allocation, computation process, and operand tracking.

Chapter 5 evaluates the proposed compiler. We first verify computation correctness and analyze the approximation accuracy of Floating-Point (FP) multiplication. We then describe the experimental setup, including workload specification, simulator configuration, and trace emitting, followed by a detailed analysis of performance and energy results compared to prior work.

Finally, Chapter 6 concludes the thesis and outlines directions for future research.

2 Background

Modern in-memory computing techniques frequently rely on the fundamental behavior and internal structure of DRAM. To motivate the architectural mechanisms exploited later in this work, we first revisit the basics of commodity DRAM organization and operation. This section outlines the hierarchical layout of DRAM, its core access primitives, and the timing and performance characteristics that shape both its traditional role as main memory and its emerging use as a substrate for computation.

2.1 DRAM Basics

Dynamic Random-Access Memory (DRAM) is the predominant technology used in modern main memory systems due to its high density, low cost per bit, and relatively low idle power consumption. Although DRAM is conceptually simple - storing each bit as charge in a capacitor - its internal organization and operation involve multiple layers of hierarchy and complex timing constraints. A proper understanding of DRAM structure and access mechanisms is essential for analyzing both its limitations and its potential for architectural extensions. This section provides an overview of the fundamental organization, operations and performance characteristics of commodity DRAM.

2.1.1 DRAM Organization

As shown in Figure 2.1, DRAM system is organized hierarchically. At the top level, a memory **channel** provides a physical interface between the memory controller and one or more DRAM modules. Each module typically consists of multiple **ranks**, where a rank is a set of DRAM chips that operate in lockstep to provide a wide data bus (e.g., 64 bits). Within each rank, the storage is divided into multiple **banks** (commonly 8-16 in DDR4/DDR5 systems). Banks are the fundamental units of parallelism: different banks can be activated and accessed independently, allowing overlap of memory requests.

Each bank is further subdivided into multiple **subarrays**. A subarray is a two-dimensional array of DRAM **cells**, typically organized into thousands of rows and columns. Each row spans across the subarray and connects to a set of sense amplifiers, collectively known as the **row**

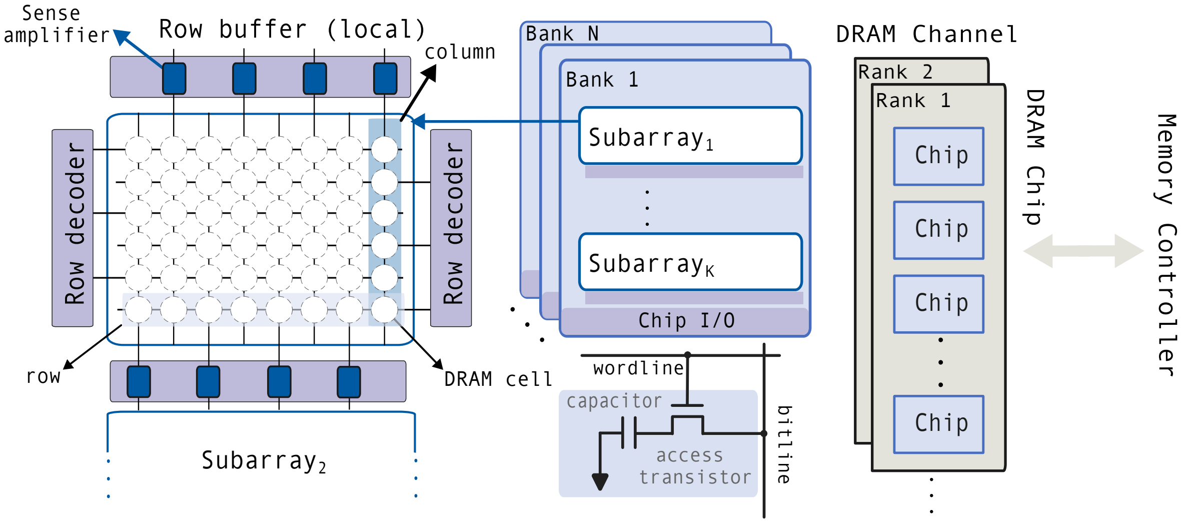

Figure 2.1: Overview of DRAM Organisation [Lim+26]

buffer. When a row is activated, the contents of the entire row are transferred into the row buffer, from which individual columns can be read or written.

Within each subarray, the basic storage element is the DRAM **cell**, consisting of a single capacitor and an access transistor. The transistor connects the capacitor to the bit-line when enabled by a word-line. Since capacitors are leaky, DRAM cells must be periodically refreshed to restore their charge, typically every 32–64 ms. The cell's simplicity allows extremely high density, but also makes it volatile and slower to access compared to Static Random-Access Memory (SRAM).

Some modern DRAM chips further group subarrays into smaller two-dimensional structures called **mats**. A mat consists of hundreds of rows and columns, with its own local sense amplifiers. The mats are tiled to form a full subarray, and global interconnects coordinate their operation. This mat-level structure is not visible to software but is important for understanding access granularity and potential internal parallelism.

2.1.2 Basic DRAM Operations

Accessing DRAM involves a sequence of fundamental operations orchestrated by the memory controller. The three key commands are **activate**, **read/write**, and **precharge**.

Activate (ACT): An activate command opens a row by raising its word-line, thereby connecting all cells in that row to their respective bit-lines. The tiny charge stored in each capacitor slightly perturbs the bit-line voltage. The corresponding sense amplifier then detects this deviation, amplifies it to a full logic level, and stores the value in the row buffer. At the same time, the sense amplifier refreshes the charge in the capacitor, restoring the data. Once a row is activated, it remains in the row buffer until explicitly precharged.

Read/Write (RD/WR): After a row is active, read and write operations can be issued to access specific columns within the row buffer. A read copies the data from the sense amplifier to the I/O bus, while a write updates both the sense amplifier and the underlying cell. Since the entire row is already sensed and buffered, column accesses are relatively fast as long as they target the currently open row.

Precharge (PRE): To access a different row within the same bank, the current row must first be closed. This is achieved by precharging, which restores the bit-lines to a neutral voltage and disconnects the active word-line. Only after precharge can another activate be issued in the same bank. The precharge step ensures signal integrity but adds latency to row-to-row accesses.

Refresh (REF). Because charge in the capacitors leaks over time, DRAM requires periodic refresh operations to prevent data loss. A refresh essentially re-activates and restores the charge of rows across all banks on a fixed schedule (e.g., every 64 ms). Refresh operations occupy memory cycles and contribute to both energy consumption and performance variability.

2.1.3 DRAM Performance Characteristics

From a system perspective, DRAM performance is shaped by the interaction of row buffers, bank-level parallelism, and refresh overheads. A key factor is whether an access results in a **row buffer hit** (the requested row is already active), a **row buffer miss** (the bank is idle and a new row must be activated), or a **row buffer conflict** (a different row is currently active and must be precharged before the new row can be activated). Row buffer hits yield the lowest latency and energy, while conflicts incur the highest overhead due to the full precharge–activate sequence.

Bank-level parallelism allows simultaneous accesses to different banks, partially hiding individual bank latencies. However, the global memory channel bandwidth still limits overall throughput, as all banks within a channel share the same I/O interface. Subarray-level parallelism has also been proposed in research, where subarrays within the same bank can be activated independently to increase concurrency, though commodity DRAM typically restricts one active row per bank at a time.

Energy consumption in DRAM is dominated by three sources: activating rows (charging large word-lines and bit-lines), refreshing cells, and transferring data across the off-chip bus. Among these, data movement between DRAM and the processor is often the most expensive, motivating the exploration of architectures that reduce such transfers.

2.2 In-DRAM Computing

This section introduces the core ideas behind in-DRAM computing as Ambit proposed in [Ses+17], an in-memory accelerator that performs bulk bit-wise operations directly within commodity DRAM arrays by carefully orchestrating multi-row activation and sense amplification. We first explain Ambit's compute substrate and primitive operations, then describe the row-address organization and command sequences that make these operations practical, and finally discuss data layout considerations (especially the vertical layout used by bit-serial SIMD computation).

2.2.1 Ambit's Computing Substrates

Ambit's central observation is that simultaneously activating three rows that share a sense amplifier causes the bit-line to resolve to the **majority** value of the three storage capacitors connected to that bit-line. As shown in Figure 2.2, this follows from (i) charge sharing from the precharged bit-line to the three capacitors and (ii) the positive feedback of the cross-coupled inverters that complete sensing: if at least two of the three cells are charged (or discharged), the sense amplifier resolves to 1 (or 0), and all three cells are then overwritten with that majority value. Ambit exposes this as a Triple-Row Activation (TRA), which implements a 3-input majority operation MAJ on every bit-line in parallel. Formally, $\mathrm{MAJ}(A, B, C) = A \cdot B + A \cdot C + B \cdot C$.

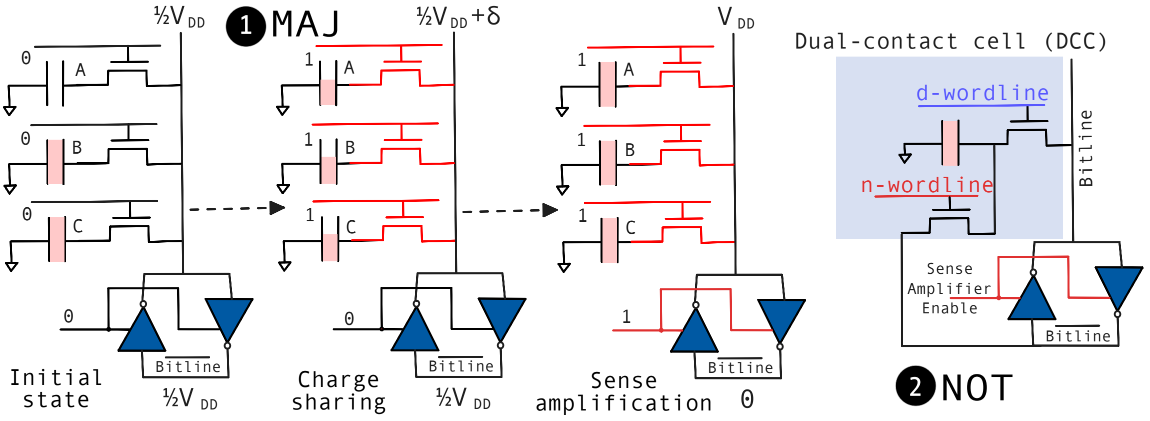

Figure 2.2: Majority and Not Logic Implementation of Ambit [Lim+26]

Because MAJ is functionally complete with a constant input, Ambit realizes bit-wise `AND` and `OR` by using a third "control" row that is pre-initialized to all-zeros or all-ones: $\mathrm{MAJ}(A, B, C_0) = A \wedge B$ and $\mathrm{MAJ}(A, B, C_1) = A \vee B$. To support this efficiently, Ambit reserves two special control rows per subarray, C_0 (all 0s) and C_1 (all 1s), which can be selected via dedicated control-row addresses.

Ambit implements `NOT` with a **Dual-Contact Cell (DCC)**. As presented in Figure 2.2, DCC connects a single storage capacitor to either side of the cross-coupled sense amplifier through two independent word-lines, commonly referred to as the **d**- and **n**- word-lines. By first activating a source row to let the sense amplifier resolve to the source value and then activating the DCC's **n**-word-line, Ambit forces the capacitor connected through the complementary side of the sense amplifier to charge to the inverted value. The bank is then precharged, and a Row-Clone copy moves the inverted data from the DCC's **d**-side into the destination row. Ambit's controller sequences these steps as: `ACT(src)` → `ACT(DCC_n)` → `PRE` → `Row-Clone(DCC_d` → `dst)`.

2.2.2 Row Address Organization

To make in-DRAM computation practical and to keep the software-visible address space contiguous, Ambit partitions the row addresses inside each subarray into three disjoint groups:

- **B-Group** (bitwise/computation rows), containing designated TRA rows (T_0–T_3) and DCC rows;

- **C-Group** (control rows), two rows C_0 and C_1 that store all-zeros and all-ones;
- **D-Group** (data rows), which hold regular application data and are the only rows exposed to the Operating System (OS) and applications.

A small B-group row decoder maps a set of reserved B-addresses (e.g., $B0-B15$) to one or multiple physical word-lines, enabling single-, double-, or triple-row activation with a single address. For example, addresses $B12$–$B15$ simultaneously drive three word-lines for TRA, while $B8-B11$ activate two word-lines to aid in result duplication. The B-group decoder provides addresses that directly trigger double/triple activations (e.g., a single $B12$ may assert three rows T_0, T_1, and T_2), minimizing controller overhead while preserving commodity interfaces. This grouping is widely adopted in subsequent work as the canonical Ambit layout (B/C/D groups).

2.2.3 AP and AAP Command Primitives

Ambit introduces two composite command sequences that the memory controller issues to DRAM:

- **ACTIVATE; PRECHARGE (AP)** A multi-row activation (e.g., TRA for MAJ) followed by a `PRE` to close the bank. `AP` encapsulates logic operations (e.g., MAJ/AND/OR) that intentionally activate multiple rows before a precharge;
- **ACTIVATE; ACTIVATE; PRECHARGE (AAP)** Two back-to-back `ACT` commands, from source to destination, followed by `PRE`. `AAP` is the Row-Clone intra-subarray fast-copy primitive that transfers the row-buffer contents into another row by overdriving the destination cells via the sense amplifier.

Together, `AP` and `AAP` form the building blocks of Ambit's instruction sequences: `AP` for computation, `AAP` for data movement and initialization (e.g., preparing temporaries, duplicating results).

Using an example, we illustrate typical sequences to compute single-bit `AND` two D-group rows A and B into a D-group destination R.

1. Use `AAP` to copy A and B from D-group into two B-group designated rows T_0 and T_1, and copy C_0 (all zeros) to T_2.
2. `AP` to triple-activate T_0, T_1, and T_2 to compute $\text{MAJ}(T_0, T_1, T_2) = T_0 \wedge T_1$.
3. `AAP` to copy the result T_0 from the designated result row to R.

2.2.4 Vertical Data Layout

While Ambit itself focuses on bulk bit-wise logic, later in-DRAM frameworks execute arithmetic by composing bit-wise primitives in a **bit-serial, row-parallel** manner. A key enabler is the **vertical** data layout: instead of storing a full data element across columns within a single row (horizontal layout), each row stores the same bit position (e.g., all Least Significant Bits (LSBs)) of many elements aligned vertically along bit-lines. Figure 2.3 illustrates the difference between horizontal and vertical data layouts within a DRAM subarray. Activating a single row then exposes one bit from each element simultaneously across all columns, turning every bit-line

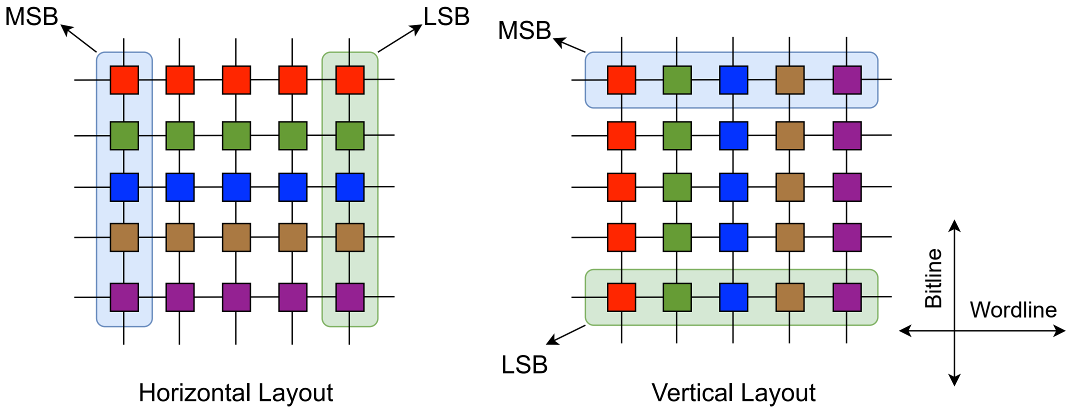

Figure 2.3: Horizontal and Vertical Data Layout in DRAM Subarray

into a SIMD lane and making bit-serial arithmetic efficient; bit shifts become row copies (e.g., shifting by one equals copying row j to row $j+1$). This layout is now standard in end-to-end PUD frameworks such as SIMDRAM [Haj+21] and underpins the "columns-as-lanes" execution model common to Ambit-based designs.

In summary, Ambit realizes in-DRAM computing by exploiting the sense amplifier's analog behavior under multi-row activation. TRA yields a per-bit-line majority operator that, combined with control rows, implements `AND`/`OR`; DCC implements `NOT`. A small set of reserved addresses and a dedicated B-group decoder enable single-address multi-row activation and efficient duplication, while `AP` and `AAP` command sequences unify logic and data movement. Organizing operands in a **vertical** layout turns columns into SIMD lanes, enabling word/row-parallel, bit-serial computation that subsequent systems generalize to rich arithmetic.

2.3 Toolchain for In-DRAM Compilation

Developing a compilation flow for in-DRAM computing requires dedicated tools that can bridge the gap between high-level arithmetic operations and low-level memory instruction sequences. In this thesis, we build upon two key components: the `mockturtle` [Soe+22] logic synthesis framework and the `lime` mapping tool. Together, they provide a systematic toolchain for translating arithmetic operators into Ambit-style instruction sequences for in-DRAM computing architectures.

2.3.1 Mockturtle

`Mockturtle` [Soe+22] is a modern `C++` logic network library designed for synthesis, optimization, and verification of digital circuits. It provides a variety of network representations, including AND-Inverter Graphs (AIGs), XOR-Inverter Graphs (XIGs), and Majority-Inverter Graphs (MIGs). A key feature of `mockturtle` is its modular design, which enables easy experimentation with different logic structures and synthesis strategies.

In the context of in-DRAM computing, we rely on `mockturtle` to synthesize arithmetic operators such as adders, subtractors, and multipliers into MIG networks. The MIG abstraction is particularly relevant because the majority operation directly aligns with the TRA primitive available in Ambit. By mapping arithmetic functions into a network of majority and inverter nodes, `mockturtle` provides an Intermediate Representation (IR) that naturally corresponds to in-DRAM bit-wise operations.

2.3.2 Lime

While MIG networks provide a logical abstraction, they must be further mapped into memory instruction sequences that can be executed on DRAM hardware. For this purpose, we employ `Lime` [Jun25], a tool designed to translate logic networks generated with the mockturtle framework into composite Ambit execution models. In our case, `lime` takes a MIG network as input and generates corresponding pseudo-code consisting of `AP` and `AAP` instructions, which are the basic operation primitives in Ambit.

The pseudo-code generated by `lime` specifies how each majority and inverter gate in the network is realized through in-DRAM commands, while respecting the resource constraints of subarrays and row groups. This allows complex arithmetic functions to be decomposed into a sequence of DRAM-native instructions that preserve functional correctness of the MIG network input.

2.4 MLIR

The Multi-Level Intermediate Representation (MLIR) is a compiler infrastructure designed to unify and extend compilation across heterogeneous systems and Domain-Specific Accelerators (DSAs). Unlike traditional Intermediate Representations (IRs) that operate at a single level of abstraction, MLIR explicitly supports multiple layers of abstraction coexisting in the same framework. This makes it possible to represent high-level tensor algebra, loop transformations, memory optimizations, and low-level instructions within a single infrastructure. By providing extensibility, modularity, and reusable optimization passes, MLIR has rapidly become a foundation for building compilers in areas ranging from machine learning frameworks to hardware synthesis.

2.4.1 Dialects

A central concept in MLIR is the **dialect**. Each dialect defines its own set of operations, types, and attributes, enabling developers to introduce domain-specific abstractions while reusing the shared infrastructure of MLIR. Dialects serve both as front-end abstractions for specific application domains and as intermediate layers for progressive lowering toward hardware back-ends.

Here we give several examples:

- `linalg`: Provides structured tensor operations such as `matmul` and `conv`, as well as the more generic `linalg.generic`. These operations make iteration spaces and data access patterns explicit, enabling optimizations such as tiling, fusion, and vectorization.

- `arith`: A generic dialect that models scalar, vector, and tensor arithmetic operations. It serves as a common denominator for representing basic arithmetic and logical primitives across the MLIR ecosystem, and is often the target of lowering from higher-level dialects.
- `affine`: Provides affine loops and memory access patterns with statically analyzable properties. This enables loop transformations such as interchange, skewing, and tiling to be applied in a structured and provably correct manner.

Together, these dialects illustrate the flexibility of MLIR. Complex operators expressed in machine learning frameworks can be lowered to structured forms in `linalg`, then refined to primitive arithmetic in `arith`, and further optimized using affine control flow. At the same time, developers can define new dialects to represent hardware-specific primitives, as will be the case in this thesis for in-DRAM operations.

2.4.2 Operations, Types and Attributes

At the core of MLIR are three fundamental concepts: **operations**, **types**, and **attributes**.

Operations are the primary building blocks of computation. They follow Static Single-Assignment (SSA) form, consuming values as operands and producing new values as results. Operations may be nested in regions and blocks, which allow MLIR to represent control flow and data flow explicitly within the same model.

Types describe the nature of values produced and consumed by operations. They define semantic information such as structure, dimensionality, or domain-specific constraints. The type system in MLIR is fully extensible, allowing new types to be introduced by dialects to model their respective domains.

Attributes represent immutable metadata associated with operations or types. They capture compile-time constants, configuration parameters, or symbolic properties that influence how operations are interpreted or transformed. Attributes are not values flowing through the program but rather annotations that refine semantics and guide transformations.

Together, operations, types, and attributes form the foundation of MLIR's intermediate representation. They provide a consistent and extensible way to model computations, enabling both high-level reasoning and low-level hardware mapping within a unified framework.

2.4.3 Lowering

One of the distinctive features of MLIR compared to traditional compiler infrastructures is its systematic support for **lowering** across multiple levels of abstraction. In MLIR, lowering refers to the process of progressively transforming operations expressed in one dialect into semantically equivalent operations in another dialect that lies closer to the hardware or target backend.

Lowering in MLIR is typically organized as a staged process. High-level dialects, such as `linalg`, express computations in a structured and domain-oriented form. These operations can then be lowered into more generic primitives in the `arith` dialect combined with loop constructs from `scf` or `affine`. Subsequently, value-based representations such as tensors are converted into memory-based representations through bufferization, producing `memref`-based

IR. At the final stage, the program is lowered into dialects that directly correspond to hardware targets, such as `llvm`, `spirv` or `pud` in this work.

The lowering process is guided by well-defined conversion rules. Each stage specifies a set of **legal** dialects, and all operations must be rewritten into these dialects before compilation can proceed. This mechanism ensures that the IR remains consistent, well-typed, and executable at each level. Moreover, because MLIR allows multiple dialects to coexist, developers can insert custom dialects into the lowering pipeline, ensuring that domain-specific abstractions can be represented and later refined without breaking the overall flow.

In summary, the design of MLIR not only makes it inherently **extensible** - allowing new dialects to be introduced with minimal integration effort - but also highly **general**, as existing compiler components such as passes, optimizations, and analyses can be reused across dialects. In the context of this thesis, these properties form the foundation of our proposed in-DRAM compiler: new dialects such as `bits` and `pud` can be seamlessly integrated into the MLIR ecosystem, while front-end compatibility with the `arith` dialect ensures that computations expressed in diverse high-level frameworks (e.g., C or PyTorch) can be uniformly lowered to memory-instruction-level representations. This combination of extensibility and generality is key to making in-DRAM computing accessible within a unified compilation infrastructure.

3 Related Work

Several compiler frameworks have been proposed to bridge high-level applications with in-memory and near-memory substrates [Ahm+19; Kha+25; Far+24a; Far+24b]. These efforts explore different abstractions, intermediate representations, and mapping strategies to exploit the execution models of emerging memory-centric architectures. In this work, we narrow our scope to those that explicitly target in-DRAM computing. This focus allows us to compare approaches that share similar device constraints, architectural primitives, and data-movement characteristics.

3.1 SIMDRAM

SIMDRAM [Haj+21] is a prominent effort that generalizes in-DRAM computing into a programmable framework, providing a systematic way for software developers to exploit the computational capabilities of commodity DRAM. Rather than proposing a fixed set of hardware primitives, SIMDRAM focuses on programmability: it introduces an abstraction model, a compilation flow, and a set of optimizations that enable diverse bit-wise and arithmetic kernels to be mapped into DRAM execution.

At its core, SIMDRAM presents DRAM subarrays as massively parallel bit-serial SIMD engines. Each row in a subarray is treated as a bit-slice, and TRA is exploited to implement majority logic, which, combined with inversion and data movement primitives, is sufficient to realize general Boolean computation. This abstraction allows programmers to conceptualize DRAM as a parallel execution substrate, not merely as a storage device. Through this SIMD-like model, kernels such as addition, multiplication, reduction, and comparison can be described in a uniform manner while being executed in parallel across DRAM subarrays.

To translate high-level computations into DRAM-executable instructions, SIMDRAM introduces the concept of **micro-programs**. A micro-program is a reusable building block that encapsulates the `AP` and `AAP` command sequences required to implement a specific bit-serial operation, such as a one-bit full adder or comparator. These micro-programs are exposed to programmers as C macros, which can be invoked repeatedly to construct larger kernels. For instance, an addition kernel is built by chaining the full-adder micro-program across successive bit positions, with carries propagated through the subarray.

This design provides a clean programming interface: programmers work with high-level macros rather than raw memory instructions, while the compiler expands these macros into the corresponding Ambit-compatible instruction sequences. The compiler also manages row allocation for inputs, outputs, and intermediate values, and ensures that all scheduling respects the structural constraints of DRAM subarrays. In this way, SIMDRAM hides the complexity of low-level DRAM operations while preserving performance efficiency.

Beyond the programming interface, SIMDRAM incorporates optimizations to reduce instruction count and improve resource utilization. These include efficient handling of carry chains in arithmetic operations, reuse of intermediate values within the B-group rows, and minimization of redundant `AAP` instructions by exploiting locality of intermediate data. Such optimizations make the generated micro-programs both reusable and efficient, enabling SIMDRAM to achieve performance and energy benefits close to those of hand-tuned implementations.

The framework supports a wide range of arithmetic and logical functions, going well beyond the limited set of primitives introduced by Ambit. By providing a programmable interface with micro-programs as reusable modules, SIMDRAM generalizes in-DRAM computing to a flexible platform that can accelerate diverse workloads.

In summary, SIMDRAM represents a shift from fixed-function in-DRAM primitives to a programmable framework. Its abstraction of DRAM as a SIMD-like engine, combined with the micro-program interface and compiler support, establishes a programming model that is both expressive and efficient. However, its programmability is still constrained by the reliance on predefined micro-programs, which limits generality across application domains. These limitations motivate further research into more flexible compilation frameworks for in-DRAM computing.

3.2 MIMDRAM

MIMDRAM [Oli+24] extends SIMDRAM by addressing one of its key limitations: the strict SIMD execution model. While SIMDRAM forced all subarrays to execute the same instruction stream in lockstep, MIMDRAM introduces a fine-grained SIMD model, effectively enabling **Multiple Instructions, Multiple Data (MIMD)**-style execution inside DRAM. This fundamental change provides significantly greater flexibility and efficiency, allowing different subarrays to execute distinct micro-programs concurrently, rather than being constrained to a uniform sequence of operations.

The central innovation of MIMDRAM is its hybrid execution model, which combines the benefits of SIMD-style parallelism with the flexibility of MIMD. In practice, each subarray can be assigned an independent instruction stream, meaning that multiple, heterogeneous kernels can be executed at the same time across different subarrays. This fine-grained control contrasts with SIMDRAM's global synchronization requirement, where all subarrays had to follow the same micro-program step by step.

The implications of this shift are substantial. First, it enables higher throughput by exploiting workload diversity: for example, one subarray can perform additions while another executes multiplications. Second, it allows better utilization of DRAM resources by reducing idle time when different operators are chained together. Finally, it introduces opportunities for subarray-level scheduling, a feature absent in SIMDRAM.

Building on SIMDRAM's concept of micro-programs, MIMDRAM generalizes them into **μPrograms**. A μProgram encapsulates the `AP` and `AAP` command sequences needed to realize specific operations, but unlike SIMDRAM's fixed C macro library, μPrograms can be composed, scheduled, and mapped independently to different subarrays. This decoupling from a global SIMD lockstep allows the programming model to scale beyond predefined patterns, giving developers more flexibility while still hiding low-level DRAM details.

To manage this extended flexibility, MIMDRAM introduces enhanced compiler strategies. The compiler translates high-level arithmetic and logic functions into μPrograms and distributes them across subarrays. It incorporates optimizations for row allocation, intermediate value reuse, and local scheduling, which reduce redundant `AAP` operations and data transfers. Importantly, the compiler ensures correctness while enabling concurrent execution of heterogeneous kernels, a capability not possible in SIMDRAM.

Evaluation shows that MIMDRAM achieves lower instruction counts, shorter execution time, and reduced energy compared to SIMDRAM, particularly for workloads involving multiple arithmetic operators. Its ability to execute different μPrograms concurrently across subarrays directly translates into better throughput and resource efficiency.

3.3 CHOPPER

CHOPPER [PWY23] presents a compiler infrastructure explicitly designed for bit-serial SIMD Processing-Using-DRAM (PUD) architectures, aiming to address two key challenges: poor programmability and low efficiency. Previous proposals such as SIMDRAM provided powerful PUD primitives but relied heavily on hand-crafted, assembly-like instruction sequences, which significantly limited usability. CHOPPER fills this gap by introducing a systematic compilation flow that automates code generation, memory management, and optimization for PUD, thereby making in-DRAM computing more practical.

CHOPPER is implemented on top of the existing **Usuba** [MD19] compiler, a domain-specific compiler originally developed for bit-sliced cryptographic applications. By extending Usuba's compilation infrastructure, CHOPPER is able to reuse its front-end parsing and IRs while introducing a back-end that targets in-DRAM execution. This choice lowers implementation effort but also means that CHOPPER's programming model is closely tied to Usuba's DSL, which restricts its applicability outside of the data-flow style supported by Usuba.

To further enhance programmability, CHOPPER introduces the concept of a **Virtual Code Emitter (VIRCOE)**. VIRCOE abstracts away the complexity of coordinating computation across multiple banks and subarrays. By synchronizing code emission and managing virtual program counters, VIRCOE enables the compiler to exploit both bank-level and subarray-level parallelism without requiring manual programmer intervention. This design makes PUD architectures significantly easier to program compared to prior frameworks.

Beyond programmability, CHOPPER incorporates several optimizations to improve efficiency in bit-serial execution. These optimizations are grouped under the umbrella of *Optimizations for Bit-Sliced codes (OBS)*:

- **Bit-sliced scheduling:** dependent operations are aggregated to minimize redundant buffering of intermediate results.
- **Bit-sliced instruction selection:** constant rows (e.g., all-zeros or all-ones) embedded in DRAM are reused to eliminate unnecessary writes.
- **Instruction renaming:** the common “Store-Copy-Compute” pattern is shortened into a more efficient “Store-Compute” sequence by removing redundant copies of one-shot operands.

These techniques reduce both the total instruction count and intra-subarray data transfers, mitigating the primary bottlenecks in PUD execution.

Among existing frameworks, CHOPPER is the most closely related to our work. Both CHOPPER and our compiler aim to raise the programming abstraction of in-DRAM computing while systematically addressing the overhead of data movement. However, there are important distinctions. CHOPPER targets programmability by exposing a data-flow interface and relies on compiler-driven bit-slicing and VIRCOE scheduling to produce optimized PUD instructions. In contrast, our work integrates logic synthesis frameworks (e.g., `mockturtle`) to construct MIGs and employs an MLIR-based infrastructure to lower high-level arithmetic operations into in-DRAM instructions. This difference in approach highlights the complementary directions pursued: CHOPPER focuses on high-level programming usability, while our compiler emphasizes extensibility and general-purpose applicability across domains.

3.4 Discussion

The prior works on in-DRAM computing illustrate a clear trajectory. Ambit first established the feasibility of supporting simple bit-wise operations within DRAM. SIMDRAM elevated this model by introducing a programming abstraction based on micro-programs, enabling developers to write more expressive kernels while still relying on a fixed library of macros. MIMDRAM extended SIMDRAM with a fine-grained SIMD model, effectively introducing MIMD execution across subarrays and thereby improving flexibility and efficiency. CHOPPER sought to address programmability and efficiency simultaneously by building a compiler framework that translates high-level data-flow programs into in-DRAM instruction sequences. However, CHOPPER remains tightly coupled to the Usuba compiler and its DSL, limiting its generality and applicability beyond the DSL it supports.

In contrast, the compiler presented in this thesis is designed from the ground up for **extensibility** and **general-purpose applicability**. By building on the MLIR framework, our approach leverages its multi-level intermediate representation infrastructure to integrate cleanly with existing compilation flows. In particular, by selecting the `arith` dialect as the compiler front-end, we ensure that any high-level language or framework whose operations can be lowered to `arith` - such as C or PyTorch via torch-mlir - can be transparently supported. This design significantly broadens the reach of in-DRAM computing by decoupling it from custom DSLs or hand-written macros.

Equally important, the MLIR-based infrastructure allows our compiler to evolve as both hardware primitives and programming requirements advance. New dialects can be introduced to represent additional in-DRAM primitives or memory-instruction sets, and existing optimization passes can be reused or extended. Compared to prior frameworks, this provides a far more systematic and modular way to bridge high-level arithmetic operations with low-level DRAM execution. From the perspective of programmability, users no longer need to learn specialized DSLs or invoke architecture-specific macros; instead, they can rely on familiar programming environments, with the compiler responsible for lowering computations to DRAM instructions.

In summary, while SIMDRAM, MIMDRAM, and CHOPPER each contributed critical steps toward making in-DRAM computing programmable and efficient, our MLIR-based compiler advances the state-of-the-art by emphasizing extensibility, generality, and seamless integration with existing ecosystems. This positions in-DRAM computing not as a specialized niche tool, but as a broadly accessible backend for diverse application domains.

4 Design and Implementation of the Compiler

In this section, we present the design and implementation of the proposed compiler in detail. We describe the complete compilation flow, outlining how high-level logical and arithmetic operations are systematically translated into executable memory instructions for the DRAM controller.

4.1 Overview of the Compilation Flow

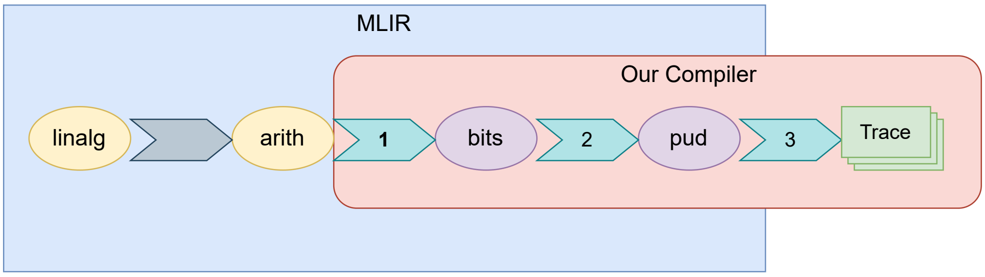

Figure 4.1: Overview of the Compilation Flow

Figure 4.1provides a concise overview of the complete compilation flow. The big blue box contains the MLIR components, ellipses representing dialects, namely `linalg` and `arith`, as well as the newly introduced `bits` and `pud` dialects in our work. Arrows between dialects denote the corresponding lowering or transformation passes. The components developed in our compiler are highlighted in the rounded pink box, including the aforementioned `bits` and `pud` dialects and the final trace file generated for execution on the simulator. In the following, we describe each dialect in detail and explain how it is lowered or transformed into other dialects or into trace files in the context of our framework.

On the left side of Figure 4.1, the two yellow ellipses denote the `linalg` and `arith` dialects. The `linalg` dialect provides structured tensor operations (e.g., matrix multiplication, convolution, and generic element-wise kernels), capturing loop nests and data access patterns at a high level. The `arith` dialect, while often introduced as the core for scalar integer and floating-point arithmetic, is type-polymorphic and also supports element-wise semantics over shaped types (vectors/tensors). To realize a general compilation flow that can interface with multiple front ends, our design explicitly starts from one-dimensional tensor operations in the `arith` dialect, including logical operations (AND, OR, XOR), integer addition, and floating-point multiplication. The gray arrow between `linalg` and `arith` summarizes a common lowering path in which structured, pointwise-amenable `linalg` ops are rewritten into `arith`'s element-wise forms, typically accompanied by compiler transformations such as tiling/partitioning, fusion, loop formation and distribution, unrolling, vectorization, bufferization, and subsequent canonicalization/cleanup to expose locality and parallelism. It should be noted that, when materializing loops, the lowering often introduces intermediate control-flow dialects such as `scf` or `affine` to explicitly represent loop nests and affine access functions; these are omitted from Figure 4.1 for clarity, as the figure focuses on the parts of the flow that directly concern arithmetic representations from which our design proceeds.

In our compiler, we begin with one-dimensional tensor operations in the `arith` dialect. (1) We first apply a bit-slicing transformation that materializes the `bits` dialect's **slice** data type and emits the corresponding bits operations (blue arrow 1). (2) We then lower these high-level, slice-typed operations to the `pud` dialect, whose operations encode DRAM-resident instructions (blue arrow 2). This stage is more involved: it performs technology mapping and scheduling for in-DRAM execution, including logic synthesis of the bit-serial datapath, address assignment/layout of operands in DRAM banks/subarrays/rows, carry generation and propagation for integer addition, and related micro-architectural constraints; the details are presented in the subsequent sections. (3) Finally, we translate `pud` programs into trace files consumable by the **NVMain** [PX12] simulator (blue arrow 3) to obtain runtime and energy estimates, and in this step we compare the simulated in-DRAM results against a CPU baseline to assess both performance/energy and functional equivalence.

In the following sections, we illustrate each component and pass of the compilation flow using, as a minimal running example, an addition of three large one-dimensional tensors of integer elements:

```
module {
  func.func @main(%arg0: tensor<8192xi32>, %arg1: tensor<8192xi32>,
      %arg2: tensor<8192xi32>) -> tensor<8192xi32>
  {
    %sum0 = arith.addi %arg0, %arg1 : tensor<8192xi32>
    %sum1 = arith.addi %sum0, %arg2 : tensor<8192xi32>
    return %sum1 : tensor<8192xi32>
  }
}
```

Listing 1. Addition of Three Integer Vectors in `arith` Dialect

4.2 Bits Dialect

The `bits` dialect follows the `arith` dialect, and its name derives from bit-serial and bit-slice - the former reflecting the computation paradigm adopted in this work, and the latter denoting the unique data type defined in the `bits` dialect. During the lowering from `arith` to `bits`, arithmetic operations such as `arith.addi` in the example are replaced by their counterparts in the `bits` dialect (e.g., `bits.addi`), while the data type is transformed from a one-dimensional tensor into a slice. The details of this data type and the transformation process will be elaborated in the following subsections. This design reshapes arithmetic operations into a form that better aligns with the structural organization of DRAM, thereby preparing them for subsequent computation in which DRAM rows serve as the fundamental unit of execution.

4.2.1 Slice Type

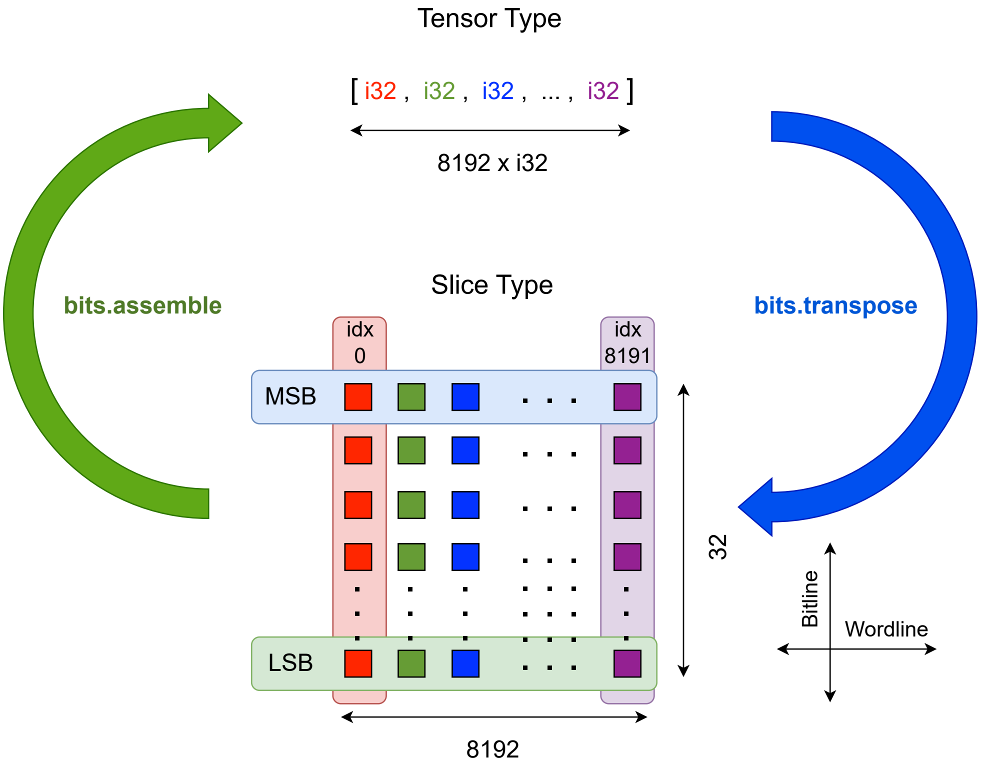

Figure 4.2: Transposed Slice Type and the Original Tensor Type

Figure 4.2 illustrates the tensor type used in our running example, which has a length of 8192 with elements of $32-bit$ integers. As introduced in Section 2.2.4, data stored in DRAM subarrays follows a vertical layout in our PUD scenario. Accordingly, the `bits.transpose` operation corresponds to the transposition process performed on the CPU, which converts data from a horizontal layout into a vertical one. In other words, we assume that data remains in a horizontal layout and has not yet been stored in DRAM prior to entering the compilation flow. During the lowering pass from the `arith` dialect to the `bits` dialect, this type is transformed by the `bits.transpose` operation into a bit-sliced layout of 32 rows and 8192 columns, where each bit corresponds to a single cell within a DRAM subarray. Viewed horizontally, the $n-th$ row corresponds to the $n-th$ bit of every $i32$ element in the original tensor, with n ranging from 0 to 31 in **Most Significant Bit (MSB)** to **LSB** order. In other words, row 0 contains the MSBs of all elements, whereas row 31 contains the LSBs. In the physical DRAM organization, each such row naturally maps to a wordline within a subarray, while each column corresponds to a bitline. Viewed vertically, the $m-th$ column corresponds to the $m-th$ $i32$ element of the original tensor. This vertical data layout forms the basis of the computation paradigm adopted in this work, enabling large-scale bit-wise SIMD-style execution within DRAM.

In Figure 4.2, the two arrows on the left and right represent the `bits.assemble` and `bits.transpose` operations in the `bits` dialect. The blue arrow denotes the `transpose` operation, which converts tensor-typed operands into the slice type before computation, whereas `assemble` performs the reverse transformation, converting the results from slice back into the original tensor type. These two operations conceptually correspond to the data type conversions that occur in practice when the CPU prepares operands for in-DRAM execution and collects the results afterward.

4.2.2 Operations

As shown in Table 4.1, the operations defined in the `bits` dialect can be categorized into two groups: computational operations and structural operations.

Category	**Operation**	**MLIR op**	**Operand → Result**
Computational	Element-wise AND	`bits.and`	slice, slice → slice
	Element-wise OR	`bits.or`	slice, slice → slice
	Element-wise XOR	`bits.xor`	slice, slice → slice
	AND Reduce	`bits.and_red`	slice → slice
	OR Reduce	`bits.or_red`	slice → slice
	XOR Reduce	`bits.xor_red`	slice → slice
	Integer Maximum	`bits.max`	slice, slice → slice
	Integer Minimum	`bits.min`	slice, slice → slice
	Integer Add	`bits.addi`	slice, slice → slice
	FP Multiply	`bits.mulf`	slice, slice → slice
Structural	Transpose	`bits.transpose`	tensor → slice
	Assemble	`bits.assemble`	slice → tensor
	Split	`bits.split`	slice → slice, slice
	Merge	`bits.merge`	slice, slice → slice

Table 4.1: Classification of Operations in the `bits` Dialect

Computational operations are organized into logical and arithmetic categories, and include both binary and unary operators. From the perspective of the `slice` type, binary operations consume two operands of identical shape and produce a result `slice` of the same shape; unary reductions consume a single `slice`. The distinction between logical and arithmetic becomes apparent at the row level: logical execution remains purely bit-wise, whereas arithmetic execution introduces inter-row dependencies. In particular, addition and multiplication (the latter approximated via repeated additions) require carry-in and carry-out signals; ensuring correct carry propagation is therefore essential for arithmetic within DRAM and will be discussed in the following sections.

The supported binary logical operators are `bits.and`, `bits.or`, and `bits.xor`, corresponding to `arith.andi`, `arith.ori`, and `arith.xori`. The binary arithmetic operators include integer addition (`bits.addi`) and floating-point multiplication (`bits.mulf`). We also provide binary comparison-based operators, `bits.maximum` and `bits.minimum`, which produce a `slice` whose each column is, respectively, the larger or smaller of the two input columns that encode one-dimensional unsigned-integer tensors; these correspond to `arith.maxui` and `arith.minui`.

Finally, three unary reduction operators - `bits.and_red`, `bits.or_red`, and `bits.xor_red` - perform per-column bit-wise reductions over the rows of the input `slice`. Their output preserves the input's column count but has exactly one row. These reductions have no direct counterparts in the `arith` dialect; their semantics align with reduction operations defined in the `linalg` dialect.

In addition to the `transpose` and `assemble` operations discussed in the previous section, the dialect also defines two structural operations: `split` and `merge`. These operations enable adaptation to different DRAM configurations as well as more flexible parallelization strategies. In our example, each slice has 8192 columns, matching the configuration of mainstream DDR4 DRAM (8 devices, each with 1024 byte columns). In this setting, all 8192 bit columns (i.e. 1024 byte columns) of a DRAM device execute the same operation simultaneously, thereby realizing SIMD-style execution at this scale. In practice, however, each bank can accept and process different memory instructions, which makes bank-level parallelism possible. Moreover, cases may arise where the tensor length exceeds the maximum number of columns that can be accommodated in DRAM. In both situations, the `split` operation is necessary to partition operand slices along the vertical dimension. Naturally, the `bits` dialect also provides the corresponding `merge` operation to combine the resulting slices after computation.

Listing 2 illustrates how the addition operation originally expressed in Listing 1 is lowered into `bits` dialect. It is worth noting that the `transpose` operation is applied only to the tensors that serve as input parameters to a `func` operation in the `func` dialect, and it is performed on demand. Similarly, the `assemble` operation is applied exclusively to the slice values that correspond to the return values of a `func` operation. In other words, if a function contains multiple addition operations internally, all intermediate results remain in the slice form throughout the computation.

```
module {
  func.func @main(%arg0: tensor<8192xi32>, %arg1: tensor<8192xi32>,
      %arg2: tensor<8192xi32>) -> tensor<8192xi32>
  {
    %0 = bits.transpose %arg0 : tensor<8192xi32> -> !bits.slice<32x8192>
    %1 = bits.transpose %arg1 : tensor<8192xi32> -> !bits.slice<32x8192>
    %2 = bits.addi %0, %1 : !bits.slice<32x8192>
    %3 = bits.transpose %arg2 : tensor<8192xi32> -> !bits.slice<32x8192>
    %4 = bits.addi %2, %3 : !bits.slice<32x8192>
    %5 = bits.assemble %4 : !bits.slice<32x8192> -> tensor<8192xi32>
    return %5 : tensor<8192xi32>
  }
}
```

Listing 2. Addition of Three Integer Vectors in `bits` Dialect

4.3 PuD Dialect

According to the design of our compilation flow shown in Figure 4.1, the `bits` dialect is subsequently lowered into the `pud` dialect, whose name is derived from **Processing using DRAM**. While the `bits` dialect targets the slice type that aligns with DRAM structures, it still represents arithmetic operations at a relatively high level. The `pud` dialect goes one step further by exposing primitives that correspond to actual in-DRAM computation. In other words, the data types and operations of this dialect must be understood as "memory-controller-like instructions". The details of this dialect will be elaborated in the following two subsections.

4.3.1 Row Type

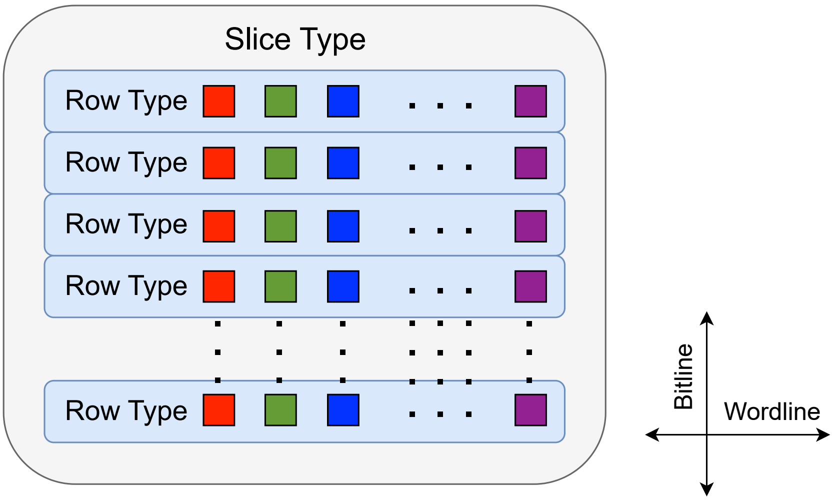

Figure 4.3: `bits.slice` Type and `pud.row` Type

The row type is the sole data type in the `pud` dialect. Logically, it represents a single row of the slice type in the bits dialect, as illustrated in Figure 4.3. Unlike the slice type, which carries explicit dimensional information in terms of rows and columns, the row type does not encode any length (i.e., number of columns). Instead, it can be regarded as a logical row in the DRAM structure, characterized only by its functional attribute - namely, whether it belongs to group `B`, `C`, or `D` of the computation-capable subarray described in the Chapter 2. The specific row being represented is determined by the `pud.get_row` operation. This design makes the row type highly flexible: it may represent a row in a specific subarray, a row replicated across all subarrays within a bank, or even a row spanning all subarrays in the DRAM device. Such flexibility enables the exploration of parallelism at different levels in subsequent stages. In this way, the row type bridges the abstraction of bit slices with the granularity of DRAM execution units.

4.3.2 Operations

The `pud` dialect, which implements the in-DRAM computation model proposed in [Ses+17], defines the following five operations:

- `pud.ap`: corresponds to the `AP` instruction, which activates one logical address, i.e., one row-typed object. In practice, this is often used to activate one address between $B12$ and $B15$, thereby triggering a TRA. This mechanism realizes the functionality of a majority logic gate.
- `pud.aap`: corresponds to the `AAP` instruction, which simultaneously activates two logical addresses, i.e., two row-typed objects. This mechanism enables row-cloning the data from the first address into the second.
- `pud.get_row`: is used to create and return a variable of type row. It accepts up to five integer parameters - $channelID$, $rankID$, $bankID$, $subarrayID$, and $rowID$. In other words, the row type itself does not embed any address information; instead, the concrete address of a row object is determined during IR construction by the `get_row` operation that instantiates it. This separation of data type and addressing improves flexibility and allows the same IR to be re-targeted across different DRAM configurations.
- `pud.store`: can be regarded as the interface to the `bits` dialect. It takes a slice object as input and returns a row object corresponding to the first row of the slice, i.e., the MSB row. The concrete physical address of this row is determined during the address allocation stage of the lowering pass from `bits` to `pud`, which will be described in detail in the next section. Since a slice type carries dimensional information, obtaining the first row implicitly defines the entire slice: the remaining rows can be created via the `get_row` operation by applying offsets relative to the address of the first row.
- `pud.load`: serves as the inverse of `store`. It takes as input a row object, which represents the first row (i.e., the MSB row) of a slice, together with an integer parameter specifying the number of rows. Based on this information, it reconstructs the corresponding slice object. In other words, given the starting row address and the row count, `load` reassembles the bit-sliced layout by implicitly retrieving subsequent rows through relative offsets, thereby restoring the full slice structure from its row-level representation.

Together with the row type, these constructs bridge bit-sliced representations with DRAM-level primitives, preparing the ground for the lowering process described in the next section.

4.4 From Bits to PuD

This section provides a detailed explanation of the lowering pass from the `bits` dialect to the `pud` dialect, covering logic synthesis, address allocation, and the handling of different logical and arithmetic computations.

4.4.1 Logic Synthesis

As discussed in Section 2.3, the Lime framework is capable of generating pseudocode of memory instructions - namely, sequences of `AP`/`AAP` instructions - for logic networks built with majority gates and inverters using the mockturtle framework. Accordingly, our task of this step is to convert all arithmetic operations in the `bits` dialect into such logic networks that can be consumed by Lime. This conversion forms the core of the logic synthesis step in this lowering pass.

Before constructing the logic network, one important point must be clarified: given the time and memory costs of compilation, it is impractical to connect every single bit of each operand slice into the network. Our construction method therefore proceeds as follows. (1) Since the `pud` dialect uses the row type as its fundamental operand, which is also the key to realizing SIMD-like execution, we do not need to create a separate network for each column; instead, we treat all columns as a single collective entity. (2) Nevertheless, directly connecting all bits of an entire column into the network would still be prohibitively expensive. For example, in our running case, performing only two 32-bit additions with proper carry handling already requires 64 single-bit full adders, each of which is itself composed of multiple majority gates and inverters. This would make compilation far too costly. To address this, we propose to treat all rows as a collective entity as well. Once each slice has been assigned DRAM addresses (i.e., once the address of each row is known), carry propagation can then be handled appropriately. In summary, this means that our compiler constructs only a single-bit full adder per addition, which significantly reduces the time and memory requirements of logic synthesis and thereby improves the performance of the compiler itself. This abstraction is essential for scaling the compiler to large tensor workloads.

The mockturtle Application Program Interface (API) provides not only the low-level functions `create_maj()` and `create_not()` for constructing the two primitive gates of an MIG network, but also higher-level functions such as `create_and()`, `create_or()`, and `create_xor()` to build the logic gates required by the logical operations in the `bits` dialect. At the highest abstraction level, it further offers a `full_adder()` function for creating an one-bit full adder, which itself is implemented by invoking the aforementioned intermediate functions like `create_xor()`. To improve the performance of our compiler, we choose to bypass these intermediate abstractions and directly construct an one-bit full adder using only the two primitive gates, i.e., three majority gates and two inverters, as illustrated in Figure 4.4.

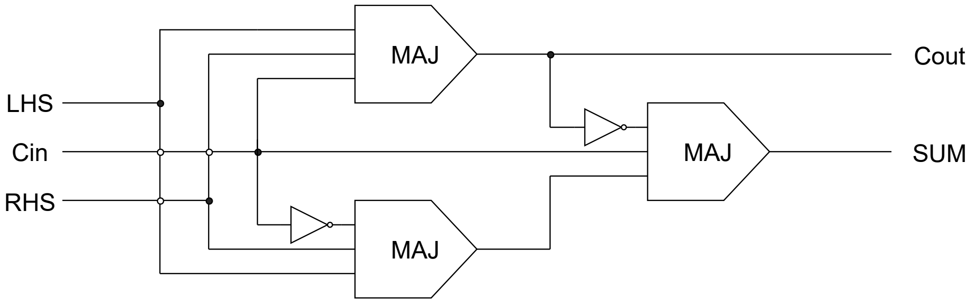

Figure 4.4: Single-bit Full Adder in Majority Gates and Inverters

Figure 4.5 shows the logic network generated by our compiler for the running example. The five triangular nodes at the bottom (labeled `1-5`) represent the network inputs: nodes `1-3` correspond to the three tensor inputs of the `func.func` operation (each lowered to a slice operand, i.e., `%0`, `%1` and `%3` in Listing 2), while nodes `4` and `5` correspond to the carry-in signals of addition 1 and addition 2, respectively. The square node labeled `0` represents the constant signal 0, which does not participate in the computation in this network. The six elliptical nodes labeled `6-11` in the middle denote the six majority gates required to construct the two one-bit

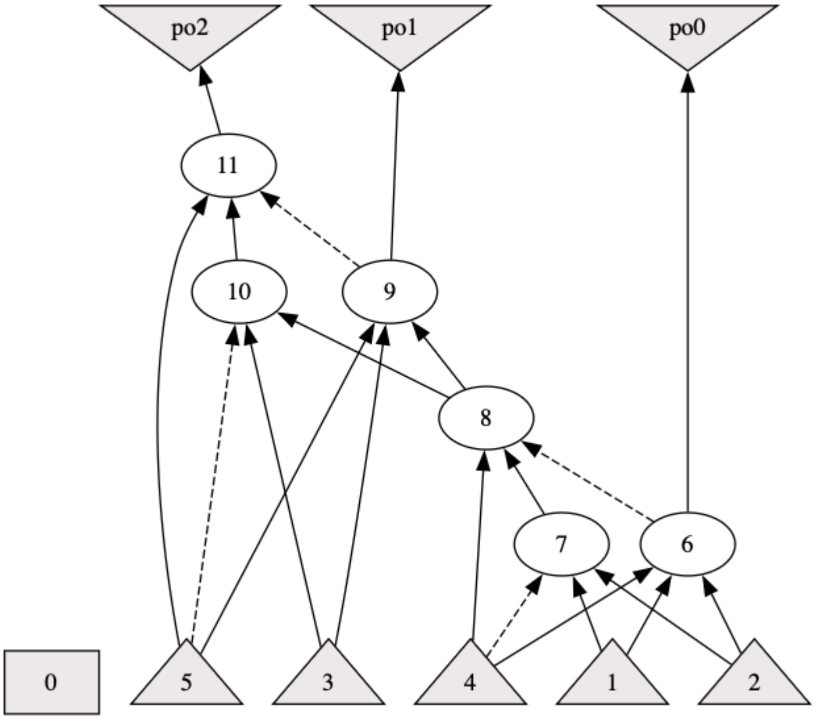

Figure 4.5: Logic Network of the Running Example, Visualized with `Graphviz` [GN00]

full adders, while the four dashed arrows indicate signals inverted by inverters. The output of node `8` represents the result of addition 1, i.e., `%2 in Listing 2`, which is directly used as an operand in the computation of addition 2. At the top of the figure, the three inverted triangular nodes represent the network outputs: `po0` and `po1` denote the carry-out signals of addition 1 and addition 2, respectively, and `po2` corresponds to the slice object that is returned as the final result of the `func.func` operation (i.e, `%4` in Listing 2). Both the carry-in and carry-out signals of the two additions are explicitly connected to the inputs or outputs of the logic network to prevent `mockturtle` from discarding them as unused nodes; this step is essential for ensuring correct carry propagation in the next stage.

Listing 3 presents the pseudocode of `AP` and `AAP` instructions generated by the `Lime` framework from the logic network shown in Figure 4.5. Unlike the example illustrated in Section 2.3, our compiler normalizes the addresses of group `Lime` into a logical address form, which facilitates the subsequent address allocation step.

4.4.2 Address Allocation

Address allocation - also referred to as **data mapping** in some prior work - denotes the process of assigning DRAM addresses to the inputs, intermediate values, and final outputs involved in computation. In our compiler, we currently adopt an First-Come, First-Serve (FCFS) policy with contiguous address allocation. Using the running example, we now illustrate the concrete process of address allocation in this step.

```
AAP I0 B11
AAP I1 B1
AAP I3 B7
AAP I3 B2
AAP B12 O0
AAP I1 B0
AP B15
AAP B1 B4
AAP B5 B1
AAP I3 B2
AP B12
AAP I4 B5
AAP I4 B1
AAP I2 B2
AAP B0 S1
AAP B12 O1
AAP I2 B1
AAP S1 B2
AP B14
AAP B0 S3
AAP I4 B0
AAP S3 B4
AAP B5 B2
AAP B12 O2
```

Listing 3. Pseudocode of `AP` and `AAP` Instructions for our Example

In the instruction sequence shown in Listing 3, the operands that require address allocation fall into three categories: `In`, `On`, and `Sn`. The first two denote the inputs and outputs of the computation, respectively, while the last represents intermediate values that must be stored during execution. It is important to note that these intermediate values are not the same as the intermediate results such as `%2` in Listing 2; rather, they refer to rows in group `B` that may need to be temporarily buffered when a refresh occurs during computation. In our example, each slice object contains 8192 rows, while the number of rows requiring allocation is far fewer than 1006, which is the number of available rows in group `D` of a single subarray. Therefore, address allocation can be confined to a single subarray, while enabling a SIMD execution scale of up to 8192 during actual computation.

We traverse the sequence in Listing 3 to perform address allocation. Among all inputs, `I0`–`I2` correspond to the three operand inputs, each of which requires 32 rows. `I3` and `I4` represent the carry-in signals of the two additions, each of which requires only one row. Similarly, `O0` and `O1`, as the carry-out signals of the two additions, are each assigned one row, while `O2`, as the final output, requires 32 rows. The remaining intermediate signals, `S1` and `S3`, are each allocated one row. In summary, the first-row address index and the number of rows allocated for each signal in our example are listed in the following table, arranged in the order of FCFS allocation from top to bottom.

Signal	First Row Index	Number of Rows
I0	0	32
I1	32	32
I3	64	1
O0	65	1
I4	66	1
I2	67	32
S1	99	1
O1	100	1
S3	101	1
O2	102	32

Table 4.2: Assigned Address of each Signal

4.4.3 Computation Process

This section will detail, step by step, how after address allocation the pseudocode in Listing 3 is transformed into `pud`-level IR that incorporates the correct address information and fully preserves the computation process.

Row Iteration

As discussed in Section 4.4.1, we treat all rows as a single unit during logic synthesis in order to improve compiler performance. Consequently, at this stage of execution, the pseudocode shown in Listing 3 must be iterated over the number-of-rows times (32 in our example). In each iteration, it is necessary to compute the correct row offsets for both operand slices and result slices, i.e., their correct addresses. Specifically, in the n-th iteration, the row index of signals `I0`, `I1`, `I2` (for reading) and `O2` (for writing) is calculated as:

$$\text{RowIndex}_{\text{op}} = \text{FirstRowIndex} + (\text{NumRows} - n - 1) \quad 0 \le n < \text{NumRows} \tag{4.1}$$

The offset $(\text{NumRows} - n - 1)$ ensures that the computation proceeds from the LSB row toward the MSB row. While this direction does not affect logical computation, it is crucial for ensuring correctness in arithmetic computation involving carry propagation. For other signals that are allocated only one single row, the same row address is used in every iteration. For example, at the $8th$ iteration (with $n = 8$, counting from zero), the row indices of the signals are as shown in Table 4.3:

Element-wise AND, OR and XOR

Before moving on to the carry propagation of our running example, it is necessary to first discuss the computation process of the three element-wise logical operations: AND, OR, and XOR. Since these operations do not involve carry propagation across bit positions, their handling is comparatively straightforward.

Signal	Row Index
I0	23
I1	55
I3	64
O0	65
I4	66
I2	90
S1	99
O1	100
S3	101
O2	125

Table 4.3: Operating Address of each Signal at the $8th$ Iteration

Once the method for determining the correct row address in each iteration is understood, the logical operations become fairly straightforward. The single-row signals `I3`, `I4`, `O0`, and `O1`, which were created and allocated in the previous section for proper carry handling, are no longer needed in logical operations. Each iteration simply reads the i-th row of every operand slice, processes the data through group `B`, and then writes the result back to the i-th row of the output slice.

AND, OR and XOR Reduce

Before logic synthesis, the logical networks generated for the three reduction operations are identical to those of their corresponding element-wise logical operations; the difference lies in how the row iteration is performed. For element-wise operations, in each iteration, the two operands are taken from the same row index of the two input slices, and the resulting value is written to the result slice at the same row index. Consequently, the number of iterations equals the number of rows in the slice.

In contrast, for reduction operations, the two operands in each iteration are the current reduction result and the next row of the single input slice. During the first iteration, the operands are the first and second rows of the input slice, and the result is temporarily stored in a specific address within the B group as one of the operands for the next iteration. Starting from the second iteration, in the n-th iteration, the compiler reads the $(n+1)$-th row of the input slice and performs the corresponding logical operation with the result produced in the previous iteration. In other words, a reduction operation consists of a number of iterations equal to the total number of rows minus one. Only the result of the final iteration is written back to the D group address as the output slice. As a result, a reduction operation produces a `slice` with a single row and the same number of columns as the input.

Max and Min

The maximum and minimum operations are slightly more complex than the aforementioned logical operations, as each involves two logic networks and two rounds of row iteration.

Consistent with conventional logic synthesis, both functions consist of a comparator and a 2-to-1 selector.

Through analysis of their truth tables, we find that an n-bit comparator can be realized simply by iterating a single MAJ gate n times. Specifically, the comparator is implemented using one AND-like MAJ gate, whose three inputs are a constant 0, and two variable inputs: I_0 and the inverted I_1. This configuration allows the gate to determine whether I_0 is greater than I_1. During iteration, I_0 and I_1 are provided in order from the LSB row to the MSB row. Starting from the second iteration, the constant-0 input of the MAJ gate is replaced by the output of the previous iteration, thereby ensuring that the comparison proceeds from lower to higher bits, with the high-bit comparison result overriding the lower-bit results. The result of the final iteration is written back to a D-group address as a mask row, which will be used in the subsequent selection stage.

The construction of the 2-to-1 selector, though involving more MAJ gates and inverters, is more intuitive in its structure and iteration. We use two AND functions and one or function to sequentially combine each row of the two inputs, I_0 and I_1, with the mask row, and the results of each iteration are written into the corresponding rows of the output slice.

Integer Addition

We now turn to our running example, namely the addition of one-dimensional integer tensors. The overall logic remains the same: in each iteration, one row of the operands is read, processed, and then written back to the corresponding row of the result. The key difference lies in carry handling, which will be elaborated in detail below.

The first point to clarify in our example is that carry handling only needs to be considered within each individual addition, namely the propagation of carries across rows. There is no carry dependency between the two additions, which explains why during the logic synthesis step the carry-out of addition 1 was not connected to the carry-in of addition 2.

When the instruction sequence in Listing 3 is iterated, in each iteration the carry-out produced by the each addition is written directly into the corresponding `O0` or O1 row. In other words, for carry-out values, no special treatment beyond the pseudocode is required. The handling of carry-in, however, involves two cases: (1) in the very first iteration, the carry-in row should be zero. Therefore, when reading `I3` or `I4`, they are substituted with the all-zero row from group `C`; (2) starting from the second iteration, the carry-out generated in the previous iteration (i.e., the previous row addition) must be taken into account. Concretely, before each iteration begins, a row-clone operation is executed to copy the carry-out of each addition into its corresponding carry-in row. This ensures that the carry-in rows are updated in time, before the carry-out values from the previous iteration are overwritten and lost.

To give an intuitive summary, the 8-th iteration of the running example should execute the instructions shown in Listing 4, where the actual row indices of the signals located in group `D` are annotated as subscripts.

AAP $O0_{65}$ $I3_{64}$
AAP $O1_{100}$ $I4_{66}$
AAP $I0_{23}$ B11
AAP $I1_{55}$ B1
AAP $I3_{64}$ B7
AAP $I3_{64}$ B2
AAP B12 $O0_{65}$
AAP $I1_{55}$ B0
AP B15
AAP B1 B4
AAP B5 B1
AAP $I3_{64}$ B2
AP B12
AAP $I4_{66}$ B5
AAP $I4_{66}$ B1
AAP $I2_{90}$ B2
AAP B0 $S1_{99}$
AAP B12 $O1_{100}$
AAP $I2_{90}$ B1
AAP $S1_{99}$ B2
AP B14
AAP B0 $S3_{101}$
AAP $I4_{66}$ B0
AAP $S3_{101}$ B4
AAP B5 B2
AAP B12 $O2_{125}$

Listing 4. Instructions Including Row Indices of 8-th Iteration for our Example

Floating-point Multiplication

FP multiplication in a bit-serial computation model presents significant challenges. Whereas integer addition only requires handling carry propagation across bits, FP multiplication involves mantissa multiplication, exponent addition, normalization, and rounding. A naive bit-serial implementation would therefore result in prohibitively deep logic networks and long latency. To address this, approximate methods are often employed, for instance by simplifying mantissa multiplication through shift-and-add approximations or by reducing the precision of intermediate values. Similar ideas have been validated in low-precision training of neural networks, where carefully controlled approximations substantially reduce memory, bandwidth, and energy costs while maintaining competitive accuracy [Gup+15] [Hao+25] [Zam+21].

Empirical results further suggest that for large-scale machine learning workloads such as LLM training, small inaccuracies caused by approximate FP multiplication are generally tolerable, as the optimization process can absorb them. Nevertheless, there is a trade-off: lowering precision too aggressively may lead to convergence issues or degraded generalization. Hence, approximate floating-point multiplication represents a practical compromise in bit-serial models, enabling feasible latency, area, and energy efficiency while retaining acceptable training accuracy.

Therefore, in this work we approximate FP multiplication by means of integer addition. Specifically, we adapt the approximate FP multiplication model proposed in [Guo+25b]

- originally designed for computing using Resistive Random-Access Memory (RRAM) and 3D-SRAM - to the context of DRAM-based bit-serial computation, and provide a detailed analysis of its accuracy in Section 5.1.2. The method proposed in [Guo+25b] is as follows:

Given two FP numbers

$$FP_A = (1 + M_A) \times 2^{E_A}, \qquad FP_B = (1 + M_B) \times 2^{E_B}, \tag{4.2}$$

where M_A and M_B denote the fractional mantissas and E_A, E_B are the exponents, the exact product can be expressed as

$$FP_A \times FP_B = (1 + M_A)(1 + M_B) \times 2^{E_A + E_B}. \tag{4.3}$$

Instead of computing the mantissa multiplication $(1 + M_A)(1 + M_B)$ directly, the method introduces a logarithmic approximation:

$$\log_2(1 + M_{AB}) = \log_2(1 + M_A) + \log_2(1 + M_B) \approx M_A + M_B, \tag{4.4}$$

which effectively reduces the multiplication of mantissas to an integer addition operation.

We implement this approximation algorithm in the context of bit-serial in-DRAM computing as follows:

Taking IEEE 754 [19] half-precision FP16 as an example - which is widely used in machine learning algorithms - it consists of 16 bits in total, including 1 sign bit, 5 exponent bits, and 10 fraction (mantissa) bits. Our interpretation of a slice representing FP16 is illustrated in Figure 4.6.

When handling the multiplication of two FP slices, our logic synthesis proceeds as follows. First, we generate an **XOR** logic network to compute the MSB row of the product slice, i.e., the sign bit. Second, we treat the remaining exponent and mantissa rows as a single slice and process them using integer addition. This design automatically accounts for the case where the sum of mantissas exceeds one, i.e., when a carry-out is produced from the highest mantissa bit, causing the exponent to increment by one. However, it should be noted that in such cases the mantissa of the product must also be right-shifted by one to implement the implicit division by two. For our scenario, this operation is prohibitively expensive: at runtime, one would need to identify all columns whose mantissa MSB generates a carry-out, extract them into a new slice, and then perform the shift operation. In a bit-serial DRAM computation model, shifting is particularly costly. For instance, in FP16, right-shifting the mantissa by one requires 9 row-clone operations. Given this high overhead, our implementation does not include this step. In other words, in such cases the exponent part of the product is correct, but the mantissa part is inaccurate. We analyze the accuracy of our FP multiplication implementation under these conditions in Section 5.1.2. Finally, we generate a logic network to subtract the extra bias from the exponent, which corresponds to an integer addition with the bitwidth of the exponent.

To summarize, the FP16 multiplication at the `pud`-dialect level is carried out in three steps:

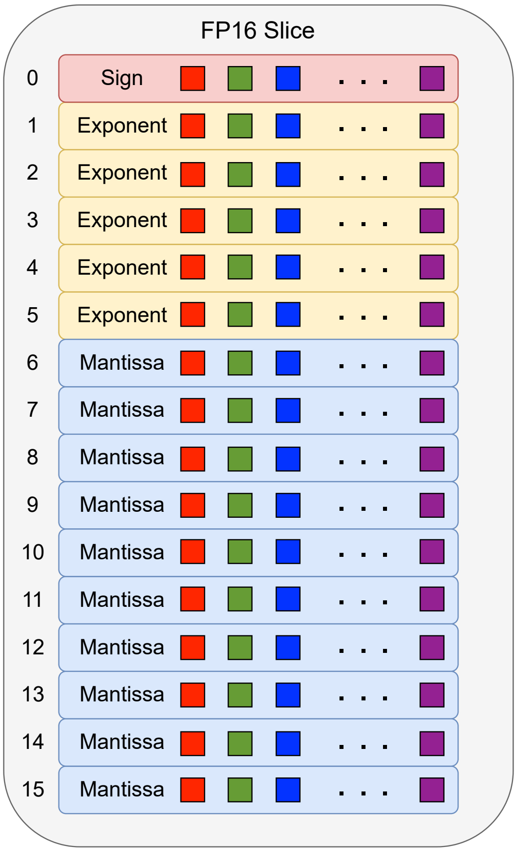

Figure 4.6: FP16 Slice Representation

1. **Sign bit computation**: The instruction sequence corresponding to the logic network computing the sign row is iterated once, with the result written into the MSB row of the product slice.
2. **Exponent and mantissa addition**: The instruction sequence for adding the exponent and mantissa parts is iterated 15 times, with results written into rows 1–15 of the product slice.
3. **Bias correction**: The instruction sequence for bias correction is executed 5 times, each time reading one row from the exponent part of the product slice and the corresponding row from a slice representing the negative bias, performing addition, and writing the result back into the exponent rows of the product slice.

4.5 Operand Tracking

In addition to the compilation flow shown in Figure 4.1, we implement an **Operand Tracker** to monitor the runtime state of each cell in groups `B` and `D` of DRAM subarrays. This feature enables more flexible and convenient debugging and tracing. The state of each cell is represented by different kinds of **expressions**.

- **Base expressions**:
    - `ConstExpr`: represents the state of a cell in group `C`. Its value is fixed to either 0 or 1 and cannot be modified at runtime.
    - `DataExpr`: represents the initialized data of a cell in group `D`. Its value can be configured to 0 or 1 using the `configure()` method.
- **Derived expressions**:
    - `NotExpr`: denotes the negation of another expression.
    - `MajExpr`: denotes the majority operation applied to other three expressions.

Whenever a concrete memory instruction in the `pud` dialect is executed, the **Operand Tracker** performs the corresponding operation (e.g., cloning, negation, majority) on the expressions associated with the target addresses. All four kinds of expressions implement the `evaluate()` method. Once the `DataExpr` values in group `D` are configured with their initial binary values, the actual binary value of any cell in groups `B` or `D` can be recursively determined at runtime by invoking `evaluate()`.

This mechanism allows us to trace the behavior of our in-DRAM computing system in detail, and it also enables verification of computational correctness when benchmarking with a simulator.

5 Evaluation

This chapter first verifies the functional correctness of our in-DRAM computing model. It then evaluates the design and implementation of our work in terms of performance and power consumption, by benchmarking against prior similar approaches.

5.1 Computation Correctness Verification

5.1.1 Verification Method

We verify the functional correctness of the compilation flow described in the previous chapter as follows. The input is an `arith`-dialect-level `func.func` function that specifies a one-dimensional tensor computation. Based on the dimensions and bitwidths of the input tensors, random operands are generated. On one hand, we directly evaluate the computation step by step using `C++` arithmetic functions according to the operations defined in the `func.func`, and the result is taken as the CPU reference. On the other hand, the same `arith`-level function is successively processed by two lowering passes, `ArithToBits` and `BitsToPuD`. The resulting `pud`-dialect-level instruction sequence is then traversed: when encountering a `pud.store`, the generated random numbers are configured into the corresponding DRAM cells as `DataExpr` values by mapping them to the rows and columns of the slice; each subsequent `pud.ap` and `pud.aap` operations is handled in order by the `Operand Tracker`; finally, when a `pud.load` is reached, the `evaluate()` method is invoked on the expressions of all cells at the corresponding addresses, and the results are converted back into the original one-dimensional tensor format as the DRAM-computation output. Comparing this result with the CPU reference output allows us to validate the correctness of the computation.

Through the verification procedure described above, we confirm that the memory instructions generated by our compiler perform logical operations and integer addition correctly, demonstrating that the design and implementation presented in Chapter 4 are functionally sound. In the following section, we evaluate and analyze the accuracy of our implemented approximation method for FP multiplication.

5.1.2 Accuracy of Floating-point Multiplication Approximation

As discussed in Section 4.4.3, the approximate FP multiplication implemented in our bit-serial in-DRAM computation model is not fully complete, as it cannot properly handle the case where the sum of the mantissas exceeds one and thus requires a right shift of the mantissa. We refer to this as case **B**. After further verification, we identified another overlooked situation: when the sum of the exponents of the multiplicands is smaller than the bias to be subtracted. If the absolute difference is **n**, the exponent should be set to zero, and the mantissa should be right-shifted by **n** positions. We refer to this as case **C**. Similar to case **B**, this requires runtime per-column checks, splitting slices column by column, and performing shifts - operations that are prohibitively expensive in in-DRAM computation.

Finally, we consider a DRAM-computed result to be an acceptable approximation if it falls within **±10%** of the corresponding CPU result; we denote such cases as case **A**.

We evaluate three floating-point formats: `FP8` (E4M3), **IEEE 754 [19]** `FP16`, and `FP32`. For each type, we conduct ten one-dimensional tensor multiplications with a length of 8192, resulting in a total of $81,920$ samples per type. The accuracy results for the three formats are summarized in Table 5.1.

Data Format	Accuracy (Case A)
FP8 (E4M3)	$\leq 10\%$
FP16 (E5M10)	81.57%
FP32 (E8M23)	85.93%

Table 5.1: Accuracy of Approximate FP Multiplication

The results shown in Table 5.1 indicate that accuracy improves with increasing bitwidth. The underlying reason is that as the number of mantissa bits grows, their proportion relative to the total number of bits increases, thereby reducing the probability that randomly generated test values fall into case B. This also explains why the `FP8` format achieves an accuracy of less than 10%, while the probability of case **B** occurring exceeds 80%, further supporting this analysis.

5.2 Experimental Setup

5.2.1 Baseline Selection

In terms of relevance, Chopper [PWY23] is the work most closely related to our compiler. Like ours, their system implements a compiler targeting bit-serial in-DRAM computation, and therefore it would naturally serve as a baseline for comparison in the evaluation of this thesis. Unfortunately, their compiler itself or its implementation has not been open-sourced, nor does the Chopper paper provide sufficient details regarding the configurations of the workloads or algorithms used for benchmarking, such as **Deep Neural Networks (DNNs)** and **Differential Privacy**. For these reasons, we are unable to include Chopper as a point of comparison in this chapter.

In contrast, MIMDRAM [Oli+24] has open-sourced its programming framework for bit-serial, MIMD-style computation in DRAM. We therefore choose it as the baseline for evaluating our work. As mentioned earlier, MIMDRAM packages the `AP` and `AAP` instruction sequences required for operations such as addition, subtraction, maximum, and minimum into basic operators, which are then exposed as macros for invocation in C programs. Similarly, as described in the previous chapter, our compiler translates commonly used arithmetic operations in the MLIR `arith` dialect - particularly those relevant to machine learning - into `AP` and `AAP` instruction sequences. This similarity makes the comparison presented in this chapter possible. The following section details the experimental setup and benchmarks used in this evaluation.

5.2.2 Workloads Specification

For binary operations, we select addition, subtraction, maximum, minimum, and element-wise AND, OR, XOR, and for unary operations, we select bit-wise reductions (AND, OR, XOR) as the set of operators implemented by both MIMDRAM and our work for direct comparison. In the case of binary operations, both operands and the result are one-dimensional tensors of 16 32-bit integers. For unary operations, the three reduction operators take an one-dimensional tensor of 16 32-bit unsigned integers as input and produce a one-dimensional tensor of 1-bit unsigned integers of the same length as output. To clearly isolate and evaluate the basic operators, each operation is executed only once. In other words, MIMDRAM performs SIMD-like rather than MIMD-like computation, corresponding to the μ**Program** introduced in SIMDRAM [Haj+21]. For each benchmarked basic computation, we compare the numbers of `AP` and `AAP` instructions generated by the MIMDRAM programming framework and by our compiler, since under the same DRAM hardware model, executing the same instruction incurs identical latency and energy cost.

In addition to the workloads mentioned above, we also present results for operations implemented in our work but not in MIMDRAM. These include the approximate FP multiplication described in the previous chapter, where the one-dimensional tensor has IEEE 754 single-precision (32-bit) floating-point elements (1 sign bit, 8 exponent bits, and 23 stored fraction bits with an implicit leading 1).

5.2.3 NVMain-PIM Simulator

After obtaining the `AP` and `AAP` instruction sequences generated by MIMDRAM and by our compiler, we further simulate both using NVMain-PIM simulator under the same DRAM configuration, in order to compare their execution time and energy consumption.

NVMain [PX12] is an architectural-level simulator designed to model both conventional DRAM and emerging Non-Volatile Memory (NVM) technologies. It provides flexible configuration of architectural parameters and enables detailed analysis of performance and energy. With its validated modeling framework, NVMain serves as a reliable tool for simulating DRAM behavior.

NVMain-PIM [SU-25] is an extended version of the NVMain simulator that supports bit-serial in-DRAM computation. It introduces specific mechanisms for simulating `AP` operations via `T` (triple row activate) and for simulating `AAP` operations via `OA`, `ODRA` and `OTRA` (overlapped activate, overlapped double row activate and overlapped triple row activate), corresponding to Row-Clone and Row-Clone with TRA operations. To simulate the workloads described in the previous section, we generate trace files in the required format from the `AP` and `AAP` instruction sequences, with the detailed procedure described in the next section. It should be noted that NVMain-PIM does not currently support accurate simulation of inter-subarray Row-Clone operations. Therefore, in our evaluation, all operations of each workload are confined to a single subarray.

For our experiments, we configure NVMain-PIM to model a DRAM system based on Micron DDR3-1333 (4 Gb, ×8) devices. The configuration assumes a 64-bit bus with two channels and two ranks per channel, each rank containing eight banks. Each bank consists of 65,536 rows and 32 visible columns, without enabling subarray-level parallelism. The timing parameters follow JEDEC DDR3 specifications, with a 666 MHz clock (1333 MT/s DDR). Power and energy parameters are set according to Micron datasheet values, with an operating voltage of 1.5 V. The memory controller adopts the First-Ready, First-Come, First-Serve (FRFCFS) scheduling policy with an open-page row buffer management scheme. This configuration provides a realistic DRAM baseline for our simulation.

5.2.4 Trace Emitting

As shown in Figure 4.1, the final stage of our implemented compilation flow is the Trace Emitter, which is designed to generate trace files for simulation with NVMain-PIM. The Trace Emitter `pud-trace-emit` is implemented as a standalone tool, independent of `mlir-opt`, and serves to integrate all components developed in this thesis.

Specifically, it takes as input an MLIR file in which a computation function is described using the `arith` dialect. The function is successively lowered through the `ArithToBits` and `BitsToPuD` passes, producing an instruction sequence in the `pud` dialect. Next, the `pud` operations are traversed: when a `pud.store` operation is encountered, a random vector consistent with the slice's dimensional information is generated (using LLVM's `APInt` to produce arbitrary-width random numbers), and the corresponding Operand Tracker configures each cell's expression accordingly. When traversing `pud.ap` or `pud.aap` operations, their addresses are converted into global hexadecimal strings and written as trace lines following the simulator's format specification, while the Operand Tracker performs the corresponding operation to simulate functionality. Once all `pud.ap` and `pud.aap` operations have been traversed, reaching a `pud.load` operation marks the completion of the trace file. At this point, each cell's expression is evaluated into concrete Boolean values, which are then converted back into the one-dimensional tensor format according to the slice's shape and dimensions, yielding the DRAM-computation result.

For comparison, the CPU reference result is computed by applying the operations described in the `arith` dialect function directly to the generated random inputs, using C++ operators overloaded for `APInt` or `APFloat`.

In summary, the Trace Emitter takes an `arith`-dialect program as input, lowers it into the `pud` dialect, and produces a trace file for NVMain-PIM simulation. At the same time, it leverages the Operand Tracker to analyze the generated instruction sequence and validate correctness, thereby overcoming the limitation of NVMain-PIM, which reports only latency and energy but cannot verify functional correctness. This integration enables functionally and quantitatively comprehensive simulation.

5.3 Results Analysis

Figure 5.1 shows the number of `AP` and `AAP` instructions generated by MIMDRAM and by our compiler for the 11 workloads mentioned earlier. Figure 5.2 and Figure 5.3 present the execution time, measured in power cycles, and the energy consumption, measured in μJ, obtained by simulating these instruction sequences using the NVMain-PIM simulator. It is important to emphasize again that since all instructions are issued within the same bank and the same subarray, the energy values reported in Figure 5.3 represent the total energy consumed by that subarray where computation takes place, including both active and refresh energy consumption.

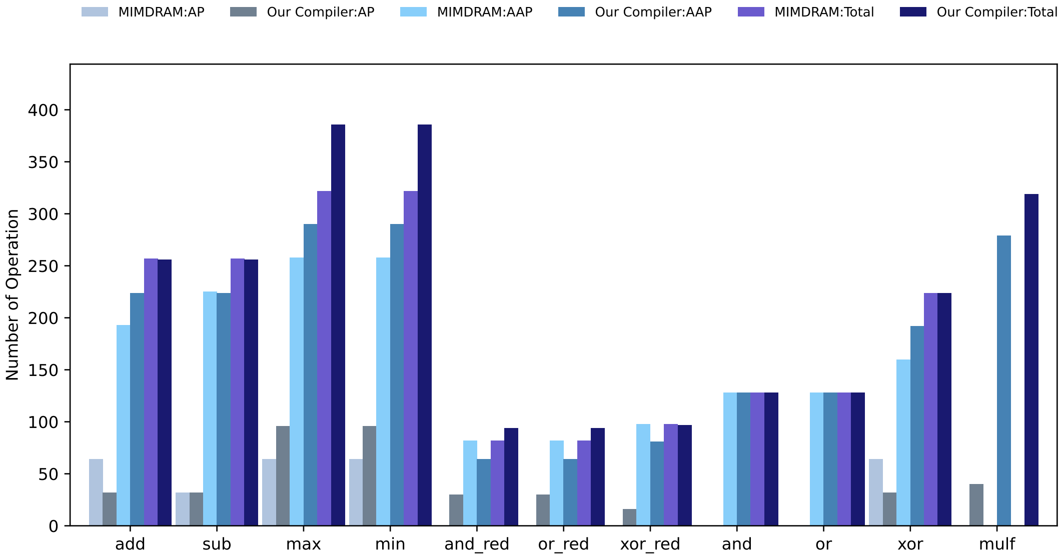

Figure 5.1: Number of AP/AAP Operations Generated by MIMDRAM and our Compiler

From Figure 5.1 it can be observed, in terms of the total number of instructions, our compiler performs on par with MIMDRAM in five workloads other than the maximum and minimum functions, with comparable numbers of `AP` and `AAP` instructions - sometimes higher in one, lower in the other. For the maximum and minimum functions, however, our compiler generates more `AP` and `AAP` instructions than MIMDRAM.

The primary reason for the shown difference between MIMDRAM and our work lies in the different strategies adopted by our compiler and MIMDRAM when mapping high-level operations into memory instruction sequences during logic synthesis. Specifically, MIMDRAM's design, which encapsulates basic operators as micro-programs, provides a finer granularity of control at the memory-instruction level. For example, when iterating from the LSB row to the MSB row, MIMDRAM can always temporarily store intermediate values - those required in the next iteration - directly in the computation-capable **B** group. This avoids the need to move the row back to the **D** group at the end of one iteration and then copy it back at the beginning of the next iteration, a process that would otherwise incur two additional `AAP` instructions. As operand bit-width increases (i.e., more iterations are required) or when multiple intermediate values are produced per iteration, the performance benefit of this approach becomes particularly significant.

In contrast, our compiler first generates a corresponding MIG logic network for each arithmetic function during the logic synthesis stage, and then employs Lime to translate it into the corresponding instruction sequence. To preserve the performance of the compiler itself, we only generate one iteration, i.e., the loop body, for each operator once and then execute it iteratively. Consequently, in order to maintain the completeness of the logic network during synthesis, intermediate values that need to be carried across iterations are treated as explicit inputs and outputs, and are allocated addresses in the **D** group for storage, which brings extra row clones in some cases.

In summary, the comparison of logic synthesis strategies reveals that MIMDRAM has hand-tuned and optimized the performance of individual operators to the extreme, but at the cost of generality and flexibility. For example, when handling a function composed of multiple additions, the MIMDRAM programming framework enforces a sequential execution: addition 1 must be completed and its result stored in the **D** group before addition 2 can begin, and so on. This not only limits instruction scheduling but also leads to inefficient utilization of the **D** group's storage resources within computation-capable subarrays.

By contrast, our compiler synthesizes the entire operator network at once and executes it iteratively as a whole. In this approach, only the inputs, final outputs, and the necessary carry signals are allocated **D** group addresses, while intermediate results from one addition can be directly consumed by the next within the **B** group, without being temporarily stored in the **D** group. Viewed from this broader perspective, our compiler demonstrates clear advantages in handling large computational functions composed of multiple operators. Moreover, in future work, we plan to explore operator scheduling strategies: decomposing independent operators into separate logic networks and exploiting bank- or subarray-level parallelism to further accelerate execution.

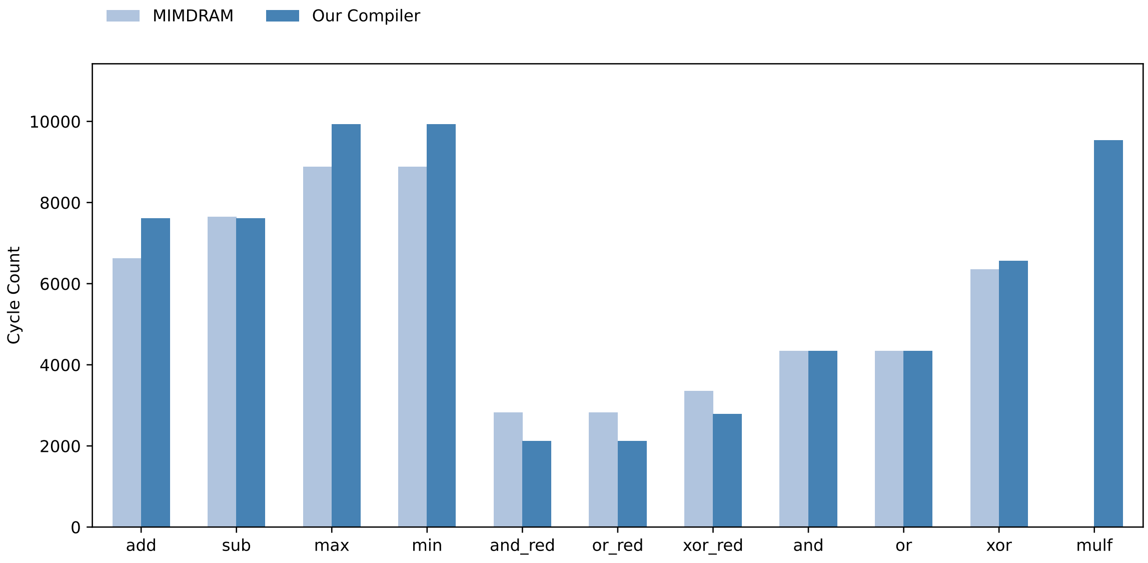

Figure 5.2: Execution Time in DRAM Cycle Count of MIMDRAM and our Compiler

In Figure 5.2 and Figure 5.3, the simulation results of execution time and energy consumption for the instruction sequences generated by MIMDRAM and our compiler closely mirror the

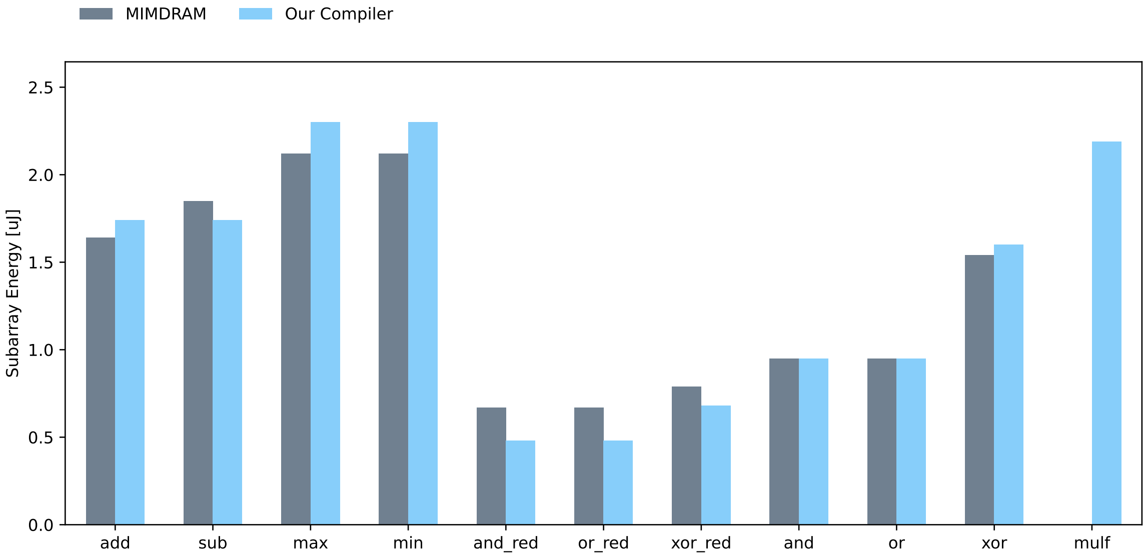

Figure 5.3: Energy Consumption in μJ of MIMDRAM and our Compiler

instruction count comparison presented in Figure 5.1. An interesting exception arises in the three reduction operations, where our compiler achieves both shorter execution time and lower energy consumption. This advantage stems from reduced data movement in our handling of both operators compared to MIMDRAM, which is reflected in the generation of fewer `AAP` instructions and a greater reliance on `AP` instructions. Since DRAM executes `AP` operations more quickly and with lower energy cost than `AAP` operations, this difference directly explains the observed performance and energy benefits.

6 Conclusion and Further Work

The slowdown of Moore's Law and the breakdown of Dennard scaling have fundamentally reshaped the landscape of computer architecture. As data movement increasingly dominates both performance and energy consumption, the traditional Von Neumann paradigm faces a growing imbalance between computation and memory. In-memory computing has emerged as a compelling alternative by performing computation directly where the data resides. Within this paradigm, in-DRAM computing - leveraging the intrinsic analog behavior of commodity DRAM circuitry - offers a particularly promising path toward scalable and energy-efficient data processing.

This thesis has presented the design and implementation of an MLIR-based compiler targeting bit-serial bulk-wise in-DRAM computing. The compiler bridges the gap between high-level arithmetic abstractions and low-level memory instruction sequences, providing a systematic and extensible compilation flow for in-DRAM architectures. By building upon the multi-level infrastructure of MLIR, our approach achieves a high degree of modularity: new dialects can be introduced to represent in-DRAM primitives, while existing dialects such as `arith` and `scf` serve as the front-end interface to describe arithmetic operations and control-flow. This design ensures compatibility with a wide range of programming frameworks—any language or compiler front-end capable of lowering to the `arith` dialect, such as C or PyTorch via `torch-mlir`, can transparently benefit from our compilation backend.

To support in-DRAM computing semantics, this work introduces two new dialects: `bits` and `pud`. The `bits` dialect models bit-serial operations, including both logical and arithmetic primitives, as well as reduction and comparison operators. These abstractions express computation at a level suitable for logic synthesis, enabling translation into logic networks through integration with the `mockturtle` framework. The `pud` dialect represents low-level in-DRAM operations, including the Ambit-inspired primitives `AP` and `AAP`, which form the executable instruction set for the DRAM controller backend. Together, these two dialects provide a structured pathway from arithmetic-level expressions to memory-instruction-level execution.

The compiler flow begins with the transformation of high-level tensor operations - typically expressed in the `arith` dialect - into bit-sliced representations via `bits.transpose`. Subsequent lowering passes convert element-wise and reduction operations into `bits` dialect instructions, which are then synthesized into MIG networks using `mockturtle`. These networks are further mapped to sequences of `pud` instructions, producing executable traces for Ambit-style DRAM architectures. This layered design cleanly separates concerns between front-end arithmetic expression, logic synthesis, and backend instruction generation, ensuring both modularity and reusability of each component.

Comprehensive evaluation demonstrates that our compiler achieves performance and energy efficiency comparable to state-of-the-art frameworks such as MIMDRAM, while maintaining significantly higher generality and flexibility. For workloads including addition, subtraction, maximum, minimum, and reduction operations, simulation results obtained from the NVMain-PIM platform show that the instruction counts, execution cycles, and energy consumption closely match those generated by MIMDRAM. The observed discrepancies are primarily due to differences in logic synthesis and intermediate value management - our compiler maintains explicit carry propagation and data dependencies within the logic network to preserve correctness, while MIMDRAM leverages micro-level optimizations at the instruction scheduling stage. Despite these differences, our approach demonstrates competitive efficiency while providing a unified and extensible compilation framework.

Compared to prior systems, the key strength of our compiler lies in its extensibility and integration capability. Unlike SIMDRAM and MIMDRAM, which rely on predefined micro-programs, or CHOPPER, which depends on a domain-specific compiler (Usuba), our MLIR-based approach decouples in-DRAM compilation from any particular language or DSL. By adopting the `arith` dialect as the universal front-end and defining modular dialects for bit-serial and memory-level representations, the compiler can naturally evolve alongside both hardware and software ecosystems. This extensibility enables future integration with additional logic synthesis back-ends, new DRAM instruction sets, or even alternative memory technologies.

In conclusion, this thesis contributes a complete and extensible compilation framework for in-DRAM computing. By integrating MLIR's dialect-based modular design with established logic synthesis and code generation tools, it demonstrates how high-level arithmetic computations can be systematically lowered to executable DRAM instructions. The results confirm that in-DRAM computing can be supported within a unified, general-purpose compiler infrastructure—bridging the gap between modern machine learning frameworks and emerging memory-centric architectures.

Through this work, we take a step toward making in-DRAM computing not merely a hardware novelty, but a programmable and accessible component of modern computing systems.

6.1 Future Work

While the current compiler demonstrates the feasibility and effectiveness of translating high-level arithmetic operations into in-DRAM instruction sequences, several directions remain for further improvement and extension.

6.1.1 Flexible Data Layout and Configuration

At present, the compiler assumes that all operands are in a horizontal layout and have not yet been stored in DRAM prior to compilation. This assumption simplifies the compilation flow but limits flexibility in practical use cases. In future versions, the compilation pipeline should incorporate options that allow users to explicitly specify, for each operand, whether it needs to be transposed, whether it is already in a vertical layout suitable for in-DRAM execution, and whether it has already been placed in DRAM. For operands already residing in memory, the compiler should also support user-defined metadata describing their precise DRAM locations. Such configurability would enable more flexible and efficient integration between pre-loaded data and runtime computation, reducing redundant transpositions and data movement.

6.1.2 Bank-Level Parallelism

Currently, the compiler targets a purely SIMD-style execution model, where all operations are executed uniformly across subarrays. To further exploit the inherent parallelism of DRAM architectures, future work will extend support to bank-level parallelism. This requires analyzing inter-operator dependencies within a computation function, partitioning independent subgraphs, and scheduling them across different DRAM banks for concurrent execution. Once the parallel computations complete, their partial results can be aggregated to produce the final output. Such an enhancement would transform the compiler from a SIMD-oriented model into a true MIMD execution framework, effectively leveraging fine-grained parallelism within and across subarrays.

6.1.3 Advanced Data Mapping

To enable MIMD-style computation like in [Oli+24; Far+24b], more flexible data mapping strategies are essential. Future versions of the compiler should support dynamic and analyzable allocation of D-group addresses for operands, intermediate values, and final results. When data must be distributed across multiple subarrays, the compiler should automatically determine mappings that minimize inter-subarray data transfers, thereby reducing both latency and energy consumption. Additionally, user-specified mapping policies should be supported, allowing advanced users to explicitly control data placement when integrating with specific hardware configurations or simulation environments.

6.1.4 Additional In-DRAM Computing Back-ends

As discussed in the conclusion, one of the major strengths of the proposed system lies in its modularity enabled by MLIR. This design allows the compiler to be easily extended with new dialects or alternative in-DRAM execution models. Beyond the Ambit-style primitives currently supported, future work will explore integration with other DRAM-based computing architectures, such as FCDRAM [Yük+24]. These extensions will further validate the generality of the MLIR-based framework and demonstrate its ability to serve as a unified compilation infrastructure for diverse in-memory computing paradigms.

6.1.5 Summary

In summary, future work will focus on improving the flexibility, scalability, and adaptability of the compiler. Enhancements in data layout handling, bank-level parallelism, and data mapping optimization will extend the current framework toward a full-featured, MIMD-capable compilation system. Meanwhile, the integration of additional DRAM computing back-ends will reinforce the extensibility of the MLIR-based approach, paving the way for a comprehensive and modular compiler ecosystem for next-generation in-memory computing architectures.

7 Acknowledgements

First and foremost, I would like to express my sincere gratitude to my supervisor, Dr. João Paulo Cardoso de Lima, for his continuous guidance and support throughout this work. His insightful discussions, weekly meetings, and thoughtful feedback have been invaluable and have greatly deepened my understanding of both the research process and the subject itself.

I am also deeply grateful to Prof. Dr.-Ing. Jerónimo Castrillón for the opportunities he has provided - from my *Studienarbeit* in 2023, through my internship as working student in 2024, to this final *Diplomarbeit*. His encouragement and trust have been essential in shaping my academic development and in making this research possible.

My thanks also go to Dr.-Ing. Asif Ali Khan and my second supervisor, Clément Fournier. The discussions with Asif were always inspiring and offered new perspectives on several technical aspects, while Clément's guidance on MLIR-related topics played a crucial role in the successful completion of this thesis.

Finally, I would like to thank my wife, all colleagues and friends who have supported me during this period, for their help, patience, and encouragement.

Bibliography

[Moo65] Gordon E. Moore. “Cramming more components onto integrated circuits”. In: *Electronics* 38.8 (1965), pp. 114–117.

[Den+74] Robert H. Dennard et al. “Design of ion-implanted MOSFET’s with very small physical dimensions”. In: *IEEE Journal of Solid-State Circuits* 9.5 (1974), pp. 256–268.

[GN00] Emden R. Gansner and Stephen C. North. “An open graph visualization system and its applications to software engineering”. In: *Softw. Pract. Exper.* 30.11 (Sept. 2000), pp. 1203–1233. ISSN: 0038-0644.

[Ham+10] Rehan Hameed et al. “Understanding sources of inefficiency in general-purpose chips”. In: *Proceedings of the 37th Annual International Symposium on Computer Architecture*. ISCA ’10. Saint-Malo, France: Association for Computing Machinery, 2010, pp. 37–47. ISBN: 9781450300537. DOI: 10.1145/1815961.1815968. URL: https://doi.org/10.1145/1815961.1815968.

[Esm+11] Hadi Esmaeilzadeh et al. “Dark silicon and the end of multicore scaling”. In: *SIGARCH Comput. Archit. News* 39.3 (2011), pp. 365–376.

[PX12] Matt Poremba and Yuan Xie. “NVMain: An Architectural-Level Main Memory Simulator for Emerging Non-volatile Memories”. In: *2012 IEEE Computer Society Annual Symposium on VLSI*. 2012, pp. 392–397. DOI: 10.1109/ISVLSI.2012.82.

[Ses+13] Vivek Seshadri et al. “RowClone: fast and energy-efficient in-DRAM bulk data copy and initialization”. In: *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-46. Davis, California: Association for Computing Machinery, 2013, pp. 185–197. ISBN: 9781450326384. DOI: 10.1145/2540708.2540725. URL: https://doi.org/10.1145/2540708.2540725.

[Hor14] Mark Horowitz. “1.1 Computing’s energy problem (and what we can do about it)”. In: *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*. 2014, pp. 10–14. DOI: 10.1109/ISSCC.2014.6757323.

[Gup+15] Suyog Gupta et al. “Deep learning with limited numerical precision”. In: *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*. ICML’15. Lille, France: JMLR.org, 2015, pp. 1737–1746.

[ITR15] ITRS Consortium. *International Technology Roadmap for Semiconductors 2.0.* https://www.semiconductors.org/resources/2015-international-technology-roadmap-for-semiconductors-itrs/. Accessed: 2025-09-10. 2015.

[Ses+16] Vivek Seshadri et al. *Buddy-RAM: Improving the Performance and Efficiency of Bulk Bitwise Operations Using DRAM.* 2016. arXiv: 1611.09988 `[cs.AR]`. URL: https://arxiv.org/abs/1611.09988.

[Wal16] M. Mitchell Waldrop. "The chips are down for Moore's law". In: *Nature* 530 (2016), pp. 144–147.

[Jou+17] Norman P. Jouppi et al. "In-Datacenter Performance Analysis of a Tensor Processing Unit". In: *Proceedings of the 44th Annual International Symposium on Computer Architecture.* ISCA '17. Toronto, ON, Canada: Association for Computing Machinery, 2017, pp. 1–12. ISBN: 9781450348928. DOI: 10.1145/3079856.3080246. URL: https://doi.org/10.1145/3079856.3080246.

[KSH17] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. "ImageNet classification with deep convolutional neural networks". In: *Commun. ACM* 60.6 (May 2017), pp. 84–90. ISSN: 0001-0782. DOI: 10.1145/3065386. URL: https://doi.org/10.1145/3065386.

[Ses+17] Vivek Seshadri et al. "Ambit: in-memory accelerator for bulk bitwise operations using commodity DRAM technology". In: *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture.* MICRO-50 '17. Cambridge, Massachusetts: Association for Computing Machinery, 2017, pp. 273–287. ISBN: 9781450349529. DOI: 10.1145/3123939.3124544. URL: https://doi.org/10.1145/3123939.3124544.

[Ahm+19] Hameeza Ahmed et al. "A compiler for automatic selection of suitable processing-in-memory instructions". In: *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE).* IEEE. 2019, pp. 564–569.

[Dev+19] Jacob Devlin et al. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers).* Ed. by Jill Burstein, Christy Doran, and Thamar Solorio. Minneapolis, Minnesota: Association for Computational Linguistics, June 2019, pp. 4171–4186. DOI: 10.18653/v1/N19-1423. URL: https://aclanthology.org/N19-1423/.

[19] "IEEE Standard for Floating-Point Arithmetic". In: *IEEE Std 754-2019 (Revision of IEEE 754-2008)* (2019), pp. 1–84. DOI: 10.1109/IEEESTD.2019.8766229.

[MD19] Darius Mercadier and Pierre-Évariste Dagand. "Usuba: high-throughput and constant-time ciphers, by construction". In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation.* PLDI 2019. Phoenix, AZ, USA: Association for Computing Machinery, 2019, pp. 157–173. ISBN: 9781450367127. DOI: 10.1145/3314221.3314636. URL: https://doi.org/10.1145/3314221.3314636.

[McK20] McKinsey & Company. *Domain-specific architectures and the future of compute.* https://www.mckinsey.com/industries/industrials-and-electronics/our-insights/domain-specific-architectures-and-the-future-of-compute. Accessed: 2025-09-10. 2020.

[Haj+21] Nastaran Hajinazar et al. "SIMDRAM: a framework for bit-serial SIMD processing using DRAM". In: *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems.* ASPLOS '21. Virtual, USA: Association for Computing Machinery, 2021, pp. 329–345. ISBN: 9781450383172. DOI: 10.1145/3445814.3446749. URL: https://doi.org/10.1145/3445814.3446749.

[Nar+21] Deepak Narayanan et al. "Efficient large-scale language model training on GPU clusters using megatron-LM". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis.* SC '21. St. Louis, Missouri: Association for Computing Machinery, 2021. ISBN: 9781450384421. DOI: 10.1145/3458817.3476209. URL: https://doi.org/10.1145/3458817.3476209.

[Zam+21] Pedram Zamirai et al. *Revisiting BFloat16 Training.* 2021. arXiv: 2010.06192 `[cs.LG]`. URL: https://arxiv.org/abs/2010.06192.

[Soe+22] Mathias Soeken et al. *The EPFL Logic Synthesis Libraries.* 2022. arXiv: 1805.05121 `[cs.LO]`. URL: https://arxiv.org/abs/1805.05121.

[PWY23] Xiangjun Peng, Yaohua Wang, and Ming-Chang Yang. "Chopper: A compiler infrastructure for programmable bit-serial simd processing using memory in dram". In: *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA).* IEEE. 2023, pp. 1275–1288.

[Wan+23] Zhengrong Wang et al. "Infinity stream: Portable and programmer-friendly in-/near-memory fusion". In: *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3.* 2023, pp. 359–375.

[Far+24a] Hamid Farzaneh et al. "C4CAM: A Compiler for CAM-based In-memory Accelerators". In: *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'24), Volume 3.* ASPLOS '24. La Jolla, CA, USA: Association for Computing Machinery, May 2024, pp. 164–177. ISBN: 9798400703867. DOI: 10.1145/3620666.3651386. URL: https://arxiv.org/abs/2309.06418.

[Far+24b] Hamid Farzaneh et al. "SHERLOCK: Scheduling Efficient and Reliable Bulk Bitwise Operations in NVMs". In: *Proceedings of the 61th ACM/IEEE Design Automation Conference (DAC'24).* DAC '24. San Francisco, California: Association for Computing Machinery, June 2024. ISBN: 9798400706011. DOI: 10.1145/3649329.3658485. URL: https://doi.org/10.1145/3649329.3658485.

[Kha+24] Asif Ali Khan et al. *The Landscape of Compute-near-memory and Compute-in-memory: A Research and Commercial Overview.* Jan. 2024. arXiv: 2401.14428 `[cs.AR]`. URL: https://arxiv.org/abs/2401.14428.

[Lim+24] João Paulo C. de Lima et al. “Full-Stack Optimization for CAM-Only DNN Inference”. In: *Proceedings of the 2024 Design, Automation and Test in Europe Conference (DATE)*. DATE’24. Valencia, Spain: IEEE, Mar. 2024, pp. 1–6. URL: https://ieeexplore.ieee.org/document/10546805.

[Nie+24] Michael Niemier et al. “Smoothing Disruption Across the Stack: Tales of Memory, Heterogeneity, and Compilers”. In: *Proceedings of the 2024 Design, Automation and Test in Europe Conference (DATE)*. DATE’24. Valencia, Spain: IEEE, Mar. 2024, pp. 1–10. URL: https://ieeexplore.ieee.org/document/10546772.

[Oli+24] Geraldo F Oliveira et al. “MIMDRAM: An end-to-end processing-using-DRAM system for high-throughput, energy-efficient and programmer-transparent multiple-instruction multiple-data computing”. In: *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE. 2024, pp. 186–203.

[Shi+24] Shunchen Shi et al. “CoPIM: A Collaborative Scheduling Framework for Commodity Processing-in-memory Systems”. In: *2024 IEEE 42nd International Conference on Computer Design (ICCD)*. IEEE. 2024, pp. 44–51.

[Yük+24] İsmail Emir Yüksel et al. “Functionally-Complete Boolean Logic in Real DRAM Chips: Experimental Characterization and Analysis”. In: *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 2024, pp. 280–296. DOI: 10.1109/HPCA57654.2024.00030.

[Guo+25a] Deyuan Guo et al. “PIMsynth: A Unified Compiler Framework for Bit-Serial Processing-In-Memory Architectures”. In: *IEEE Computer Architecture Letters* (2025).

[Guo+25b] Lidong Guo et al. “Towards Floating Point-Based Attention-Free LLM: Hybrid PIM with Non-Uniform Data Format and Reduced Multiplications”. In: *Proceedings of the 43rd IEEE/ACM International Conference on Computer-Aided Design*. ICCAD ’24. Newark Liberty International Airport Marriott, New York, NY, USA: Association for Computing Machinery, 2025. ISBN: 9798400710773. DOI: 10.1145/3676536.3676776. URL: https://doi.org/10.1145/3676536.3676776.

[Hao+25] Zhiwei Hao et al. *Low-Precision Training of Large Language Models: Methods, Challenges, and Opportunities*. 2025. arXiv: 2505.01043 `[cs.LG]`. URL: https://arxiv.org/abs/2505.01043.

[Hu+25] Xiaobo Sharon Hu et al. “Cross-Layer Design and Design Automation for In-Memory Computing based on Non-Volatile Memory Technologies”. In: *IEEE Design & Test, Special Issue on the 20 years of the IEEE CEDA* 42.6 (Aug. 2025), pp. 75–86. DOI: 10.1109/MDAT.2025.3603495. URL: https://ieeexplore.ieee.org/document/11142851.

[SU-25] SU-JonesLab. *NVMain-PIM*. Accessed: Oct. 12, 2025. 2025. URL: https://github.com/SU-JonesLab/NVMain-PIM.

[Jun25] Niklas Jungnitz. *Technology-Aware Cost Functions and Code Optimisation for Digital CIM using E-graphs*. Accessed: Oct. 12, 2025. 2025. URL: https://github.com/jungnitz/lime.

[Kha+25] Asif Ali Khan et al. “CINM (Cinnamon): A Compilation Infrastructure for Heterogeneous Compute In-Memory and Compute Near-Memory Paradigms”. In: *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS’25), Volume 4*. ASPLOS ’25. Rotterdam, The Netherlands: Association for Computing Machinery, Mar. 2025, pp. 31–46. ISBN: 9798400703911. DOI: 10.1145/3622781.3674189. URL: https://dl.acm.org/doi/pdf/10.1145/3622781.3674189.

[Lim+25] João Paulo Cardoso De Lima et al. “All-in-memory Stochastic Computing using ReRAM”. In: *Proceedings of the 62nd ACM/IEEE Design Automation Conference (DAC’25)*. DAC ’25. San Francisco, California: Association for Computing Machinery, June 2025.

[Liu+25] Jiantao Liu et al. “OptiPIM: Optimizing Processing-in-Memory Acceleration Using Integer Linear Programming”. In: *Proceedings of the 52nd Annual International Symposium on Computer Architecture*. 2025, pp. 867–883.

[Yao+25] Lian Yao et al. “DAGSIS: A DAG-Aware MAGIC based Synthesis Framework for In-Memory Computing”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2025).

[Zhe25] Tianrui Zheng. *Cinnamon*. Accessed: Oct. 12, 2025. 2025. URL: https://github.com/CapZTr/Cinnamon/tree/simdram.

[Lim+26] João Paulo C. de Lima et al. “Count2Multiply: Reliable In-Memory High-Radix Counting”. In: *Proceedings of the 32nd IEEE International Symposium on High-Performance Computer Architecture (HPCA 2026)*. IEEE. Los Alamitos, CA, USA: IEEE Computer Society, Feb. 2026.

List of Figures

2.1 Overview of DRAM Organisation [Lim+26] 6
2.2 Majority and Not Logic Implementation of Ambit [Lim+26] 8
2.3 Horizontal and Vertical Data Layout in DRAM Subarray 10

4.1 Overview of the Compilation Flow 21
4.2 Transposed Slice Type and the Original Tensor Type 23
4.3 `bits.slice` Type and `pud.row` Type 26
4.4 Single-bit Full Adder in Majority Gates and Inverters 28
4.5 Logic Network of the Running Example, Visualized with `Graphviz` [GN00] . . 29
4.6 FP16 Slice Representation 36

5.1 Number of AP/AAP Operations Generated by MIMDRAM and our Compiler 43
5.2 Execution Time in DRAM Cycle Count of MIMDRAM and our Compiler . . 44
5.3 Energy Consumption in μJ of MIMDRAM and our Compiler 45

List of Tables

4.1 Classification of Operations in the `bits` Dialect 24
4.2 Assigned Address of each Signal 31
4.3 Operating Address of each Signal at the $8th$ Iteration 32

5.1 Accuracy of Approximate FP Multiplication 40

List of Listings

1 Addition of Three Integer Vectors in `arith` Dialect 22
2 Addition of Three Integer Vectors in `bits` Dialect 25
3 Pseudocode of `AP` and `AAP` Instructions for our Example 29
4 Instructions Including Row Indices of 8-th Iteration for our Example 33