Embedded Systems Lecture Notes:

Memory Systems and Domain-Specific Architectures (DSA)

1. Understanding Cache Limitations and Memory Hierarchy

Why Cache Size Matters

In modern computing systems, cache memory plays a crucial role in performance, but faces significant limitations:

- **L1 and L2 Cache Constraints**
    - L1 cache is typically limited to a few kilobytes up to ~200KB
    - L2 cache, while larger, still has strict size limitations
    - These limitations directly impact how we handle large data structures

Practical Example: Matrix Operations Let's consider a real-world scenario with a 1k × 1k matrix:

- Each integer requires 4 bytes of storage
- Total memory requirement = 1000 × 1000 × 4 bytes = 4MB
- This far exceeds typical L1 cache size (~200KB)
- Cannot fit entire matrices in cache simultaneously

Why Can't We Just Make Caches Bigger?

- Physical chip area is limited
- Larger caches:
    - Increase manufacturing costs
    - Consume more power
    - Can actually slow down access times due to increased complexity
    - Take space away from other critical components

Solution: Block-Based Processing

To overcome these limitations, we implement block-based processing:

1. **Matrix Splitting Strategy**
    - Break large matrices into smaller, manageable blocks

- Each block is sized to fit in cache
- Process multiplication block by block

Code Implementation Changes Original approach (inefficient for large matrices):
python
Copy
```python
for i in range(N):

    for j in range(N):

        C[i][j] = A[i][j] + B[i][j]
```
Block-based approach:
python
Copy
```python
for block_i in range(0, N, BLOCK_SIZE):

    for block_j in range(0, N, BLOCK_SIZE):

        for i in range(block_i, min(block_i + BLOCK_SIZE, N)):

            for j in range(block_j, min(block_j + BLOCK_SIZE, N)):

                C[i][j] = A[i][j] + B[i][j]
```
2. Benefits:
   - Better cache utilization
   - Improved temporal locality
   - Reduced cache misses

2. Domain-Specific Architecture (DSA): A Deep Dive

What Makes DSAs Different?

Domain-Specific Architectures are specialized computing systems designed for particular application domains. Unlike general-purpose processors, they prioritize efficiency for specific types of computations.

Key Characteristics:

1. Optimized for domain-specific operations
2. Higher efficiency than general-purpose systems
3. Limited flexibility but superior performance in target domain

Real-World Applications

Image Processing Domain

Understanding the Basics:

- Images are represented as pixel grids (matrices)
- Each pixel contains intensity/color information
- Common operations include:
    - Blending: Combining multiple images
    - Denoising: Removing unwanted artifacts
    - Blurring: Smoothing image details

Stencil Operations Explained:

Plus-Shaped Stencil:
Copy

```
   [P]

[P][C][P]

   [P]
```

- C: Center pixel
- P: Peripheral pixels
- Used for edge detection, blurring

1. Processing Method:

    - Window slides across image
    - Applies operation to pixels within stencil
    - Updates center pixel based on computation
    - Moves to next position

Machine Learning Applications

Matrix Operations in Neural Networks:

1. Basic Operations:
    - Matrix multiplication dominates computations
    - Example in CNN: `output = weight_matrix × input_matrix`
2. Convolution Transformation:
    - Traditional convolution is computationally expensive
    - Im2Col transformation converts convolution to matrix multiplication
    - Benefits:
        - Reuses existing matrix multiplication hardware
        - More efficient on modern architectures

3. Four Core Principles of DSA Design

A. Specialized Hardware and Data Types

Data Type Specialization: Traditional systems are constrained:

- Must use 32/64-bit floating-point
- Integer values always use 32 bits minimum

DSA Approach:

Copy

```
Example: Value range 0-7

Traditional: 32 bits required

DSA: Only 3 bits needed (2³ = 8 values)
```

Real-World Example: Neural Networks

- Traditional Network: 32-bit floating-point weights
- Optimized Network: 8-bit integer weights
- Binarized Neural Network: 1-bit weights (0 or 1)
- Result: Massive reduction in memory and computation requirements

B. Parallelism in DSAs

Understanding Multi-Level Parallelism: Unlike general-purpose processors with 8-32 cores, DSAs implement parallelism at multiple levels:

1. **Processing Element (PE) Level**
   - Multiple Multiply-Accumulate (MAC) units per PE
   - Each MAC operates independently

Example:
Copy

```
PE Structure:
PE1: [MAC1][MAC2][MAC3][MAC4]
PE2: [MAC1][MAC2][MAC3][MAC4]
```

2. **Module Level**
   - Multiple PEs work together
   - Each module handles specific computation type

Example from Transformer Accelerator:
Copy
```
Attention Module:
  - PE1: Self-attention computation
  - PE2: Key-value processing
  - PE3: Query processing
```

3. **System Level**
    - Multiple modules operate concurrently
    - Independent data streams processed simultaneously
    - Reduces overall processing time

Memory Considerations in Parallel Processing:

- Local memory per PE reduces data movement
- Hierarchical memory structure matches parallelism levels
- Minimizes communication overhead

C. Memory System Optimization

Why Memory Matters: Traditional memory access is extremely expensive:

- Energy cost increases with distance
- Data movement often dominates total system energy

Example:
Copy
```
Operation Energy Costs:
32-bit ADD: 0.1 pJ
32-bit MULT: 3.1 pJ
32-bit DRAM access: 640 pJ
```

DSA Memory Solutions:

1. **Localized Memory Architecture**
    - Small, distributed memory units
    - Located close to processing elements
    - Reduces data movement distance
2. **Data Compression Techniques**

    Benefits:

    - Reduced memory footprint

    - Increased effective bandwidth
    - Lower error probability

  Trade-offs:

    - Compression overhead
    - Decompression latency
    - Implementation complexity

Example:
Copy

```
Original: 64-byte data word
Compressed: 4-byte representation
Result: 16x effective bandwidth increase
```

D. Control Overhead Reduction

Understanding Pipeline Concepts:

Latency vs Throughput Using the laundry analogy:
Copy

```
Process Steps:
Washing (45 min) → Drying (60 min) → Pressing (15 min)

Sequential Processing:
Total time per load = 120 minutes
Throughput = 1 load/120 minutes

Pipelined Processing:
Initial load still takes 120 minutes
But next load starts after first wash (45 min)
Throughput = 1 load/45 minutes (after pipeline fills)
```

1. **In-Order vs Out-of-Order Execution** In-Order Processing:
    - Strict sequential execution
    - Waits for each step to complete
    - Simpler control logic
    - Lower performance
2. Out-of-Order Processing:
    - Dynamic instruction scheduling
    - Utilizes idle time
    - Complex control logic
    - Higher performance

Practical Implications and Applications

1. **System Design Considerations**
   - Match architecture to application domain
   - Balance parallelism with memory access
   - Consider energy efficiency at all levels
2. **Performance Optimization**
   - Use appropriate data types
   - Exploit natural parallelism in algorithms
   - Minimize data movement
   - Reduce control overhead
3. **Trade-offs to Consider**
   - Flexibility vs Efficiency
   - Area vs Performance
   - Power vs Speed
   - Complexity vs Maintainability

Near-Memory Computing

Introduction

- **Role of GPUs**: GPUs excel in accelerating applications that can perform the same instruction on multiple data (SIMD).
    - **Compute-bound applications**: Dominated by intensive computations.
    - **Memory-bound applications**: Dominated by data reads and writes.

Examples:

- **Compute-bound**: Image processing and rendering.
- **Memory-bound**: Database queries that require extensive data access.
- **Operations per byte (Ops/byte)**: Measures the ratio of operations to data movement.
    - **High Ops/byte**: Indicates compute-bound applications.
    - **Low Ops/byte**: Indicates memory-bound applications.
- **Benchmarks**: The **Standard Performance Evaluation Corporation (SPEC)** provides benchmarks to assess compute-bound and memory-bound applications.
- **GPU Programming Structure**:
    - **Host part**: Runs on the CPU.
    - **Device part**: Runs on the GPU.

Key Principles of Domain-Specific Architectures

1. **Specialization**: Focus on data and hardware for specific tasks.
2. **Parallelism**: Uses SIMD, MIMD, etc.
3. **Efficient Memory Organization**: Relies on small, local memories.
4. **Reduced Control Overhead**: Avoids extensive control circuitry.

Traditional Computing Systems: Compute Outside Memory

- **Data Movement**: CPU fetches instructions and data from memory, moving data over the same bus, which creates a **bottleneck**.
- **Bus Limitation**: Increases in CPU and memory speeds are limited by bus capacity, impacting overall performance.

Non-Von-Neumann Computing Paradigms

Near and In-Memory Computing

1. **Compute Near Memory (CNM)**:
    - Adds logic close to memory arrays on the same die.
2. **Compute In Memory (CIM)**:
    - Performs operations directly in memory arrays.
    - Types:
        - **Same-array in-memory computing (SAM)**: Utilizes memory cells for computation.
        - **Same-peripheral in-memory computing (SMP)**: Uses peripheral circuits for computation.

Question: Why not replace the CPU entirely with near-memory computing?

- **Physical Limitations**: Die size restricts capacity.
- **Complexity**: Control logic integration would reduce memory capacity and increase synchronization issues.
- **Hybrid Approach**: Host CPU performs general operations, offloading specific tasks to near-memory devices.

Data Transfer and Computation Offloading

- **Dedicated Units**: Data processing units (ALUs) are integrated within memory systems.
- **Uploading**: Transferring computations to specific devices.

Question: How does the CPU decide when to upload computations?

- **Cost Models**: Device-specific models estimate performance, but generalized models are under research.

Architectural Taxonomy

1. **Compute Outside Memory**: Traditional systems.
2. **Compute Near Memory**: Adds logic near memory arrays.
3. **Compute In Memory**: Performs operations within memory arrays (SAM & SMP).

Near-Memory Architectures Overview

1. **UPMEM (France)**: Specialized DRAM chips with integrated compute units.
2. **HBM-PIM (Samsung)**: Near-memory architecture using DRAM.

3. **SK Hynix**: DRAM-based near-memory solution.

UPMEM Architecture

- **Inspired by GPUs**: Integrates small compute units on DRAM chips.
- **Chip Components**:
    - **MRAM**: Main RAM.
    - **WRAM**: Working RAM or scratchpad.
    - **Control Interface**: For DMA.
    - **DPU (Data Processing Unit)**: Simple RISC core with a pipelined architecture.

Question: What's the difference between scratchpads and caches?

- **Caches**: Hardware-managed.
- **Scratchpads**: Software-managed, explicitly allocated by programmers.

Direct Memory Access (DMA)

- **Purpose**: Transfers data between memory and other components without CPU intervention.
- **Setup**: CPU sets parameters; DMA engine completes the transfer independently.

UPMEM Structure

- **Organization**:
    - 2 ranks per DIMM, 8 chips per rank, 8 DPUs per chip.
    - Each DPU has: 64 MB MRAM and 64 KB WRAM (scratchpad).
- **DPU Characteristics**:
    - 14-stage pipeline, 24 threads, 450 MHz.
    - Limited communication; relies on host for data sharing.

UPMEM DPU ISA

- **RISC-based**.
- **No FPUs**: Floating-point operations are emulated.
- **Multipliers**: Initially emulated in software; recent versions have small 8-bit multipliers.
- **No vector instructions**.

Question: Why use hardware FPUs and multipliers over software emulation?

- **Hardware Units**: Accelerate floating-point and multiplication operations.
- **Software Emulation**: Adds multiple instructions, creating performance overhead.

Question: What is software emulation?

- **Emulation**: Simulates missing hardware functions in software.
- **Example**: Floating-point addition split into integer operations, handling carry separately.

UPMEM Programmability

- **SDK**: Includes simulator and compiler toolchain.
- **Two-Part Application**:
    - **Host Code**: Compiled for x86, manages DPU allocation and data.
    - **Device Code**: Compiled using dpu-clang, manages synchronization, memory, and DMA.
- **Execution Modes**:
    - **Synchronous**: Host waits for DPU completion.
    - **Asynchronous**: Host and DPUs operate independently.

High-Level Compilation Frameworks for UPMEM

- **Purpose**: Simplifies UPMEM programming by abstracting details.
- **Example**: **Cinnamon flow**
    - Analyzes code, identifies parallelizable parts, and generates host and device code.
    - Produces optimized, sometimes superior code compared to manual efforts.

Code Analysis

- **Intermediate Representation (IR)**: Platform-independent code for optimization.
    - **LLVM IR**: Widely used.
    - **MLIR**: New approach by Google.
- **Analysis Focus**:
    - Identifies operations and loops.
    - Examines data dependencies for parallelization potential.

Samsung's FIMDRAM Architecture

- **Specialized for Machine Learning**:
    - Uses HBM interface for high bandwidth.

- **Key Features**:
    - Bank groups, computing units with SIMD FPUs, MAC units, and registers.
    - Bank-level parallelism with simple control circuitry.
- **Software Stack**: Proprietary but partially open-source simulator.

SK Hynix's AiM Architecture

- **Designed for Machine Learning**:
    - **Components**: MAC units, adder tree, SRAM.
    - **Memory**: Uses GDDR DRAM for balanced bandwidth.
    - **Efficiency**: Supports row cloning for optimized data manipulation.

Conclusion

- **No Best Architecture**: Choice depends on application needs.
- **Future Directions**: Memory accelerators, in-memory computing, and memory-centric technologies.
- **Integration**: Anticipated to work alongside CPUs, GPUs, and FPGAs in heterogeneous systems.

UPMEM Systems and DMA

1. Why are Direct Memory Access (DMA) engines introduced in UPMEM systems, and what is their purpose?

DMA engines are introduced in UPMEM to efficiently transfer data, freeing the CPU from continuous monitoring and allowing it to perform other tasks. Normally, transferring small data amounts from main memory to caches or scratchpads requires the CPU to monitor each byte sequentially, which can be slow. With DMA, the CPU initiates the transfer by specifying the data amount, source address, and destination. The DMA then handles the transfer, and once complete, it sends an interrupt to the CPU, indicating the transfer is done.

2. What are the components of UPMEM, and how do they work?

UPMEM architecture uses a distributed computing model similar to GPUs, integrating DPUs (Data Processing Units) within DRAM chips to enable local computation on data. The main components are:

- **Host:** The main CPU with traditional DRAM, managing DPU tasks and data transfer.
- **Device:** UPMEM DIMMs with DPUs.
- **DPU (Data Processing Unit):** Lightweight RISC processors with threads but no floating-point units, requiring software-emulated floating-point operations.
- **MRAM (Main RAM):** DRAM attached to each DPU, acting as device memory.
- **WRAM (Working RAM):** Small, fast SRAM as scratchpad memory, managed by software for fast data access.
- **IRAM (Instruction RAM):** Stores instructions for each DPU.

Workflow: The host transfers data to device memory (MRAM), assigns tasks to DPUs, which process data in MRAM and WRAM, and transfers the results back to the host memory.

---

3. Why can't DPUs communicate directly?

DPUs cannot communicate directly to keep control logic and data coherency protocols simple. Direct communication between DPUs would require complex consistency mechanisms. Instead, DPUs share data through the host, which can slow data sharing but maintains data consistency.

---

4. What is scratchpad memory in UPMEM systems?

Scratchpad memory in UPMEM is represented by WRAM. It's a small, fast memory managed by software, unlike caches managed by hardware. It provides direct control over data storage and retrieval, allowing optimization of data access for each DPU.

---

5. What is the DPU ISA?

The DPU uses a simplified RISC ISA, like ARM processors but with fewer instructions. Due to the lack of floating-point units, DPUs rely on software emulation for floating-point operations, which slows performance but allows essential computations.

---

6. How is multiplication emulated in software in UPMEM systems?

Without hardware multipliers, DPUs use software-emulated multiplication via shift and add operations. This method breaks down multiplication into shifts and adds, effectively emulating the process in integer instructions.

7. How are UPMEM systems programmed, and what is host and device memory?

8. What are synchronous and asynchronous workflows in UPMEM host and device interaction?

In a synchronous workflow, the host waits for DPU tasks to finish before proceeding, ensuring data availability. Asynchronously, the host initiates DPU tasks and continues with other processes, enabling greater parallelism but requiring the programmer to manage synchronization carefully.

---

9. How do MRAM and WRAM affect efficiency in UPMEM systems?

WRAM provides fast, direct access, while MRAM is larger but slower. Efficiently managing data between these two improves DPU processing speed and overall system performance, as frequent data transfers to and from MRAM can slow down the DPUs.

---

10. What is data coherency, and how is it managed in UPMEM systems?

Data coherency ensures consistent, up-to-date values across DPUs, critical when multiple DPUs access shared data. For instance, if DPUs modify variable "A" concurrently, UPMEM’s DPU libraries use synchronization to maintain coherency, ensuring each DPU reads and writes the latest value.

---

11. What is load balancing, and why is it essential in UPMEM systems?

Load balancing distributes tasks across DPUs to maximize system utilization. Efficient load balancing prevents overloading certain DPUs while leaving others idle, achieving faster execution and optimized resource use. Independent tasks execute in parallel, enhancing UPMEM’s parallel processing benefits.

12. What is a summary of the UPMEM system?

UPMEM offers localized computation in memory, reducing data movement and improving efficiency. However, the lack of direct DPU-to-DPU communication and a limited instruction set means it’s best suited for applications with repetitive, data-intensive tasks.

---

13. What are basic blocks in compilers?

Basic blocks are instruction sequences with a single entry and exit point, used as the fundamental building blocks for compiler analysis and optimization.

---

14. How does Multi-Level Intermediate Representation (IR) work?

MLIR by Google allows multi-level code abstraction, aiding analysis and optimization across hardware targets, supporting hardware-specific optimizations and better performance across diverse architectures.

---

15. What is Cinnamon flow in UPMEM?

Cinnamon is a high-level UPMEM compiler framework that takes general code and generates optimized host/device code, handling tasks like load balancing for efficient DPU use.

---

16. What is BLAS optimization?

BLAS (Basic Linear Algebra Subprograms) is a library of optimized routines for linear algebra, providing a performance benchmark for optimizing similar computational tasks on UPMEM.

---

17. What are scatter and gather operations?

Scatter distributes data from the host to multiple DPUs, while gather collects processed results from DPUs back to the host. These operations are essential for efficient data transfer in parallel UPMEM processing.

---

18. What is Samsung FIMDRAM, and how does it relate to SK Hynix AiM?

Samsung FIMDRAM and SK Hynix AiM are alternative CNM architectures designed for machine learning. FIMDRAM uses high-bandwidth memory and SIMD FPUs for high data throughput, while AiM features optimized operations for ML workloads, including multiplier/adder trees.

19. How does row cloning work?

Row cloning enables fast data copying within DRAM banks, allowing direct row-to-row data transfers without CPU involvement. This method leverages DRAM structure for faster data handling, improving memory operation speed.

Step-by-Step Guide to Designing a Computing Architecture

1. Understand the Requirements of the System

- **Purpose**: What will this system do? Is it meant for general-purpose computing, machine learning, real-time processing, or something else?
- **Performance Needs**: Determine the speed and efficiency required. For example, do you need low latency (fast responses) or high throughput (processing lots of data quickly)?
- **Resource Constraints**: Are there limitations on power consumption, size, or budget? Embedded systems, for instance, often have strict power and size constraints.
- **Scalability**: Will this system need to handle more workload in the future, or is it a single-use setup?

2. Choose the Core Architecture Type

- **Von Neumann vs. Harvard**:
  - **Von Neumann** is simpler, using a single memory for both instructions and data.
  - **Harvard** separates instructions and data, which can improve performance but adds complexity.
- **Specialized Architectures**: Consider Domain-Specific Architectures (DSAs) if you're focusing on a specialized task (e.g., GPUs for graphics, TPUs for deep learning).

3. Select the Instruction Set Architecture (ISA)

- **CISC vs. RISC**: CISC (Complex Instruction Set Computing) has many specialized instructions, often used in desktops. RISC (Reduced Instruction Set Computing) is simpler and can be more efficient, commonly used in embedded and mobile devices.
- **Determine Instruction Needs**: Do you need instructions for basic tasks (RISC) or complex operations (CISC)?

4. Design the Memory Hierarchy

- **Cache and Main Memory**: High-speed caches are critical to reduce latency. Design a hierarchy (L1, L2, L3) where smaller, faster caches store frequently used data, and larger, slower memories hold less-used data.
- **Data Flow**: Optimize how data moves between CPU and memory, minimizing the distance data needs to travel and thus reducing delay.
- **Memory Access Pattern**: Will data be accessed sequentially or randomly? Sequential patterns benefit from cache optimization, while random access may require other techniques like wider buses.

5. Define the Processing Elements (PEs)

- **Parallelism**: How many cores or processing elements are needed? For example, tasks like image processing may need multiple cores working in parallel.
- **Specialized Units**: Do you need units specifically for tasks like matrix multiplication (common in deep learning)?
- **Near-Memory Computing**: Placing processing close to memory to reduce data transfer time may be an option (e.g., near-memory architectures like UPMEM).

6. Optimize for Power and Performance

- **Power Consumption**: For embedded systems, power efficiency is critical. Techniques like dynamic voltage scaling and turning off unused components help.
- **Latency vs. Throughput**: Balancing response time (latency) and total data processed (throughput) depends on the use case.

7. Ensure Programmability and Flexibility

- **Programming Model**: Choose a model that aligns with your architecture. DSAs often require specific programming frameworks (e.g., CUDA for GPUs).
- **Adaptability**: Think about whether you need flexibility in software updates or hardware modifications. General-purpose processors are more flexible, while ASICs (Application-Specific Integrated Circuits) are efficient but fixed.

Example Problem Statement and Solution

Problem Statement:

Design an embedded architecture for a **smart traffic light system** that can:

- Detect the number of cars at each intersection.
- Prioritize lanes with heavier traffic to improve flow.
- Operate in real-time with minimal delay.
- Use minimal power, as it will be powered by a solar battery.

Solution:

1. **Requirements Analysis**
    - **Real-Time Operation**: Requires fast response and minimal latency.
    - **Power Efficiency**: Needs low power consumption as it's solar-powered.
    - **Limited Memory & Compute Needs**: Will be handling basic traffic detection and lane priority algorithms, so high computational power isn't necessary.
2. **Core Architecture Choice**:
    - **Harvard Architecture**: We'll use this because it allows separate data and instruction memory, improving speed in real-time tasks.
    - **RISC-Based ISA**: Choose a RISC architecture as it is simpler, uses less power, and is effective for embedded systems with basic instructions.
3. **Memory Hierarchy Design**:
    - **Small, Local Cache**: An L1 cache can hold the most recent traffic data, reducing main memory access.
    - **Non-Volatile Memory**: For storing firmware and system settings in case of power loss.
4. **Processing Elements (PEs)**:

    - **Single-Core CPU with Low-Power Mode**: Since it doesn't require heavy processing, a single-core CPU with a low-power mode will save energy.
    - **Small Digital Signal Processor (DSP)**: For faster processing of image or sensor data to count cars. DSPs are optimized for such tasks and use less power than a general-purpose CPU.
5. **Data Flow and Near-Memory Processing**:
    - **Direct Sensor Data Input to DSP**: Connect the car sensors directly to the DSP to minimize data movement, reducing latency.
    - **Data Compression**: Compress the sensor data when storing in cache to fit more data without increasing cache size.
6. **Power Optimization**:
    - **Dynamic Power Management**: Turn off DSP when no cars are detected for a period.
    - **Adjustable Light Brightness**: Adjust the brightness of traffic lights based on ambient light to save power.
7. **Programming and Control Logic**:
    - **Basic Control Software**: Simple logic for prioritizing lanes based on data from the DSP.
    - **Automated Firmware Updates**: Built-in programmability for updates as traffic patterns change over time.

Summary of the Solution

This smart traffic light system is a **Harvard-based RISC architecture** with **near-memory processing** for minimal data movement and a **DSP for sensor data** processing. By optimizing power and using minimalistic design elements, this solution meets the requirements effectively.

Compute Near Memory (CNM) Recap

- **Overview**:
    - Traditional memory systems include a CPU, memory, and a bus connecting them. Instructions and data must travel via this bus.

    - CPU fetches instructions and data as needed, creating bottlenecks due to bus movement and memory speed limitations.
- **Data Access**:
    - Accessing cache data is at least 100x faster than memory, prompting a shift toward placing compute units near memory.
- **CNM Design**:
    - CNM keeps the primary memory array unchanged but adds compute units around it.
- **Examples**:
    - **UPMEM**: Uses commercially available CNM.
    - **General-purpose RISC CPU**: A general-purpose processor for CNM.
    - **Samsung HBM systems with AIM**: Specialized for machine learning, with max units followed by tree structures.
- **Programming Complexity**:
    - CNM systems have unique software ecosystems for managing data movement between main memory and compute units.
    - **UPMEM Flow**: Simplifies programming by abstracting hardware details, allowing Python-based code while managing data movement.
- **Benefits**:
    - Reduces data movement over the external bus compared to traditional systems but retains movement between memory array and compute units.

Compute In Memory (CIM) Introduction

- **Overview**:
    - CIM performs computation using memory cells, aiming to eliminate data movement by using the memory device's physical properties.
    - Also known as in-memory computing, processing in memory, or processing using memory.
- **Characteristics**:
    - CIM systems are not general-purpose; specific memory devices allow specific operations.

Types of CIM Systems

1. **SIMD (Single Instruction, Multiple Data)**:
    - Executes operations on multiple data elements simultaneously.

    - **Types of SIMD in CIM**:
        - **Basic**: Adds logic in sensing circuitry without modifying memory array.
        - **Hybrid**: Adds support in sense amplifiers with partial support from the memory array.
2. **Applications**:
    - **Matrix-vector multiplication**: Achieves constant time complexity (O(1)) compared to traditional $O(n^2)$ or $O(n^3)$ methods.
    - **Bulk bitwise logic operations**: Useful for database operations where data movement is a bottleneck.
    - **CAM (Content Addressable Memory)**: Enables constant-time search operations for applications like database queries and recommendation systems.

Memory Technologies for CIM

- **Types of Memory**:
    - **SRAM (Static RAM)**: Common but volatile, requiring continuous power.
    - **DRAM (Dynamic RAM)**: Volatile, with refresh requirements.
    - **NVM (Non-Volatile Memory)**: Retains data without power, with types including:
        - **PCM (Phase Change Memory)**: Uses resistive material switching between high/low resistance.
        - **MRAM (Magnetic RAM)**: Stores data through magnetic orientation, similar to hard drives.
        - **RRAM (Resistive RAM)**: Uses resistive states for data storage.
        - **Racetrack Memory**: A newer NVM using magnetic material with sequential data access.
- **Challenges of NVM**:
    - Slow and power-intensive write operations.
    - Limited write endurance due to finite write cycles.
- **Optimizations**:
    - **Wear leveling**: Distributes writes evenly across memory to prevent cell failure.
    - **Non-flash storage cache**: Temporarily holds frequent writes before transferring to NVM.

Implementing CIM Using a Crossbar Array

- **Crossbar Array**:
    - Used for matrix-vector multiplication within memory using memristors.
- **Process**:
    - Program matrix (G) into memory cells (memristors).

    - Activate all rows simultaneously.
    - Apply input vector (Vi) to columns.
    - Cumulative current at bit lines yields the dot product for each row of the matrix.
- **Matrix-Matrix Multiplication**:
    - Achieved by sequentially applying transposed rows of one matrix to the crossbar array.
    - Computation occurs in the analog domain, resulting in approximate results suitable for machine learning.

Examples of CIM Accelerators

1. **HP Lab CIM Accelerator**:
    - One of the first accelerators built on crossbar architecture.
2. **Other CIM Systems**:
    - **STEMS**: Demonstrated CIM feasibility.
    - **COMAS**: Based on STEMS with user-friendly programming interface and a simulator (COMAS-Sim) for testing.

- **CIM Accelerator Architecture**:
    - Composed of:
        - **Tiles**: Multiple compute units per tile.
        - **Crossbars**: For matrix operations.
        - **ADCs/DACs**: Digital-to-analog and analog-to-digital conversion.
        - **Registers**: Input/output storage.
        - **Shift-and-hold registers**: For multi-bit integer multiplication shifting and accumulation.