Address Increment Patterns for DMA and Descriptor-Based Data Movement

Document Version: 1.0 **Last Updated:** 2026-01-13 **Scope:** Comprehensive analysis of address increment patterns for descriptor-based systems **Reference Implementation:** RAPIDS (Rapid AXI Programmable In-band Descriptor System)

Table of Contents

---

1. Executive Summary

Address increment patterns determine how DMA engines and descriptor-based data movement systems calculate memory addresses during transfers. The choice of pattern significantly impacts:

- **Bandwidth efficiency**: Alignment and burst optimization
- **Application fit**: Different workloads require different patterns
- **Hardware complexity**: More patterns = more control logic
- **Software flexibility**: Richer patterns = simpler programming model

Pattern Categories

Category	Description	Use Cases
Linear/Contiguous	Sequential address increment	Block copy, streaming data
2D Stride	Row-major with inter-row gaps	Image processing, video frames
3D Volume	Multi-plane with inter-plane gaps	3D imaging, tensor operations
Scatter-Gather	Non-contiguous fragment list	Protocol buffers, fragmented memory
Circular	Wrap-around at boundary	Audio streaming, network buffers
Tensor	Multi-dimensional with strides	Neural network accelerators

---

2. RAPIDS Current Implementation

2.1 Descriptor Format (256-bit)

RAPIDS uses a fixed 256-bit descriptor structure defined in `rapids_pkg.sv`:

```
Bit Field          Width   Description
──────────────────────────────────────────────────────────────
[63:0]              64-bit  src_addr - Source address (byte-
aligned)
[127:64]         64-bit  dst_addr - Destination address (byte-
aligned)
[159:128]       32-bit  length - Transfer length in BEATS (not bytes)
[191:160]                    32-bit    next_descriptor_ptr  -
Next descriptor address
[207:200]           8-bit  desc_priority - Transfer priority
[199:196]           4-bit  channel_id - Channel ID
[195]               1-bit  error - Error flag
[194]               1-bit  last - Last descriptor in chain
[193]               1-bit  gen_irq - Generate interrupt
[192]               1-bit  valid - Valid descriptor
[255:208]          48-bit  reserved - Future use
```

2.2 Current Address Increment Modes

Mode 1: Beat-Based Linear (Default)

```
Address Calculation:
  next_addr = current_addr + (beat_count × 64 bytes)
```

```
Where: RAPIDS_BYTES_PER_BEAT = 64 bytes (512-bit data width)
```

- All transfers aligned to 64-byte boundaries for optimal throughput
- Descriptor length field specifies count in beats
- Software converts byte count to beats: `beats = (bytes + 63) >> 6`

Mode 2: Chunk-Based Alignment (Sub-Beat Transfers)

For alignment/final phases handling partial beats:

```
Address Calculation for Alignment Phase:
  offset = address[5:0]                 // 6-bit offset (0-63)
  bytes_to_boundary = (offset == 0) ? 0 : 64 - offset

  Transfer Phases:
 Phase 1 (Alignment): Transfer bytes_to_boundary to reach 64B boundary
  Phase 2 (Streaming): Transfer full 64B beats
  Phase 3 (Final):     Transfer remaining < 64 bytes
```

2.3 Alignment Information Structure

```
typedef struct packed {
    logic [15:0] first_chunk_enables;    // Phase 1 chunk mask
    logic [15:0] streaming_chunk_enables; // Phase 2 (always 0xFFFF)
    logic [15:0] final_chunk_enables;     // Phase 3 chunk mask
    logic [31:0] first_transfer_bytes;    // Phase 1 size
    logic [31:0] streaming_bytes;         // Phase 2 size
    logic [31:0] final_transfer_bytes;    // Phase 3 size
    logic [3:0]  first_start_chunk;       // Starting chunk (0-15)
    logic [3:0]  first_valid_chunks;      // Chunks in phase 1
    logic [3:0]  final_valid_chunks;      // Chunks in phase 3
} alignment_info_t;
```

2.4 AXI Translation

Transfer Phase	AxSIZE	AxBURST	Address Increment
Alignment	3'b010	INCR	4 bytes × (AxLEN+1)
Streaming	3'b110	INCR	64 bytes × (AxLEN+1)
Final	3'b010	INCR	4 bytes × (AxLEN+1)

3. AXI Protocol Burst Modes

The AXI protocol defines three burst types that constrain address generation:

3.1 FIXED Burst (AxBURST = 2'b00)

```
Address Formula:
  beat[n].addr = AxADDR   (constant for all beats)
```

Use Cases: - FIFO access (single address register) - Memory-mapped I/O ports - Audio codec sample registers

Constraints: - All beats transfer to same address - Cannot cross 4KB boundary (trivially satisfied)

3.2 INCREMENT Burst (AxBURST = 2'b01)

```
Address Formula:
  beat[n].addr = AxADDR + n × (1 << AxSIZE)

  Example (AxSIZE=6, 64-byte):
    beat[0] = 0x1000
    beat[1] = 0x1040
    beat[2] = 0x1080
    ...
```

Use Cases: - Linear memory access - Block copy operations - Stream data to/from memory

Constraints: - Cannot cross 4KB boundary - Maximum burst length varies by protocol version

3.3 WRAP Burst (AxBURST = 2'b10)

```
Address Formula:
  wrap_boundary = AxADDR & ~((AxLEN+1) × (1<<AxSIZE) - 1)
  wrap_size = (AxLEN+1) × (1 << AxSIZE)

 beat[n].addr = wrap_boundary + ((AxADDR + n×(1<<AxSIZE)) mod wrap_size)

  Example (AxADDR=0x1020, AxLEN=3, AxSIZE=4):
    Wrap boundary: 0x1000
    Wrap size: 64 bytes (4 × 16)
    beat[0] = 0x1020
    beat[1] = 0x1030
    beat[2] = 0x1000  (wrapped)
    beat[3] = 0x1010
```

Use Cases: - Cache line fills (critical-word-first) - Circular buffer access

Constraints: - AxLEN must be 1, 3, 7, or 15 - Starting address must be aligned to transfer size

---

4. Linear Address Increment Patterns

4.1 Simple Contiguous

The most basic pattern - sequential addresses:

```
Descriptor Fields:
  src_addr:  Starting source address
  dst_addr:  Starting destination address
  length:    Total transfer size

Address Calculation:
  for each beat b in [0, length/beat_size):
    src = src_addr + b × beat_size
    dst = dst_addr + b × beat_size
```

RAPIDS Implementation: Native support via beat-based transfers.

4.2 Unaligned Start/End

Handling transfers that don't start or end on natural boundaries:

```
Example: Transfer 200 bytes from 0x1040
 Phase 1 (Align): Transfer 64 bytes (0x1040-0x107F) to reach 0x1080
 Phase 2 (Stream): Transfer 128 bytes (0x1080-0x10FF) in 2 full beats
  Phase 3 (Final):  Transfer 8 bytes (0x1100-0x1107)
```

RAPIDS Implementation: Native support via alignment_info_t structure.

4.3 Size-Constrained Bursts

Breaking large transfers into AXI-compliant bursts:

```
Constraints:
  - AXI4: Max 256 beats per burst
  - No 4KB boundary crossing

Address Calculation:
  remaining = total_length
  current_addr = start_addr
```

```
  while (remaining > 0):
    burst_len = min(remaining, 256×beat_size)
         burst_len  =  min(burst_len,  next_4kb_boundary  -
current_addr)
    issue_burst(current_addr, burst_len)
    current_addr += burst_len
    remaining -= burst_len
```

RAPIDS Implementation: Handled by AXI engine burst splitting.

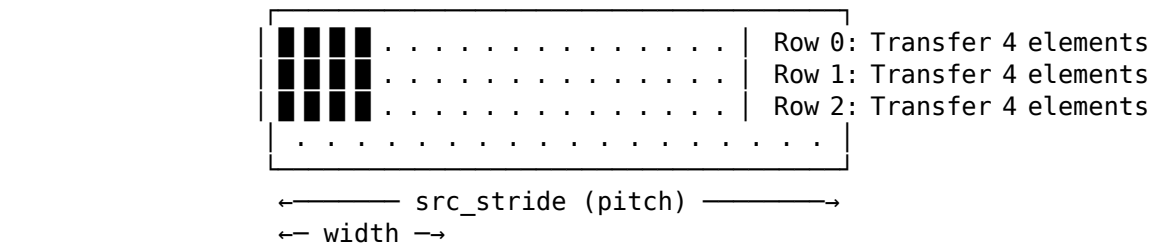
---

5. 2D Transfer Patterns (Stride-Based)

5.1 Concept

2D transfers move rectangular regions where source and destination may have different memory layouts (pitches/strides).

Memory Layout:



5.2 Descriptor Fields (Extended)

```
2D Descriptor Format:
  src_addr:    Starting source address (top-left corner)
  dst_addr:    Starting destination address
  x_length:    Bytes per row (width of transfer region)
  y_length:    Number of rows
  src_stride:  Source bytes between row starts
  dst_stride:  Destination bytes between row starts
```

5.3 Address Calculation

```
for row in [0, y_length):
  row_src = src_addr + row × src_stride
  row_dst = dst_addr + row × dst_stride
  transfer(row_src, row_dst, x_length)
```

5.4 Use Cases

Image Processing:

```
Example: Extract 640×480 ROI from 1920×1080 frame (32-
bit pixels)
  src_addr:   frame_base + (y_offset × 1920 + x_offset) × 4
  dst_addr:   roi_buffer
  x_length:   640 × 4 = 2560 bytes
  y_length:   480
  src_stride: 1920 × 4 = 7680 bytes
  dst_stride: 640 × 4 = 2560 bytes (packed destination)
```

Video Frame Tiling:

```
Example: Copy 16×16 macroblock
  x_length:   16 × bytes_per_pixel
  y_length:   16
  src_stride: frame_width × bytes_per_pixel
  dst_stride: 16 × bytes_per_pixel
```

5.5 Analog Devices AXI DMAC Implementation

Reference implementation from ADI's high-speed DMA controller:

```
Registers:
  X_LENGTH  (0x418): Row width - 1
  Y_LENGTH  (0x41C): Row count - 1
  SRC_STRIDE (0x424): Source row spacing
  DEST_STRIDE (0x420): Destination row spacing

Address Formulas:
  ROW_SRC_ADDRESS = SRC_ADDRESS + SRC_STRIDE × N
  ROW_DEST_ADDRESS = DEST_ADDRESS + DEST_STRIDE × N
```

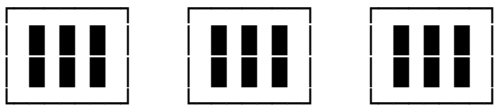
---

6. 3D Transfer Patterns

6.1 Concept

3D transfers extend 2D with an additional depth/plane dimension:

```
Memory Layout (Z planes stacked):
  Plane 0:    Plane 1:    Plane 2:
```



6.2 Descriptor Fields (Extended)

```
3D Descriptor Format:
  src_addr:     Starting address (origin corner)
  dst_addr:     Destination address
  x_length:     Width in bytes
  y_length:     Height in rows
  z_length:     Depth in planes
  src_stride_x: Source row stride (unused, implicit x_length)
  src_stride_y: Source row-to-row stride
  src_stride_z: Source plane-to-plane stride
  dst_stride_y: Destination row stride
  dst_stride_z: Destination plane stride
```

6.3 Address Calculation

```
for plane in [0, z_length):
  for row in [0, y_length):
    src = src_addr + plane×src_stride_z + row×src_stride_y
    dst = dst_addr + plane×dst_stride_z + row×dst_stride_y
    transfer(src, dst, x_length)
```

6.4 Use Cases

3D Medical Imaging:

```
Example: Extract 64×64×64 VOI from 512×512×256 CT volume
  src_addr:    volume_base + z_off×512×512 + y_off×512 + x_off
  x_length:     64 bytes
  y_length:     64
  z_length:     64
  src_stride_y: 512
  src_stride_z: 512 × 512
```

Neural Network Tensors:

```
Example: Move 4D tensor slice [batch, channel, H, W]
  Requires nested 3D operations or generalized tensor addressing
```

---

7. Scatter-Gather Patterns

7.1 Concept

Scatter-Gather (SG) handles non-contiguous memory regions using linked lists of descriptors:

```
Descriptor Chain:
  ┌──────────┐     ┌──────────┐     ┌──────────┐
  │ Desc 0   │────→│ Desc 1   │────→│ Desc 2   │────→ NULL
  │ addr=A0  │     │ addr=A1  │     │ addr=A2  │
  │ len=L0   │     │ len=L1   │     │ len=L2   │
  └──────────┘     └──────────┘     └──────────┘
```

7.2 SG Descriptor Fields

```
Scatter-Gather Descriptor:
  buffer_addr:      Address of this fragment
  buffer_length:    Size of this fragment
  next_desc_ptr:    Address of next descriptor (0 = last)
  control_flags:   Completion interrupt, error handling, etc.
  status:           Completion status (written by DMA)
```

7.3 Operation Modes

Gather (Multiple Source -> Single Destination):

```
Source fragments: [A0:L0], [A1:L1], [A2:L2]
Destination: Contiguous buffer B

Result: B = concat(mem[A0:A0+L0], mem[A1:A1+L1], mem[A2:A2+L2])
```

Scatter (Single Source -> Multiple Destinations):

```
Source: Contiguous buffer B of length L0+L1+L2
Destination fragments: [A0:L0], [A1:L1], [A2:L2]

Result: mem[A0:A0+L0] = B[0:L0]
        mem[A1:A1+L1] = B[L0:L0+L1]
        mem[A2:A2+L2] = B[L0+L1:end]
```

7.4 AMD/Xilinx AXI DMA SG Descriptor Format

```
Standard Descriptor (32-bit addressing):
  Offset  Field            Description
  ──────────────────────────────────────────────────
  0x00    NXTDESC          Next descriptor pointer
  0x04    Reserved
  0x08    BUFFER_ADDRESS   Data buffer pointer
  0x0C    Reserved
  0x10    Reserved
  0x14    Reserved
  0x18    CONTROL          Transfer length, SOF, EOF
  0x1C    STATUS           Completion status
```

7.5 Use Cases

Network Packet Assembly:

```
Example: Assemble TCP packet from scattered headers and payload
  Desc 0: Ethernet header (14 bytes at addr_eth)
  Desc 1: IP header (20 bytes at addr_ip)
  Desc 2: TCP header (20 bytes at addr_tcp)
  Desc 3: Payload (1460 bytes at addr_payload)
```

RAPIDS Implementation: Native support via next_descriptor_ptr chaining.

---

8. Circular and Ring Buffer Patterns

8.1 Concept

Circular buffers wrap addresses at a boundary, enabling continuous streaming without descriptor reload:

```
Memory Layout:

  ┌─────────────────────────────────────┐
  │ 7 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | (wraps)
  └─────────────────────────────────────┘

     ↑               ↑
   write           read
```

8.2 Descriptor Fields (Extended)

```
Circular Buffer Descriptor:
  base_addr:    Buffer base address
 buffer_size:  Total buffer size (must be power of 2 for efficient wrap)
  current_ptr:  Current read/write position
  wrap_enable:  Enable address wrapping
```

8.3 Address Calculation

```
Efficient wrap (power-of-2 size):
 next_addr = base_addr + ((current_ptr + increment) & (buffer_size -
1))

General wrap:
  offset = (current_ptr + increment) % buffer_size
  next_addr = base_addr + offset
```

8.4 Ping-Pong Double Buffering

Common implementation using two descriptors pointing to each other:

```
Descriptor A:                Descriptor B:
  buffer_addr = BUF_A          buffer_addr = BUF_B
  next_desc = &Desc_B          next_desc = &Desc_A

Operation:
  While DMA processes BUF_A, CPU fills BUF_B
  On completion, DMA auto-loads Desc_B, starts BUF_B
  CPU processes BUF_A while DMA runs BUF_B
  (repeat)
```

8.5 Use Cases

Audio Streaming:

```
Example: 48kHz stereo audio, 20ms buffers
  buffer_size: 48000 × 2 channels × 2 bytes × 0.02s = 3840 bytes
  Ring with 4 buffers: 15360 bytes total
  Wrap at: base + 15360
```

Network RX Ring:

```
Example: Ethernet receive ring
  descriptor_count: 256 (power of 2)
  Each descriptor points to packet buffer
  Last descriptor.next points to first descriptor
  Hardware auto-advances through ring
```

---

9. Tensor Accelerator Memory Access Patterns (PRIORITY 0)

Tensor accelerators for neural network inference represent the most demanding and sophisticated address generation requirements. This section provides comprehensive coverage of industry-standard approaches.

9.1 Systolic Array Dataflow Fundamentals

Systolic arrays (used in Google TPU, NVIDIA Tensor Cores, etc.) require coordinated data movement across three matrices: inputs (activations), weights, and outputs (partial sums).

Three Primary Dataflows:

Dataflow	Stationary Data	Moving Data	Best For
Weight Stationary (WS)	Weights preloaded in PEs	Inputs + partial sums flow	Weight reuse (many inputs)
Input Stationary (IS)	Inputs held in PEs	Weights + partial sums flow	Input reuse (many kernels)
Output Stationary (OS)	Partial sums accumulate in PEs	Inputs + weights flow	Output accumulation

Address Pattern Implications:

```
Weight Stationary:
  - Weights: Load once per layer, stream through array
  - Inputs: Stream row-by-row, reuse across weight tiles
  - Address pattern: 2D tiled with weight-tile-aligned boundaries

Output Stationary:
  - Outputs: Accumulate in place
  - Inputs/Weights: Double-buffered streaming
  - Address pattern: Output-tile-centric addressing
```

9.2 Data Cube Model (NVDLA Reference)

NVIDIA's Deep Learning Accelerator (NVDLA) defines tensor addressing using a "data cube" model with three key stride parameters:

Data Cube Structure:

```
Dimensions: Width (W) × Height (H) × Channels (C)

Memory Layout:
  - Data divided into 1×1×32byte "atom cubes"
  - Scanning order: C'(32B) → W → H → C (surfaces)
  - C' changes fastest (innermost loop)

Address Calculation:
  element_addr = base_addr
                 + surface_index × surface_stride
                 + line_index × line_stride
                 + element_offset_within_line
```

Stride Parameters:

Parameter	Description	Alignment
`line_stride`	Bytes from line N to line N+1	32 bytes
`surface_stride`	Bytes from surface N to surface N+1	32 bytes
`base_addr`	Starting address	32 bytes

NVDLA Register Configuration:

```
D_DATAIN_FORMAT     : Data format selection
D_DATAIN_SIZE_0     : Width × Height
D_DATAIN_SIZE_1     : Channels
D_LINE_STRIDE       : Line-to-line spacing (0x5040)
D_SURF_STRIDE       : Surface-to-surface spacing

Example:
  Width=26, Height=32, Channels=3
  line_stride = 0x1A0 (aligned width)
  surface_stride = 0x1520 (line_stride × height)
```

9.3 Tensor Memory Accelerator (TMA) - NVIDIA Hopper

NVIDIA's Hopper architecture introduces hardware TMA for efficient multi-dimensional tensor transfers:

TMA Descriptor Format:

```
CUtensorMap structure (64-bit packed descriptor):
  - Global memory base address
  - Tensor rank (1D to 5D supported)
  - Dimension sizes and strides
  - Swizzle pattern for bank conflict avoidance
  - Box dimensions (tile size for transfer)
```

Key Constraints:

```
Alignment Requirements:
  - Contiguous dimension must have stride=1
  - All other strides must be multiples of 16 bytes
  - For float tensors [M,N] with stride [N,1]: N % 4 == 0

Tile Quantization:
  - TPU/GPU systolic arrays process 128×128 or 128×8 tiles
 - Tensors must align to tile boundaries to avoid padding waste
  - Performance penalty for partial tiles (idle MACs)
```

Address Generation (Hardware):

```
TMA eliminates per-thread address calculation:
 Traditional: Each thread computes addr = base + thread_id × stride
  TMA: Single descriptor, hardware generates all addresses

Coordinate-based access:
 ArithTuple(row, col) → Hardware translates to physical address
 Automatic out-of-bounds predication (no manual boundary checks)
```

9.4 Convolution Address Patterns

9.4.1 Direct Convolution

```
Input Feature Map:  [N, C_in, H_in, W_in]
Weight Kernel:      [C_out, C_in, K_h, K_w]
Output Feature Map: [N, C_out, H_out, W_out]

Output position (n, c_out, h_out, w_out):
  For each (c_in, k_h, k_w):
    input_addr = base_in + n×(C_in×H_in×W_in)
                         + c_in×(H_in×W_in)
                         + (h_out×stride_h + k_h)×W_in
                         + (w_out×stride_w + k_w)

    weight_addr = base_w + c_out×(C_in×K_h×K_w)
                         + c_in×(K_h×K_w)
                         + k_h×K_w
                         + k_w
```

9.4.2 Im2Col Transformation Converts convolution to GEMM for systolic array efficiency:

```
Original Convolution:
  Output[h,w] = sum(Input[h+kh, w+kw] × Weight[kh, kw])

Im2Col Transformation:
  - Unroll each receptive field into a column
  - Weight kernels become rows
  - Convolution becomes matrix multiplication

Address Pattern for Im2Col:
  For output position (h_out, w_out):
    col_index = h_out × W_out + w_out
    For kernel position (c_in, k_h, k_w):
      row_index = c_in × K_h × K_w + k_h × K_w + k_w
    src_addr = base + c_in×(H×W) + (h_out×s_h+k_h)×W + (w_out×s_w+k_w)
    dst_addr = im2col_base + row_index × num_output_positions + col_index
```

```
Data Expansion Factor:
  Naive: K_h × K_w × data replication
 Hardware Im2Col: Generate addresses on-the-fly, no physical expansion
```

9.4.3 Strided and Dilated Convolution

```
Strided Convolution (stride > 1):
  input_h = output_h × stride + kernel_h
  input_w = output_w × stride + kernel_w
  Reduces output spatial dimensions

Dilated/Atrous Convolution:
  input_h = output_h + kernel_h × dilation
  input_w = output_w + kernel_w × dilation
  Expands receptive field without increasing parameters

Address Modification:
  base_addr + (h × stride + kh × dilation) × line_stride
            + (w × stride + kw × dilation) × element_size
```

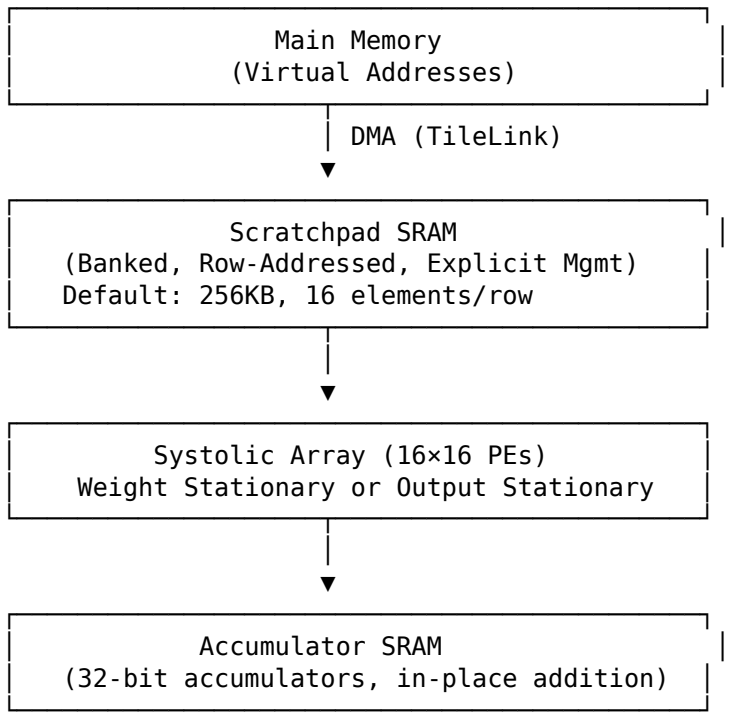
9.4.4 Depthwise Separable Convolution

```
Standard Convolution: C_in × C_out × K × K multiplications
Depthwise Separable: C_in × K × K + C_in × C_out multiplications

Phase 1 - Depthwise (per-channel):
  For each channel c:
    output[c, h, w] = conv2d(input[c], kernel[c])

  Address Pattern:
    Each channel processes independently
    No cross-channel data movement
    input_addr = base + c × H × W + h × W + w

Phase 2 - Pointwise (1×1 convolution):
  Standard GEMM across channels
  input_addr = base + c × (H × W) + position
```

9.5 Gemmini Accelerator Architecture (UC Berkeley Reference)

Open-source systolic array generator with well-documented DMA and addressing:

Memory Hierarchy:



Scratchpad Address Format (32-bit):

```
Bit 31:    0=Scratchpad, 1=Accumulator
Bit 30:    Accumulator mode (0=overwrite, 1=accumulate)
Bit 29:    Read format (0=scaled inputType, 1=raw accType)
Bits 28:0: Row address within selected memory

Row Width:
  Scratchpad: DIM × inputType bits (default: 16 × 8 = 128 bits)
  Accumulator: DIM × accType bits (default: 16 × 32 = 512 bits)
```

DMA Operations:

```
// mvin: Move data from DRAM to scratchpad
// Configuration registers set stride
mvin(
    dram_addr,          // Virtual address (64-bit)
    sp_addr,            // Scratchpad row address
    rows,               // Number of rows to transfer
    cols                // Elements per row (≤ DIM)
);

// mvout: Move data from accumulator to DRAM
// Supports integrated max-pooling
```

```
mvout(
    dram_addr,          // Destination address
    acc_addr,           // Accumulator row address
    rows, cols,
    pool_size,          // Optional pooling window
    pool_stride
);
```

Stride Configuration:

```
config_mvin:
  stride[0]: Source memory stride for mvin_A
  stride[1]: Source memory stride for mvin_B
  stride[2]: Source memory stride for mvin_D

config_mvout:
  stride: Destination memory stride
  pool_params: Pooling window and stride
```

9.6 Tiling for Large Matrices

When matrices exceed on-chip memory, tiling is essential:

Hierarchical Tiling (CUTLASS Model):

```
Level 1 - Thread Block Tile:
  Load tile from global memory to shared memory
  Tile size: M_tile × K_tile (A), K_tile × N_tile (B)

Level 2 - Warp Tile:
  Load from shared memory to registers
  Tile size: Warp_M × Warp_K (A), Warp_K × Warp_N (B)

Level 3 - Thread Tile:
  Individual thread's computation
  Register-to-register operations

Address Generation per Level:
  Global: base + block_id × block_tile_size
  Shared: smem_base + warp_id × warp_tile_size
  Register: Direct indexing within tile
```

Double-Buffering for Latency Hiding:

```
Buffer Structure:
  tile_buffer[2][TILE_ROWS][TILE_COLS]

Pipeline:
```

```
  Cycle N:   Compute on buffer[0], Load to buffer[1]
  Cycle N+1: Compute on buffer[1], Load to buffer[0]

Address Pattern:
  compute_addr = base + (cycle % 2) × tile_size + element_offset
  load_addr   = base + ((cycle + 1) % 2) × tile_size + element_offset
```

9.7 Sparse Tensor Patterns

Sparse accelerators require index-based indirect addressing:

Compressed Sparse Row (CSR) Format:

```
Dense: [a 0 b]    CSR: values = [a, b, c, d]
       [0 c 0]         col_idx = [0, 2, 1, 0]
       [d 0 0]         row_ptr = [0, 2, 3, 4]

Address Calculation:
  For row r, iterate col_idx[row_ptr[r]] to col_idx[row_ptr[r+1]-
1]
   value_addr = values_base + idx
   col_addr = col_idx_base + idx

Indirect Access:
   actual_col = memory[col_addr]
   dense_addr = base + r × stride + actual_col × element_size
```

Hardware Implications:

```
Dense Accelerator (TPU): Initiation interval = 1 (fully pipelined)
Sparse Accelerator (EIE): Initiation interval = 17 (metadata overhead)

Trade-off:
   - Sparse saves memory bandwidth (skip zeros)
   - But adds index indirection latency
   - Effective for >90% sparsity
```

9.8 Transformer/Attention Patterns

Modern LLM accelerators require specialized patterns:

Q, K, V Matrix Generation:

```
Input: X [seq_len, d_model]
Weights: W_Q, W_K, W_V [d_model, d_k]

Q = X × W_Q  →  [seq_len, d_k]
K = X × W_K  →  [seq_len, d_k]
```

```
V = X × W_V  →  [seq_len, d_k]

Address Pattern: Standard GEMM with different weight matrices
  q_addr = q_base + seq_idx × d_k + head_idx
  k_addr = k_base + seq_idx × d_k + head_idx
  v_addr = v_base + seq_idx × d_k + head_idx
```

Attention Score Computation:

```
Attention = softmax(Q × K^T / sqrt(d_k)) × V

Memory Challenge:
  Q × K^T produces [seq_len × seq_len] matrix
  O(n^2) memory for sequence length n

FlashAttention Tiling:
  Tile Q, K, V into blocks that fit in SRAM
 Compute attention incrementally without materializing full matrix

  Tile Address Pattern:
    q_tile_addr = q_base + tile_row × tile_size × d_k
    k_tile_addr = k_base + tile_col × tile_size × d_k

  Online Softmax:
    Track running max and sum across tiles
    No need to store intermediate attention matrix
```

9.9 Programmable Address Generation Unit (PAGU)

Academic reference for flexible tensor addressing:

PAGU Instruction Set:

```
Operations Supported:
  - Standard Convolution
  - Padded Convolution
  - Strided Convolution
  - Dilated (Atrous) Convolution
  - Pooling (Max, Average)
  - Upsampled Convolution
  - Transposed Convolution

Performance: 1.7 cycles per address for convolution
Area Overhead: ~4.6× vs fixed datapath (justified by flexibility)
```

Configuration Registers:

```
tensor_config_t:
```

```
  base_addr[63:0]           // Tensor base address
  dim_sizes[4][31:0]        // Size of each dimension
  dim_strides[4][31:0]      // Stride of each dimension
  kernel_size[15:0]         // Convolution kernel size
  conv_stride[7:0]          // Convolution stride
  dilation[7:0]             // Dilation factor
  padding[7:0]              // Padding amount
  operation_mode[3:0]       // Select operation type
```

Address Generation FSM:

```
always_ff @(posedge clk) begin
    case (state)
        IDLE: begin
            if (start) begin
                dim_counters <= '0;
                state <= GENERATE;
            end
        end

        GENERATE: begin
            // Multi-dimensional address calculation
            addr <= base_addr;
            for (int d = 0; d < num_dims; d++) begin
                case (operation_mode)
                    CONV_STANDARD:
                        addr <= addr + dim_counters[d] * dim_strides[d];
                    CONV_STRIDED:
                        addr <= addr + (dim_counters[d] * conv_stride) * dim_strides
                    CONV_DILATED:
                        addr <= addr + (dim_counters[d] * dilation) * dim_strides[d]
                endcase
            end

            // Increment counters (innermost first)
            increment_counters();

            if (done) state <= IDLE;
        end
    endcase
end
```

---

10. Implementation Comparison Matrix

10.1 Pattern Complexity vs. Capability

Pattern	Descriptor Fields	Address Logic	Hardware Cost	SW Flexibility
Linear	3 (src, dst, len)	Simple add	Low	Limited
2D Stride	6 (+x_len, y_len, strides)	Nested loop	Medium	Good
3D Volume	9 (+z_len, z_strides)	Triple nested	Medium-High	Very Good
Scatter-Gather	4 per fragment	List traversal	Medium	Excellent
Circular	4 (+size, wrap)	Modulo/mask	Low-Medium	Good
Tensor	N (variable stride)	Sum-of-products	High	Excellent

10.2 Use Case Mapping

Application Domain	Primary Pattern	Secondary Pattern
Block memory copy	Linear	-
Network packets	Scatter-Gather	Linear
Audio streaming	Circular	Linear
Image processing	2D Stride	Linear
Video encode/decode	2D Stride	Scatter-Gather
3D graphics	3D Volume	2D Stride
Neural networks	Tensor	2D Stride
Database operations	Scatter-Gather	Linear

10.3 RAPIDS Current vs. Extended Capability

Capability	Current RAPIDS	Industry Standard
Linear contiguous	YES	YES
Unaligned transfers	YES (3-phase)	Varies
Descriptor chaining	YES	YES
2D stride	NO	Common
3D volume	NO	Rare
Scatter-Gather	Partial (chain)	Full SG
Circular wrap	NO	Common
Tensor addressing	NO	Emerging

---

11. RAPIDS Extension Recommendations

11.0 PRIORITY 0: Tensor Accelerator Addressing (CRITICAL)

Rationale: Essential for AI/ML workloads. Tensor accelerators represent the highest-growth market for DMA engines. Without tensor addressing, RAPIDS cannot serve neural network inference applications.

Implementation Approach: Data Cube Model (NVDLA-style)

This approach provides the best balance of flexibility and hardware complexity, supporting convolution, pooling, and GEMM operations.

Extended Descriptor Format:

```
typedef struct packed {
    // === Base Fields (existing, 256 bits) ===
    logic [63:0]  src_addr;           // Base source address
    logic [63:0]  dst_addr;           // Base destination address
    logic [31:0]  length;             // For linear mode: beats
    logic [31:0]  next_descriptor_ptr;
    logic [7:0]   desc_priority;
    logic [3:0]   channel_id;
    logic         error;
    logic         last;
    logic         gen_irq;
    logic         valid;

    // === Tensor Extension (repurpose reserved bits + add 2nd descriptor word) ===
    // Mode selection
    logic [3:0]   transfer_mode;      // 0=linear, 1=2D, 2=3D/tensor, 3=conv, 4=pool

    // Data cube dimensions
    logic [15:0]  width;              // W dimension (elements)
    logic [15:0]  height;             // H dimension (lines)
    logic [15:0]  channels;           // C dimension (surfaces)

    // Stride parameters (32-byte aligned)
    logic [31:0]  line_stride;        // Bytes between lines
    logic [31:0]  surface_stride;     // Bytes between surfaces

    // Convolution parameters (when transfer_mode == CONV)
    logic [7:0]   kernel_width;       // Kw
    logic [7:0]   kernel_height;      // Kh
    logic [7:0]   conv_stride_x;      // Horizontal stride
    logic [7:0]   conv_stride_y;      // Vertical stride
    logic [7:0]   dilation_x;         // Horizontal dilation
```

```
    logic [7:0]   dilation_y;         // Vertical dilation
    logic [7:0]   pad_left;           // Left padding
    logic [7:0]   pad_right;          // Right padding
    logic [7:0]   pad_top;            // Top padding
    logic [7:0]   pad_bottom;         // Bottom padding

    // Pooling parameters (when transfer_mode == POOL)
    logic [7:0]   pool_width;
    logic [7:0]   pool_height;
    logic [7:0]   pool_stride_x;
    logic [7:0]   pool_stride_y;
    logic [1:0]   pool_type;          // 0=max, 1=avg, 2=min

    // Data type and precision
    logic [3:0]   data_type;          // 0=int8, 1=int16, 2=fp16, 3=bf16, 4=fp32
    logic [3:0]   element_size;       // Bytes per element (1, 2, 4)
} descriptor_tensor_t;
```

Address Generation Unit (AGU) Architecture:

```
module tensor_address_generator #(
    parameter int PIPE_STAGES = 3     // Pipeline for multiply-accumulate
) (
    input  logic        clk,
    input  logic        rst_n,

    // Configuration from descriptor
    input  descriptor_tensor_t desc,
    input  logic        start,

    // Address output stream
    output logic [63:0] addr,
    output logic        addr_valid,
    input  logic        addr_ready,

    // Status
    output logic        done
);

    // Dimension counters
    logic [15:0] w_cnt, h_cnt, c_cnt;
    logic [7:0]  kw_cnt, kh_cnt;       // Kernel position (for conv)

    // Pipeline registers for address calculation
    logic [63:0] addr_stage[PIPE_STAGES];

    // FSM states
```

```
typedef enum logic [2:0] {
    IDLE,
    CALC_BASE,
    CALC_LINE,
    CALC_SURFACE,
    OUTPUT
} state_t;
state_t state;

// Address calculation based on mode
always_ff @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        state <= IDLE;
        w_cnt <= '0;
        h_cnt <= '0;
        c_cnt <= '0;
    end else begin
        case (state)
            IDLE: begin
                if (start) begin
                    w_cnt <= '0;
                    h_cnt <= '0;
                    c_cnt <= '0;
                    state <= CALC_BASE;
                end
            end

            CALC_BASE: begin
                // Stage 1: Base address
                addr_stage[0] <= desc.src_addr;
                state <= CALC_LINE;
            end

            CALC_LINE: begin
                // Stage 2: Add line offset
                case (desc.transfer_mode)
                    MODE_LINEAR:
                        addr_stage[1] <= addr_stage[0] + (w_cnt * desc.element_s

                    MODE_2D, MODE_3D:
                        addr_stage[1] <= addr_stage[0]
                                       + (h_cnt * desc.line_stride)
                                       + (w_cnt * desc.element_size);

                    MODE_CONV:
                        // Convolution: account for stride and kernel position
```

```
                addr_stage[1] <= addr_stage[0]
                                + ((h_cnt * desc.conv_stride_y + kh_cnt *
                                + ((w_cnt * desc.conv_stride_x + kw_cnt *

            MODE_POOL:
                addr_stage[1] <= addr_stage[0]
                                + ((h_cnt * desc.pool_stride_y) * desc.li
                                + ((w_cnt * desc.pool_stride_x) * desc.el
        endcase
        state <= CALC_SURFACE;
    end

    CALC_SURFACE: begin
        // Stage 3: Add surface offset (for 3D/tensor)
        if (desc.transfer_mode inside {MODE_3D, MODE_CONV}) begin
            addr_stage[2] <= addr_stage[1] + (c_cnt * desc.surface_strid
        end else begin
            addr_stage[2] <= addr_stage[1];
        end
        state <= OUTPUT;
    end

    OUTPUT: begin
        if (addr_ready) begin
            // Increment counters (innermost first)
            if (desc.transfer_mode == MODE_CONV) begin
                // Convolution: iterate kernel, then spatial, then chann
                if (kw_cnt < desc.kernel_width - 1) begin
                    kw_cnt <= kw_cnt + 1;
                end else begin
                    kw_cnt <= '0;
                    if (kh_cnt < desc.kernel_height - 1) begin
                        kh_cnt <= kh_cnt + 1;
                    end else begin
                        kh_cnt <= '0;
                        // Continue to spatial/channel iteration
                        increment_spatial_counters();
                    end
                end
            end else begin
                increment_spatial_counters();
            end

            if (transfer_complete())
                state <= IDLE;
            else
```

```
                            state <= CALC_BASE;
                        end
                    end
                endcase
            end
        end

        assign addr = addr_stage[PIPE_STAGES-1];
        assign addr_valid = (state == OUTPUT);
        assign done = (state == IDLE) && !start;

endmodule
```

Scratchpad Integration (Gemmini-style):

```
// Memory address format for tensor scratchpad
typedef struct packed {
    logic        mem_select;       // 0=scratchpad, 1=accumulator
    logic        acc_mode;         // 0=overwrite, 1=accumulate
    logic        read_format;      // 0=scaled, 1=raw
    logic [28:0] row_addr;         // Row address within memory
} scratchpad_addr_t;

// DMA configuration for tensor loads
typedef struct packed {
    logic [63:0] dram_addr;        // Virtual address in main memory
    scratchpad_addr_t sp_addr;     // Scratchpad destination
    logic [15:0] rows;             // Number of rows to transfer
    logic [15:0] cols;             // Elements per row
    logic [31:0] dram_stride;      // Source memory stride
    logic [15:0] sp_stride;        // Scratchpad row stride (usually 1)
} tensor_dma_config_t;
```

Alignment Requirements:

```
All tensor addresses must satisfy:
  - base_addr aligned to 32 bytes (NVDLA requirement)
  - line_stride aligned to 32 bytes
  - surface_stride aligned to 32 bytes
 - For optimal performance: align to 64 bytes (RAPIDS beat size)

Verification:
  assert((desc.src_addr & 32'h1F) == 0);
  assert((desc.line_stride & 32'h1F) == 0);
  assert((desc.surface_stride & 32'h1F) == 0);
```

Performance Considerations:

```
Address Generation Rate:
```

```
  - Target: 1 address per cycle (pipelined)
  - Convolution: 1.7 cycles/address typical (due to kernel iteration)

Memory Bandwidth:
  - TPU-class: 600+ GB/s (HBM)
  - FPGA-class: 20-80 GB/s (DDR4/5)
  - RAPIDS target: Match AXI interface bandwidth

Tiling Support:
  - Hardware computes tile boundaries
  - Double-buffering via ping-pong descriptors
  - Software provides tile dimensions, hardware handles addressing
```

11.1 Priority 1: 2D Stride Support

Rationale: High value for image/video processing with moderate complexity. Also serves as foundation for tensor addressing.

Descriptor Extension:

```
typedef struct packed {
    // Existing fields (256 bits)...

    // 2D extension (96 bits in reserved space)
    logic [31:0] x_length;        // Bytes per row
    logic [15:0] y_length;        // Number of rows
    logic [31:0] src_stride;      // Source row stride
    logic [31:0] dst_stride;      // Destination row stride
    logic        mode_2d;         // Enable 2D mode
} descriptor_2d_t;
```

Address Generation:

```
always_comb begin
    if (mode_2d) begin
        row_src_addr = src_addr + current_row * src_stride;
        row_dst_addr = dst_addr + current_row * dst_stride;
        row_length   = x_length;
    end else begin
        row_src_addr = src_addr;
        row_dst_addr = dst_addr;
        row_length   = length * BYTES_PER_BEAT;
    end
end
```

11.2 Priority 2: Circular Buffer Mode

Rationale: Essential for streaming applications, low complexity addition.

Descriptor Extension:

```
typedef struct packed {
    // Existing fields...

    // Circular buffer extension
    logic [31:0] buffer_size;     // Total buffer size
    logic [31:0] buffer_mask;     // Size-1 for power-of-2
    logic        circular_mode;   // Enable wrap-around
    logic        src_circular;    // Apply to source
    logic        dst_circular;    // Apply to destination
} descriptor_circular_t;
```

Address Generation:

```
function automatic logic [63:0] wrap_address(
    logic [63:0] base,
    logic [63:0] offset,
    logic [31:0] mask,
    logic        enable
);
    if (enable)
        return base + (offset & {32'b0, mask});
    else
        return base + offset;
endfunction
```

11.3 Priority 3: Enhanced Scatter-Gather

Rationale: Full SG capability for network and fragmented memory use cases.

Current Limitation: Chain-only SG requires one descriptor per fragment.

Enhancement: Add fragment list support within single descriptor:

```
typedef struct packed {
    logic [63:0] fragment_list_ptr;  // Pointer to fragment array
    logic [15:0] fragment_count;     // Number of fragments
    logic        sg_mode;            // Enable SG mode

    // Fragment entry format (in memory):
    // [63:0]  address
```

```
    // [31:0]  length
    // [31:0]  flags
} descriptor_sg_t;
```

---

12. References

Industry Documentation

1. AMD AXI DMA Product Guide (PG021) - Scatter-Gather descriptor format
2. Analog Devices AXI DMAC - 2D transfer implementation
3. Understanding AXI Addressing (ZipCPU) - Burst mode analysis
4. WRAP Address Calculation (Verification Guide) - Wrap burst formulas

GPU and Accelerator Patterns

5. NVIDIA CUDA Memory Coalescing - Coalesced access patterns
6. CUDA Pitch Linear Memory - 2D array allocation
7. NVIDIA TMA Tutorial (CUTLASS) - Tensor Memory Accelerator
8. Efficient GEMM in CUDA (CUTLASS) - Hierarchical tiling

DMA Architecture

9. Intel DMA Descriptors - PCIe DMA linked lists
10. DMA330 Microcode - Linked-list implementation

Neural Network Accelerators

11. NVDLA Hardware Architecture - Data cube model, CDMA
12. NVDLA In-Memory Data Formats - Line stride, surface stride
13. NVDLA Unit Description - Convolution DMA details
14. Gemmini Accelerator (UC Berkeley) - Open-source systolic array
15. Gemmini Documentation (Chipyard) - DMA and scratchpad
16. Google TPU Architecture - Systolic array overview
17. TPU System Architecture - Memory hierarchy
18. Programmable AGU for DNNs - Flexible address generation

Convolution and Tensor Operations

19. Im2Col Characterization - Convolution memory patterns
20. ST Neural-ART Programming Model - NPU descriptor concepts

21. Systolic Array Dataflows Survey - Weight/input/output stationary
22. FlashAttention Analysis - Attention tiling patterns
23. Depthwise Separable Convolutions - Channel-wise addressing

Sparse Tensor Formats

24. FLAASH Sparse Tensor Accelerator - CSR/CSF formats
25. SparTen CNN Accelerator - Sparse tensor addressing
26. Indirection Stream Architecture - Hardware indirect addressing

---

Document History

Version	Date	Author	Changes
1.0	2026-01-13	RTL Design Sherpa	Initial comprehensive analysis

---

This document is part of the RAPIDS (Rapid AXI Programmable In-band Descriptor System) design documentation.