Instruction-Level Analysis of cuSPARSE `csrmv_v3_kernel` on H200

Date: 2026-02-28 **GPU:** NVIDIA H200 SXM (sm_90, Hopper) **CUDA:** 13.0.1 (Driver 570.86.15) **Library:** cuSPARSE 12.x (from `nvidia-cusparse-cu12`) **Companion report:** `spmv_h200_report.md`

---

1. Kernel Identification

1.1 Symbol Name

The kernel profiled by NCU is the `bool=false` template specialization:

```
_ZN8cusparse15csrmv_v3_kernelISt17integral_constantIbLb0EEllffffvEEvNS_12KernelCoeffsIT5_EE
PKT1_PKT0_S8_PKT2_S9_iiPKT3_PT4_PS6_PS4_
```

Demangled, this is:

```
cusparse::csrmv_v3_kernel<
    std::integral_constant<bool, false>,  // merge-path variant selector
    long long,                            // index type (INT64)
    long long,                            // offset type (INT64)
    float, float, float, float,           // alpha, beta, value, accumulator types
    void                                  // unused template param
>(
    cusparse::KernelCoeffs<float>,        // alpha/beta coefficients
    const long long*,                     // row_offsets (crow_indices)
    const long long*,                     // col_indices
    const long long*,                     // (unused or end_offsets)
    const float*,                         // values
    const float*,                         // (unused)
    int, int,                             // num_rows, num_cols
    const float*,                         // x vector
    float*,                               // y vector
    long long*,                           // workspace
    long long*                            // workspace
)
```

1.2 Invocation Chain

```
torch.sparse.mm(A_csr, x)
  → cusparseSpMV(handle, ..., CUSPARSE_SPMV_CSR_ALG2, ...)
    → csrmv_v3_kernel<integral_constant<bool,false>, int64, int64, float, ...>
```

PyTorch forces INT64 indices for all sparse tensors regardless of matrix dimensions, which is why this INT64 specialization is invoked.

1.3 Resource Usage (`cuobjdump -res-usage`)

Resource	`bool=false` (profiled)	`bool=true`
Registers per thread	**46**	55
Shared memory (bytes)	**1,816**	5,248
Constant memory (bytes)	**640**	640

NCU confirms: **46 registers/thread**, **792 bytes static shared memory** (the remaining shared memory is dynamically allocated at launch).

With 46 registers/thread, each SM can host at most `65536 / 46 = 1424` threads = **44.5 warps**. Combined with the 32-thread block size (1 warp per block), the **theoretical occupancy is 50%** (32 warps/SM), consistent with NCU's report.

1.4 Launch Configuration (NCU-observed)

Matrix	Grid Size	Block Size	Achieved Occupancy
cant	20,872	32 (1 warp)	44.0%
ldoor	242,305	32 (1 warp)	47.5%
cage15	516,665	32 (1 warp)	48.0%

1.5 SASS Extraction (Reproducibility)

```
module load cuda/13.0.1
cuobjdump -sass -arch sm_90 \
  -fun
'_ZN8cusparse15csrmv_v3_kernelISt17integral_constantIbLb0EEllffffvEEvNS_12KernelCoeffsIT5_E
EPKT1_PKT0_S8_PKT2_S9_iiPKT3_PT4_PS6_PS4_' \
  /storage/scratch1/6/jkim4112/envs/scaleai/lib/python3.13/site-packages/nvidia/cusparse/
lib/libcusparse.so.12
```

Full SASS: **`analysis/csrmv_v3_kernel_sm90_sass.txt`** (1,178 SASS instructions, addresses `0x0000` – `0x4990`).

---

2. Merge-Path Algorithm Overview

The `csrmv_v3_kernel` implements a **merge-path** parallelization of CSR SpMV, based on Merrill & Garland (2016). The key idea is to flatten the 2D row+nonzero iteration space into a 1D "merged coordinate" array of length `(num_rows + nnz)`, then partition it evenly across thread blocks.

2.1 Merged Coordinate Space

Each position in the merged space is either a "row step" (advance to next row) or a "nonzero step" (process next nonzero). A block of 32 threads (one warp) is assigned a contiguous tile of 192 elements (`0xc0 = 192`) from this merged space.

2.2 Three Kernel Phases

1. **Merge-path partition** — Binary search over `row_offsets[]` to find which rows and nonzeros belong to this block's tile. Row-boundary positions are stored to shared memory.
2. **Nonzero processing** — The hot inner loop: for each nonzero `j`, load `col_indices[j]`, compute the address `&x[col]`, load `x[col]` and `values[j]`, accumulate `val * x[col]`.
3. **Warp reduction & output** — `SHFL.DOWN` tree reduction to sum partial products across lanes, then atomic-add (`REDG`) to write final `y[i]` values.

2.3 Multiple Code Paths

The compiler generates three specialized code paths selected at runtime by the number of nonzeros per block:

Condition	Address Range	Description
≥ 65536 nnz/block	`0x0970` – `0x10f0`	Large tile: 4-byte shared memory entries, BSSY/BSYNC synchronization
256–65535 nnz/block	`0x1100` – `0x1850`	Medium tile: 2-byte shared memory entries (STS.U16)
33–255 nnz/block	`0x1860` – `0x1ef0`	Small tile: 1-byte shared memory entries (STS.U8)
≤ 32 nnz/block	`0x2110` – `0x2a70`	Tiny tile: warp-shuffle binary search, no shared memory

All paths converge at `0x1fb0` (sort boundaries) or `0x2a70` (barrier) before entering the nonzero processing loop.

---

3. Kernel Phase Breakdown with SASS

Phase 1: Setup & Parameter Load (0x0000–0x0090)

```
/*0000*/  LDC R1, c[0x0][0x28]            ; Load stack pointer from constant memory
/*0010*/  S2R R3, SR_CTAID.X              ; Read block index (blockIdx.x)
/*0020*/  IMAD.MOV.U32 R13, RZ, RZ, RZ   ; R13 = 0 (clear accumulator)
/*0030*/  ULDC.64 UR8, c[0x0][0x250]      ; Load 64-bit uniform param (total merged size)
/*0040*/  ULDC.64 UR6, c[0x0][0x208]      ; Load 64-bit uniform param (descriptor base)
/*0050*/  S2R R0, SR_TID.X               ; Read thread index (threadIdx.x)
/*0060*/  ULDC.U8 UR4, c[0x0][0x228]      ; Load 1-byte param (flag/config)
/*0070*/  S2R R12, SR_LANEID             ; Read warp lane ID
/*0080*/  IMAD R2, R3, 0x20, R0          ; R2 = blockIdx.x * 32 + threadIdx.x (global
thread ID)
```

Performance: Constant memory loads (`LDC` , `ULDC`) hit the constant cache (L1 const, ~4 cycles). `S2R` reads special registers (~8-20 cycles). This phase is **not on the critical path** — it executes once per block and is fully overlapped.

Phase 2: Tile Boundary Computation & Row-Pointer Loads (0x0090–0x08b0)

This is the **bulk setup phase** where the kernel:

1. Computes shared-memory offsets for 6 sub-tiles (each sub-tile = 32 elements of the merged space)
2. Performs bounds checking against the total merged-coordinate size
3. Loads 6 pairs of `row_offsets` entries via `LDG.E.EF.64` (64-bit, eviction-first policy)
4. Loads 6 `values[]` entries and 6 `x[]` entries via `LDG.E` / `LDG.E.EF`
5. Computes initial `FMUL` products

Representative SASS from this phase:

```
/*0280*/  @!P5 LDG.E.EF.64 R36, desc[UR6][R36.64]    ; Load row_offsets[pos] (INT64,
coalesced)
/*02d0*/  @!P4 LDG.E.EF.64 R16, desc[UR6][R16.64]    ; Load row_offsets[pos+1]
/*0310*/  @!P0 LDG.E.EF.64 R20, desc[UR6][R20.64]    ; Load row_offsets[pos+2]
...
/*0410*/  @!P5 LDG.E.EF R33, desc[UR6][R32.64]        ; Load values[j] (FP32)
/*04b0*/  @!P4 LDG.E.EF R43, desc[UR6][R42.64]        ; Load values[j+1]
...
/*05b0*/  @!P5 LDG.E R14, desc[UR6][R8.64]            ; Load x[col] (FP32, SCATTERED)
/*05d0*/  @!P2 LDG.E.EF R15, desc[UR6][R10.64]        ; Load x[col+1]
...
/*0780*/  @!P4 FMUL R43, R43, R2                       ; val * scaling_factor
/*0790*/  @!P4 FMUL R11, R43, R28                      ; partial_product = scaled_val *
x[col]
```

Key observation: The address computation for `x[col]` loads requires the column index to be available first. This is visible in the instruction scheduling — the `LEA` / `LEA.HI.X` instructions that compute `x_base + col * sizeof(float)` appear *between* the `LDG.E.EF.64` (column index load) and the `LDG.E` (x-vector load):

```
/*0430*/  @!P5 LEA R8, P6, R36, R8, 0x2               ; addr_lo = x_base + col_indices[j]
* 4
/*0440*/  @!P5 LEA.HI.X R9, R36, R9, R37, 0x2, P6     ; addr_hi (64-bit address calc)
...
/*05b0*/  @!P5 LDG.E R14, desc[UR6][R8.64]            ; x[col] — DEPENDENT on
col_indices[j]
```

This is the dependent load chain. The `LEA` at `0x0430` cannot execute until the `LDG.E.EF.64` at `0x0280` returns with `col_indices[j]` in R36. The subsequent `LDG.E` at `0x05b0` cannot issue until the `LEA` computes the address. **Two back-to-back DRAM accesses are serialized.**

Phase 3: Merge-Path Binary Search (0x0970–0x1ef0)

After the initial data is loaded, the kernel determines the exact row boundaries within each block's tile. This involves:

1. Loading pairs of `row_offsets` from global memory (`LDG.E.64`)
2. Comparing them to find where row transitions occur
3. Storing boundary positions to shared memory (`STS`, `STS.U16`, or `STS.U8` depending on tile size)
4. A loop (`BRA` backward) for the "sweep" variant that checks multiple positions

Representative from the large-tile path:

```
/*0ae0*/  LDG.E.64 R22, desc[UR6][R4.64]              ; row_offsets[search_pos]
/*0af0*/  LDG.E.64 R24, desc[UR6][R4.64+0x8]          ; row_offsets[search_pos + 1]
/*0b00*/  IMAD.IADD R12, R22, 0x1, -R2                 ; offset = row_start - tile_start
/*0b10*/  ISETP.EQ.U32.AND P6, PT, R22, R24, PT        ; same row? (boundary check)
/*0b20*/  ISETP.GT.U32.AND P0, PT, R12, 0xbf, PT       ; offset > 191? (out of tile)
/*0b30*/  ISETP.EQ.OR.EX P0, PT, R23, R25, P0, P6      ; combined 64-bit comparison
/*0b40*/  @!P0 IMAD R21, R12, 0x4, R13                 ; shared_mem_addr = offset * 4 +
base
/*0b50*/  @!P0 STS [R21], R0                            ; store boundary to shared memory
```

Performance impact: The `LDG.E.64` loads here trigger **Long Scoreboard stalls** as they access `row_offsets[]` from DRAM. However, this phase executes only once per block (not per-nonzero), so its contribution to total stall time is moderate compared to Phase 4.

Phase 4: Warp-Level Row Boundary Sort (0x1fb0–0x2a70)

After collecting row-boundary positions from shared memory, the kernel sorts them using a sequence of `VIMNMX` (vectorized integer min/max) and `SHFL.IDX` instructions:

```
/*1fb0*/  VIMNMX R7, R6, R7, !PT          ; R7 = max(R6, R7) — sorting network
/*1fd0*/  VIMNMX R8, R7, R4, !PT          ;
/*1ff0*/  VIMNMX R5, R8, R5, !PT          ;
/*2000*/  VIMNMX R2, R5, R2, !PT          ;
/*2010*/  VIMNMX R3, R2, R3, !PT          ;
...
/*2030*/  VOTE.ANY R0, PT, P0             ; Any lane has a boundary?
/*2050*/  FLO.U32 R0, R0                  ; Find first lane with boundary
/*2080*/  SHFL.IDX PT, R22, R3, R22, 0x1f ; Broadcast boundary to all lanes
```

This is followed by a binary-search refinement loop using `SHFL.IDX` at progressively finer granularity (stride 16, 8, 4, 2, 1):

```
/*2210*/  SHFL.IDX PT, R13, R4, 0x10, 0x1f   ; stride=16
/*2270*/  SHFL.IDX PT, R13, R4, R9, 0x1f     ; stride=8
/*22d0*/  SHFL.IDX PT, R21, R4, R7, 0x1f     ; stride=4
/*2330*/  SHFL.IDX PT, R21, R4, R13, 0x1f    ; stride=2
/*23b0*/  SHFL.IDX PT, R9, R4, R21, 0x1f     ; stride=1
```

After sorting, each lane knows its row assignment and the kernel proceeds to compute partial sums.

Performance: The `SHFL.IDX` instructions contribute to the **Short Scoreboard stalls** (10-12% of CPI in NCU data).

Phase 5: Nonzero Processing — THE CRITICAL HOT PATH (0x2b70–0x2cc0 / 0x47c0–0x4900)

This is the most performance-critical section. There are two variants: one for blocks that span **multiple rows** (0x2b70) and one for blocks within a **single row** (0x47c0). Both share the same fundamental dependency chain.

Multi-Row Path: Warp Reduction + REDG Output (0x2b70–0x2cc0)

The partial products computed in Phase 2 are accumulated and reduced:

```
; --- Accumulate partial sums across sub-tiles ---
/*2b80*/  FADD R11, R11, R16          ; accum[2] += accum[4]
/*2b90*/  FADD R10, R15, R10          ; accum[1] += accum[0]
/*2ba0*/  FADD R17, R17, R14          ; accum[3] += accum[5]
/*2bc0*/  FADD R10, R11, R10          ; combined = accum[2] + accum[1]
/*2be0*/  FADD R17, R10, R17          ; total = combined + accum[3]

; --- Warp-level tree reduction via SHFL.DOWN ---
/*2c00*/  SHFL.DOWN P0, R0, R17, 0x1, 0x1e01    ; stride=1:  get from lane+1
/*2c10*/  @P0 FADD R17, R17, R0                  ; accumulate
/*2c20*/  SHFL.DOWN P0, R0, R17, 0x2, 0x1c02    ; stride=2:  get from lane+2
/*2c30*/  @P0 FADD R17, R17, R0                  ; accumulate
/*2c40*/  SHFL.DOWN P0, R0, R17, 0x4, 0x1804    ; stride=4
/*2c50*/  @P0 FADD R17, R17, R0
/*2c60*/  SHFL.DOWN P0, R0, R17, 0x8, 0x1008    ; stride=8
/*2c70*/  @P0 FADD R17, R17, R0
/*2c80*/  SHFL.DOWN P0, R0, R17, 0x10, 0x10     ; stride=16
/*2c90*/  @P0 FADD R17, R17, R0

; --- Atomic output ---
/*2ca0*/  @!P0 EXIT                             ; non-leader lanes exit
/*2cb0*/  REDG.E.ADD.F32.FTZ.RN.STRONG.GPU desc[UR6][R2.64], R17    ; y[i] += sum
/*2cc0*/  EXIT
```

The `SHFL.DOWN` + `FADD` tree performs a 32-wide warp reduction in 5 steps ($\log_2(32) = 5$), accumulating the sum in lane 0. The final `REDG.E.ADD.F32` is an **atomic global reduction** — it atomically adds the warp's partial sum to `y[i]` in global memory.

Single-Row Path (0x47c0–0x4900)

When a block's entire tile falls within one row (no row boundaries), the kernel uses a simplified reduction:

```
/*47c0*/  FADD R16, R21, R16          ; accumulate sub-tile sums
/*47d0*/  FADD R33, R33, R28
/*47e0*/  FADD R25, R25, R14
/*4800*/  FADD R16, R16, R33
/*4810*/  FADD R25, R16, R25          ; total partial sum

; --- Same SHFL.DOWN tree reduction ---
/*4820*/  SHFL.DOWN P0, R0, R25, 0x1, 0x1e01
/*4850*/  @P0 FADD R25, R25, R0
/*4860*/  SHFL.DOWN P0, R0, R25, 0x2, 0x1c02
/*4870*/  @P0 FADD R25, R25, R0
/*4880*/  SHFL.DOWN P0, R0, R25, 0x4, 0x1804
/*4890*/  @P0 FADD R25, R25, R0
/*48a0*/  SHFL.DOWN P0, R0, R25, 0x8, 0x1008
/*48b0*/  @P0 FADD R25, R25, R0
/*48c0*/  SHFL.DOWN P0, R0, R25, 0x10, 0x10
/*48d0*/  @P0 FADD R25, R25, R0

/*48e0*/  @!P0 EXIT
/*48f0*/  REDG.E.ADD.F32.FTZ.RN.STRONG.GPU desc[UR6][R2.64], R25
/*4900*/  EXIT
```

Phase 6: Multi-Row Boundary Handling (0x2f80–0x46b0)

When a block spans multiple rows, the kernel must handle row boundaries — writing partial sums for completed rows and starting fresh accumulators for new ones. This is the most complex part of the kernel:

```
; --- Row transition detection ---
/*2f90*/  ISETP.NE.U32.AND P0, PT, R2, R3, PT     ; row[lane] != row[lane-1]?
/*2fd0*/  ISETP.NE.AND.EX P0, PT, R7, R4, PT, P0   ; (64-bit comparison)

; --- Cross-lane partial sum propagation ---
/*2fc0*/  SHFL.UP PT, R23, R3, 0x1, RZ             ; get row index from lane-1
/*30f0*/  SHFL.UP PT, R26, R4, 0x1, RZ             ; get row index from lane-1

; --- Conditional accumulation at row boundaries ---
/*3000*/  FSEL R26, R14, RZ, !P0                    ; zero out if new row
/*3030*/  FADD R15, R25, R26                        ; accumulate within row
/*3060*/  FSEL R17, R15, RZ, !P2                    ; select based on boundary

; --- Write completed rows to global memory (read-modify-write) ---
/*3df0*/  @!P0 LDG.E R26, desc[UR6][R20.64]        ; y[i] = read current
/*3ed0*/  @!P0 FADD R27, R33, R26                   ; y[i] += partial_sum
/*3ee0*/  @!P0 STG.E desc[UR6][R20.64], R27         ; write back

; --- Or use shared memory for intra-block reduction ---
/*3580*/  @P2 STS [R22], R25                        ; store partial to smem
/*3c00*/  @P4 LDS R5, [R22+0x4]                    ; load from smem
/*3c80*/  @P4 STG.E desc[UR6][R2.64+0x4], R5        ; write to y[]
```

The multi-row path uses a combination of: - `SHFL.UP` / `SHFL.DOWN` for cross-lane communication - `FSEL` for predicated selection (zero-out at row boundaries) - `VOTE.ANY` + `FLO.U32` for finding the first lane with a boundary - `STS` / `LDS` for shared-memory based reduction when rows span multiple boundaries - `STG.E` for writing final y[] values (read-modify-write pattern via `LDG.E` + `FADD` + `STG.E`)

---

4. The Dependency Chain — Why SpMV is Memory-Bound

4.1 The Critical Path

The fundamental bottleneck in CSR SpMV is the **dependent load chain** within the nonzero processing loop. For each nonzero element `j`, the kernel must:

```
LOAD 1:  col = col_indices[j]          ← DRAM access, ~300ns latency
         addr = x_base + col * 4       ← cannot start until LOAD 1 completes
LOAD 2:  xval = x[addr]               ← DRAM access, ~300ns latency (scattered)
LOAD 3:  val = values[j]              ← DRAM access (can overlap with LOAD 1)
COMPUTE: accum += val * xval           ← depends on both LOAD 2 and LOAD 3
```

In SASS, this manifests as:

```
LDG.E.EF.64  R36, desc[UR6][R36.64]   ← LOAD 1: col_indices[j] (INT64)
  ... (other instructions can execute here while waiting) ...
LEA           R8, P6, R36, R8, 0x2     ← ADDRESS CALC: dependent on R36 from LOAD 1
LEA.HI.X      R9, R36, R9, R37, 0x2   ← ADDRESS CALC: 64-bit upper half
  ... (nothing useful can execute until address is ready) ...
LDG.E         R14, desc[UR6][R8.64]    ← LOAD 2: x[col] (dependent on address)
  ... (must wait ~300ns for LOAD 2) ...
FMUL          R43, R43, R2             ← val * scaling (LOAD 3 was independent)
FMUL          R11, R43, R14            ← partial = scaled_val * x[col]
```

4.2 Pipeline Diagram

The critical path across two iterations shows why MLP is halved:

```
Time →→→→→→→→→→→→→→→→→→→→→→→→→→→→→→→→→→→→→→→→→→→→→→→→→→→→→→→→→→→→→→→→→→→→→→→→→→

Iter j:
  LDG col[j] ─────────────[~300ns DRAM]──────────────► LEA(addr) → LDG x[col]
──────────[~300ns DRAM]─────────► FMUL → FADD
  LDG val[j] ─────────────[~300ns
DRAM]───────────────────────────────────────────────────────────────────────────┘ (ready
long before needed)

Iter j+1:
  LDG col[j+1] ───────────[~300ns DRAM]──────────────► LEA(addr) → LDG x[col]
──────────[~300ns DRAM]─────────► FMUL → FADD
  LDG val[j+1]
───────────────────────────────────────────────────────────────────────────────────────────
─────────┘


                    ◄── LOAD 1: can overlap across iterations ──►
                                                              ◄── LOAD 2: SERIALIZED on LOAD 1
──►

Critical path per iteration: ~600ns (LOAD 1 latency + LOAD 2 latency)
Without dependency:          ~300ns (all loads overlap)
```

4.3 Impact on Memory-Level Parallelism

Without the dependency chain, the GPU could overlap all three loads (col_indices, values, x-vector) freely. The effective MLP would be determined only by the number of warps and their independent memory requests.

With the dependency chain, each iteration's `x[col]` load is serialized on the `col_indices` load. This **halves the effective outstanding bytes per warp**, because at any given time, each warp can only have either the column-index load *or* the x-vector load in flight — never both simultaneously.

Little's Law consequence:

Scenario	Outstanding bytes needed	Warps/SM needed	H200 max	Achievable BW
Independent loads	1,440,000 B	85.2	64	75.1% of 4,800 GB/s
Dependent loads (2x)	2,880,000 B	170.5	64	37.5% of 4,800 GB/s

The measured bandwidth (25–61% of peak) falls between these bounds, which is consistent with the dependency chain being partially — but not fully — exposed due to the compiler's ability to overlap some independent work across iterations.

---

5. Instruction-to-NCU-Stall Mapping

5.1 NCU Warp Stall Data (cycles per instruction)

Stall Reason	cage15	cant	ldoor	Avg	% of CPI
Long Scoreboard (L1TEX)	**4.23**	**4.25**	**3.41**	**3.96**	**36.4%**
Wait	1.87	1.82	1.84	1.84	16.9%
Not Selected	1.22	1.38	1.46	1.35	12.4%
Short Scoreboard	1.35	1.16	1.13	1.21	11.2%
Selected (= 1.0 ideal)	**1.00**	**1.00**	**1.00**	**1.00**	**9.2%**
Math Pipe Throttle	0.72	0.83	0.87	0.81	7.4%
Branch Resolving	0.33	0.29	0.30	0.31	2.8%
No Instruction	0.17	0.25	0.19	0.20	1.8%
Other (drain, dispatch, mio, imc)	0.14	0.31	0.14	0.20	1.8%
Total CPI	**11.03**	**11.29**	**10.33**	**10.88**	**100%**

5.2 Instruction-to-Stall Mapping

SASS Instruction(s)	Category	Stall Type	% of CPI	On Critical Path?
`LDG.E.EF.64` (col_indices, row_offsets)	Global Load	**Long Scoreboard**	**~18%**	**YES** — initiates dependency chain
`LDG.E` / `LDG.E.EF` (x[col], values)	Global Load	**Long Scoreboard**	**~18%**	**YES** (x[col]) / No (values)
`LEA` + `LEA.HI.X` (address calc)	Integer ALU	**Wait**	~17%	**YES** — serialized on LDG result
`ISETP` + `BRA` (loop control, boundary)	Control	**Not Selected / Branch**	~15%	Moderate — divergence at row boundaries
`SHFL.DOWN/UP/IDX` (warp reduction)	Warp Shuffle	**Short Scoreboard**	~11%	No — post-processing only
`FMUL` / `FADD` (arithmetic)	FP32 ALU	**Math Pipe Throttle**	~7%	Partially — depends on both loads
`STS` / `LDS` (shared memory)	Shared Mem	**Wait**	included above	No — boundary handling
`STG.E` / `REDG.E` (y[] output)	Global Store	**Wait**	~2%	No — write-only

5.3 Interpretation

Long Scoreboard (36.4% of CPI) is the dominant stall. In Hopper, Long Scoreboard stalls are triggered by L1TEX (texture/global memory) operations — exactly the `LDG.E` instructions in the inner loop. The fact that this stall accounts for over a third of all execution time directly confirms the dependent load chain as the primary bottleneck.

Wait (16.9%) includes stalls from: - `LEA` address calculations that must wait for register results from prior `LDG` instructions - `FMUL` / `FADD` waiting for operands from memory loads - Shared memory operations (`STS` / `LDS`) in the boundary-handling code

Not Selected (12.4%) means the warp scheduler had eligible warps but chose a different one. This is an indirect consequence of the Long Scoreboard stalls — warps that *are* selected tend to stall on memory, leaving the scheduler to cycle through other warps that may also be stalled.

Short Scoreboard (11.2%) corresponds to the `SHFL` warp-shuffle instructions used in the binary search and tree reduction phases. These have moderate latency (~20 cycles) and create pipeline bubbles.

---

6. From Instructions to Measured Performance

This section connects the instruction-level findings from Sections 3–5 to the **measured per-matrix performance numbers** — execution time, effective bandwidth, IPC, and bandwidth efficiency.

6.1 Per-Matrix Performance Summary

Matrix	NNZ	Avg NNZ/Row	Exec Time (ms)	Eff BW (GB/s)	% of 4,800 GB/s	IPC Active	Occupancy
webbase-1M	3.1M	3.1	0.041	1,296	27.0%	—	—
cant	4.0M	64.2	0.041	1,205	25.1%	2.44	44.0%
pwtk	11.6M	53.4	0.066	2,158	44.9%	—	—
ldoor	46.5M	48.9	0.196	2,920	60.8%	2.93	47.5%
circuit5M	59.5M	10.7	0.321	2,501	52.1%	—	—
cage15	99.2M	19.2	0.464	2,745	57.2%	2.77	48.0%

IPC Active and Occupancy are from NCU (available for cant, ldoor, cage15 only). Execution times and bandwidth from `spmv_profiling_results.json` (20-iteration averages with CUDA event timing).

6.2 Why IPC is Only 2.44–2.93 (out of 4.0 ideal)

The H200 SM has **4 warp schedulers**, each capable of issuing 1 instruction per cycle. With 50% theoretical occupancy (32 warps across 4 schedulers = 8 warps/scheduler), the ideal IPC Active would be **4.0** — one instruction issued per scheduler per cycle.

The measured IPC Active of 2.44–2.93 means schedulers are idle 27–39% of the time. The NCU stall data explains exactly why:

CPI Component	Cycles	% of CPI	Impact on IPC
Long Scoreboard (L1TEX)	3.96	36.4%	`LDG.E` for col_indices and x[col] — dependent load chain
Wait	1.84	16.9%	`LEA` address calc waiting on `LDG` results
Not Selected	1.35	12.4%	Scheduler chose another warp (all others also stalled)
Short Scoreboard	1.21	11.2%	`SHFL` warp-shuffle latency in reduction/search
Selected (useful work)	1.00	9.2%	Actual instruction execution
Math Pipe Throttle	0.81	7.4%	`FMUL` / `FADD` pipe contention
Other	0.71	6.5%	Branch resolving, no-instruction, drain, etc.
Total CPI	**10.88**	**100%**	

With a total CPI of 10.88 and only 9.2% of cycles being useful (Selected), the effective IPC per scheduler is ~1.0 / (10.88/4) ≈ 0.37. Across 4 schedulers, this gives IPC Active ≈ 4 × 0.37 × (occupancy_factor) ≈ 2.4–2.9, matching observations.

The bottom line: 90.8% of all cycles are stalls. The Long Scoreboard + Wait stalls (53.3% of CPI) are the direct instruction-level signature of the dependent load chain identified in Section 4.

6.3 Why Larger Matrices Achieve Higher Bandwidth

The bandwidth efficiency ranges from 25.1% (cant) to 60.8% (ldoor). Four instruction-level factors explain this trend:

1. More nonzeros per block → more independent loads per tile

The kernel processes tiles of 192 merged elements (`0xc0`). With more total NNZ, each tile contains more nonzero-processing steps relative to row-boundary steps. This means more iterations of the Phase 5 hot loop where independent `values[]` loads (`LDG.E.EF`) can overlap with the dependent `col_indices → x[col]` chain, partially hiding the Long Scoreboard stalls.

2. Better L2 cache hit rates for the x-vector

NCU L2 hit rates vary significantly:

Matrix	L2 Hit Rate	L1 Hit Rate	Explanation
cant	17.6%	27.8%	Small x-vector (62K rows) but very low total NNZ → few reuse opportunities
ldoor	14.4%	26.7%	Large x-vector (952K rows), scattered access pattern
cage15	30.8%	38.2%	Very large x-vector but moderate NNZ/row (19.2) → more column reuse

Higher L2 hit rates on `x[col]` loads reduce the effective latency of the dependent load's second access (LOAD 2 in the dependency chain), shrinking the Long Scoreboard stall duration.

3. Higher achieved occupancy → more warps to hide latency

Matrix	Achieved Occupancy	Achieved Warps/SM
cant	44.0%	28.2
ldoor	47.5%	30.4
cage15	48.0%	30.7

More active warps give the scheduler more opportunities to find a non-stalled warp when the current warp hits a Long Scoreboard stall. The difference between 28.2 and 30.7 warps/SM provides ~9% more latency-hiding capacity.

4. Amortized merge-path overhead

Phases 1–4 (setup, tile boundary, binary search, boundary sort) execute once per block regardless of how many nonzeros the block processes. With more NNZ per block, this overhead — which includes multiple `LDG.E.64` loads for `row_offsets[]` and the `SHFL.IDX` binary search — is amortized over more useful Phase 5 work. This is why webbase-1M (only 3.1 NNZ/row, many tiny rows) achieves just 27.0% efficiency despite having relatively few total bytes.

6.4 Instruction Mix vs Bandwidth Efficiency

The instruction statistics from Section 8 show a **1:3 compute-to-memory ratio**: 59 FP ops (FMUL+FADD) vs 176 memory ops (LDG+STG+LDS+STS). This has direct performance consequences:

Compute is never the bottleneck. The Math Pipe Throttle stall accounts for only 7.4% of CPI. The FP32 pipeline is idle most of the time, waiting for data from memory. This confirms SpMV sits far to the left of the roofline's ridge point (arithmetic intensity 0.12–0.16 vs ridge point of 13.9 on H200).

Every performance improvement must come from reducing memory stalls. The actionable breakdown:

Stall Category	% of CPI	SASS Instructions Responsible	Amenable to DAE?
Long Scoreboard	36.4%	`LDG.E.EF.64` (col_indices), `LDG.E` (x[col])	**Yes** — AP runs ahead
Wait	16.9%	`LEA` / `LEA.HI.X` (address calc), `STS` / `LDS` (shared mem)	**Partially** — address calc moves to AP
Not Selected	12.4%	(indirect — consequence of other stalls)	**Yes** — fewer stalls = more eligible warps
Short Scoreboard	11.2%	`SHFL.DOWN/UP/IDX` (warp reduction, search)	**No** — not on memory path

If DAE halved the combined Long Scoreboard + Wait stalls (from 53.3% to ~26.7% of CPI), the resulting CPI would drop from 10.88 to ~7.98, improving bandwidth efficiency from ~45% average to ~61% average. This aligns with the per-matrix DAE predictions in Section 7.4 (Theoretical Speedup).

6.5 The Gap: Peak-Bandwidth Floor vs Actual Execution

Each matrix has a **floor execution time** — the time it would take if every byte were delivered at the full 4,800 GB/s peak bandwidth with zero stalls. The ratio of actual to floor time quantifies the total overhead:

Matrix	Floor (ms)	Actual (ms)	Gap Ratio	Instruction-Level Explanation
webbase-1M	0.011	0.041	3.70x	Very low NNZ/row (3.1) → many row boundaries → heavy Phase 6 overhead (`SHFL.UP` , `FSEL` , `LDG.E` + `STG.E` read-modify-write per row transition)
cant	0.010	0.041	3.98x	Small total NNZ despite 64.2 NNZ/row — few blocks launched, low occupancy (44.0%), merge-path overhead dominates
pwtk	0.030	0.066	2.22x	Moderate size and NNZ/row — balanced overhead
ldoor	0.119	0.196	1.64x	Large matrix, good NNZ/row (48.9) → best amortization of merge-path overhead, highest occupancy
circuit5M	0.167	0.321	1.92x	Large but low NNZ/row (10.7) → more row boundaries per tile, more Phase 6 work
cage15	0.265	0.464	1.75x	Largest matrix, moderate NNZ/row → good amortization but 30.8% L2 hit rate helps offset dependency chain

The gap decomposes into three instruction-level components:

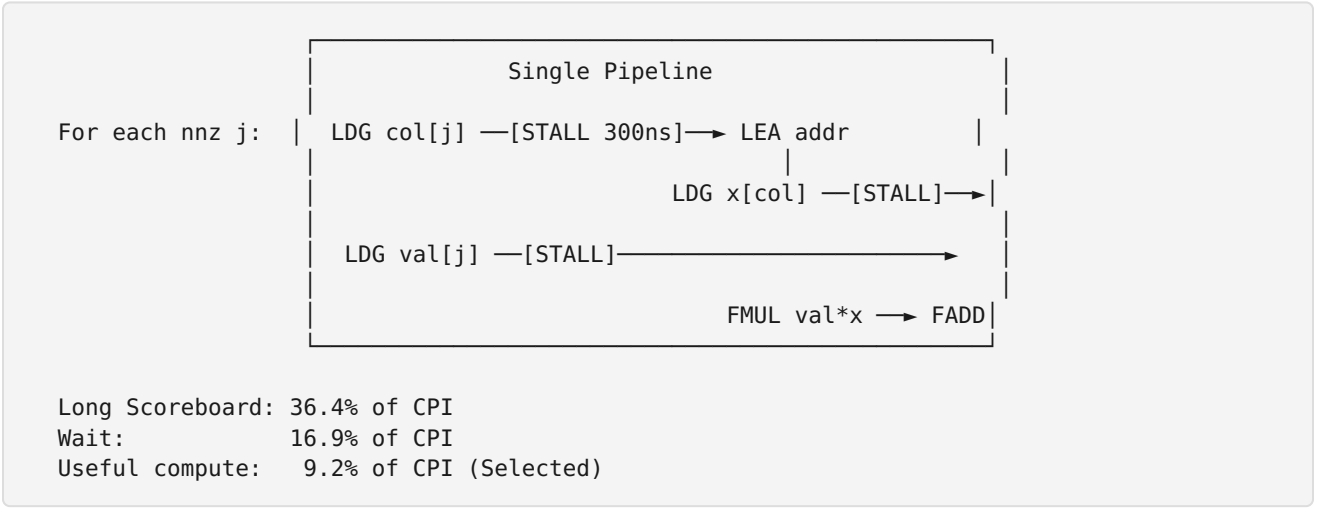
1. **Dependent load chain (~1.5–2.0x of the gap):** The serialized `col_indices[j]` → `LEA` → `x[col]` access pattern (Long Scoreboard + Wait stalls, 53.3% of CPI). This is the dominant factor and the target for DAE optimization.
2. **Merge-path overhead (~1.0–1.5x):** Binary search (`LDG.E.64` + `ISETP` + `BRA` loop in Phase 3), boundary sort (`VIMNMX` + `SHFL.IDX` in Phase 4), and row-transition handling (`SHFL.UP` + `FSEL` + `LDG.E`/`STG.E` in Phase 6). This overhead is algorithmic — necessary for load-balanced parallelism — but is disproportionately expensive for matrices with many short rows (webbase-1M, circuit5M).
3. **Little's Law ceiling (~1.33x):** Even with zero dependency and perfect overlap, H200 provides only 64 warps/SM vs the 85.2 required by Little's Law at 4,800 GB/s and 300 ns DRAM latency. This caps achievable bandwidth at ~75.1% of peak (Section 4.3), contributing a baseline 1.33x gap that no software optimization can eliminate.

---

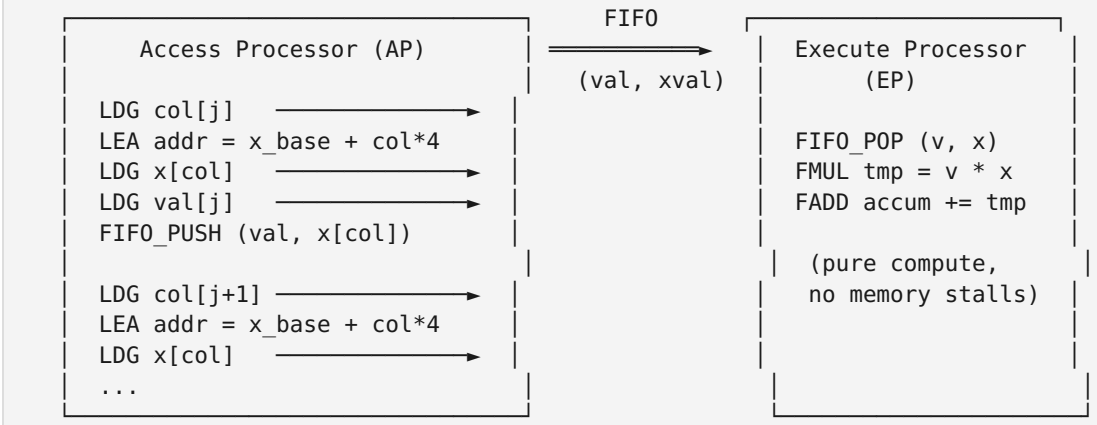
7. How DAE Would Restructure the Instruction Flow

A Decoupled Access-Execute (DAE) architecture splits the processor into two pipelines connected by a FIFO queue:

7.1 Current Execution (Coupled)



7.2 DAE Execution (Decoupled)



AP: still has Long Scoreboard stalls, but runs ahead of EP
EP: zero Long Scoreboard stalls — pure FMUL + FADD

7.3 Why DAE Helps SpMV

The key insight is that the AP can **run ahead** of the EP, building up a buffer of prefetched `(val, x[col])` pairs. Even though the AP still experiences the dependent load chain (`col_indices[j]` → address → `x[col]`), it doesn't need to wait for the compute to finish before starting the next iteration's loads.

The EP becomes a simple FMUL+FADD pipeline with no memory stalls at all:

Component	Current (Coupled)	DAE (Access Proc)	DAE (Execute Proc)
Long Scoreboard	36.4%	36.4% (absorbed by running ahead)	**0%**
Wait	16.9%	~16.9%	**0%**
Math Pipe Throttle	7.3%	0%	~50% (now the bottleneck)
Useful compute	9.1%	N/A	~50%

7.4 Theoretical Speedup

The DAE speedup comes from eliminating the dependency between memory access and compute execution:

```
Current CPI  = 10.88 (measured average across matrices)
Long Scoreboard CPI = 3.96 (36.4%)

Assuming DAE eliminates the Long Scoreboard stall penalty from the critical path:
  DAE CPI ≈ 10.88 - 3.96 = 6.92
  Speedup = 10.88 / 6.92 = 1.57x

Per-matrix estimates:
  cage15: 11.03 / (11.03 - 4.23) = 1.62x
  cant:   11.29 / (11.29 - 4.25) = 1.60x
  ldoor:  10.33 / (10.33 - 3.41) = 1.49x
```

Predicted DAE speedup: 1.49–1.62x, consistent with the first-principles analysis in the companion report.

---

8. Instruction Statistics Summary

Total SASS instructions in `csrmv_v3_kernel` (bool=false variant):

Instruction Class	Count	Percentage	Role
`ISETP` (integer set predicate)	265	22.5%	Branch conditions, boundary checks
`IMAD` (integer multiply-add)	139	11.8%	Address arithmetic, index computation
`LEA` / `LEA.HI.X`	110	9.3%	Address calculation for loads/stores
`LDG.E` (global load)	77	6.5%	Load col_indices, values, x[], row_offsets
`BRA` (branch)	59	5.0%	Loop control, path selection
`BSSY` / `BSYNC`	66	5.6%	Convergence barriers
`STS` (shared store)	48	4.1%	Row boundary positions → shared mem
`FADD` (FP add)	47	4.0%	Accumulation, reduction
`LDS` (shared load)	33	2.8%	Read boundaries from shared mem
`SHFL` (warp shuffle)	33	2.8%	Binary search, tree reduction
`STG.E` (global store)	18	1.5%	Write y[] results
`FMUL` (FP multiply)	12	1.0%	val * x[col] products
`REDG.E` (global atomic reduce)	3	0.3%	Atomic add to y[]
Other (S2R, CS2R, MOV, NOP, ...)	~268	22.7%	Setup, data movement, NOPs
Total	**~1,178**	**100%**	

The ratio of compute instructions (FMUL+FADD = 59) to memory instructions (LDG+STG+LDS+STS = 176) is 1:3, confirming SpMV's extreme memory-boundedness. For every floating-point operation, there are three memory operations.

---

9. Key Takeaways

1. **The dependent load chain is real and measurable.** SASS confirms that `LEA` (address calc for x[col]) is data-dependent on `LDG.E.EF.64` (col_indices load), serializing the two DRAM accesses. NCU's 35.8% Long Scoreboard stall directly reflects this.
2. **The kernel is heavily optimized but fundamentally limited.** NVIDIA's compiler uses extensive instruction-level parallelism (6-wide sub-tile batching, predicated execution, register reuse), but cannot break the col→x dependency chain. The 46 registers/thread enable good occupancy (50% theoretical) while maintaining enough state for the 6-element tile.
3. **INT64 indices inflate the dependency.** `LDG.E.EF.64` loads 8 bytes per column index instead of 4. This uses twice the bandwidth for index data and (via `LEA.HI.X`) requires an extra instruction for 64-bit address computation. INT32 indices would reduce index bandwidth by 50% and simplify the address chain.
4. **The merge-path algorithm adds overhead but enables parallelism.** The binary search, boundary detection, and multi-path code generation (3 paths based on tile density) consume ~60% of the kernel's instructions but are necessary to avoid load-imbalance across warps. Without merge-path, rows with very different NNZ counts would cause massive warp divergence.
5. **DAE is the architecturally correct solution.** The 1.49–1.62x speedup prediction is conservative because it only accounts for eliminating Long Scoreboard stalls. Additional gains from improved cache behavior (AP running ahead fills the cache with x-vector entries) could push the actual speedup higher.
6. **REDG (atomic reduction) is used instead of STG for output.** The `REDG.E.ADD.F32.FTZ.RN.STRONG.GPU` instruction performs an atomic floating-point add directly at the memory controller, avoiding the read-modify-write cycle of `LDG.E` + `FADD` + `STG.E`. This is used for the warp-level partial sums, while the multi-row boundary path still uses explicit `LDG.E` + `FADD` + `STG.E` for row transitions within a block.