TRAIN NEO BIT

Cognichip Hackathon

EcoTraining: An Energy-Efficient Hardware Design for Gradient Compression in LLM Training

By Feiyu Jia, Heng Pu, Lixuan Xu, Yuhan Jiang

https://github.com/fjia-xu/Cognichip-Hackathon-by-Train-Neo-Bit.git

CONTENTS

- 01 Problem Statement & Motivation
- 02 RTL Architecture
- 03 Design Methodology
- 04 Simulation Results
- 05 Challenges & Lessons Learned
- 06 Future Work

01

Problem Statement & Motivation

Problem Statement: “Memory Wall” in AI Training

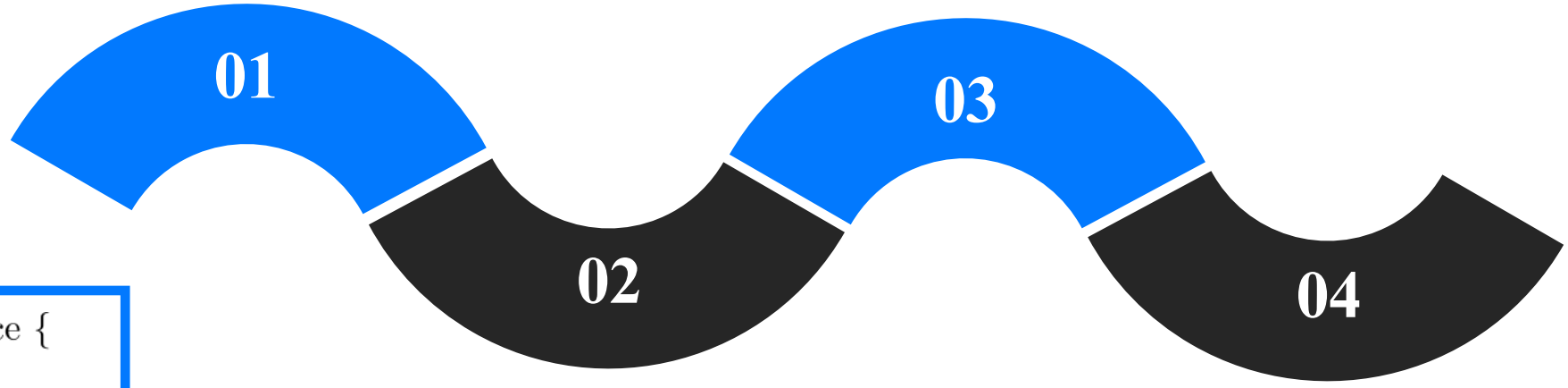
GPU Training Steps

Forward Pass

Memory wall: HBM/cache bandwidth; activation write traffic

Backward Pass

Memory wall: heavy gradient/activation traffic + write-backs



Repeat Until Convergence {

for i = 1...m {

$\omega \leftarrow \omega - \alpha * \nabla_w L_m(w)$

}

}

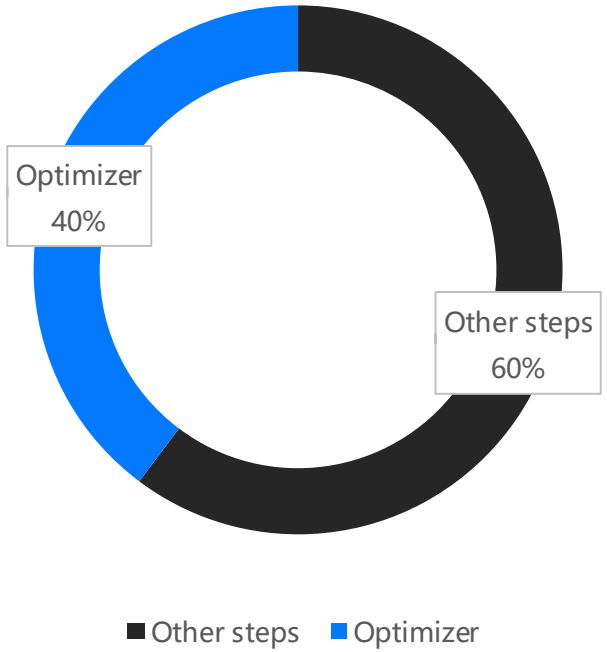
Loss Computation

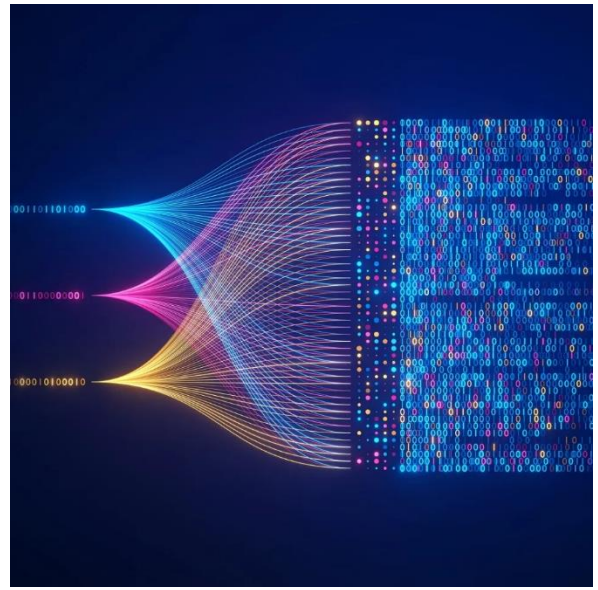
Optimizer Step

Memory wall: write-intensive state updates + cache/DRAM pressure

Problem Statement: “Memory Wall” in Gradient Update

The **memory wall** becomes acute at the **gradient update (optimizer) stage** because the update is dominated by streaming reads/writes of large tensors: for each parameter, the optimizer typically reads the weight and its gradient, reads and updates optimizer state (often larger than the model itself), and writes back the updated state and weight.

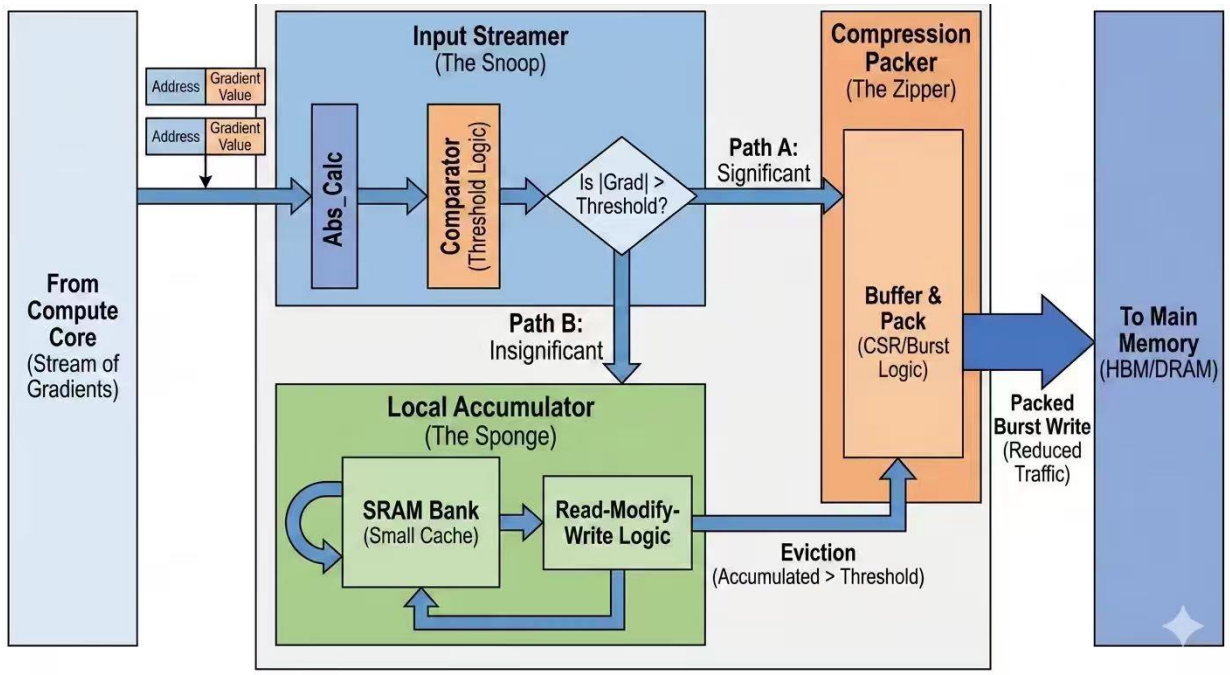




02

RTL Architecture

Gradient Compressor Architecture



This diagram illustrates our group's design of a **gradient compressor for LLM acceleration**. The architecture is organized into three main stages: the Streamer, the L1 Accumulator, and the L2 Writeback Buffer. We assume a single-cycle processing model for simplicity.

This design reduces DRAM traffic by separating large and small gradient handling:

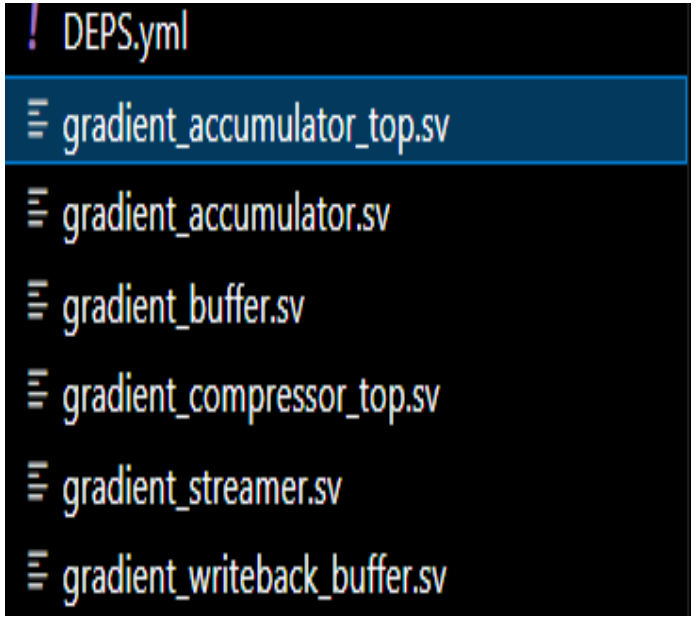
Large gradients bypass accumulation and are written back directly.

Small gradients are accumulated locally in L1 to compress updates before writing back.

L2 aggregates writeback traffic to enable efficient burst transmission to DRAM.

(In the future we can try our best to make these stages pipelined but this time we assume it was the single-cycle assumption)

Gradient Compressor Architecture: RTL Design Logic hierarchy



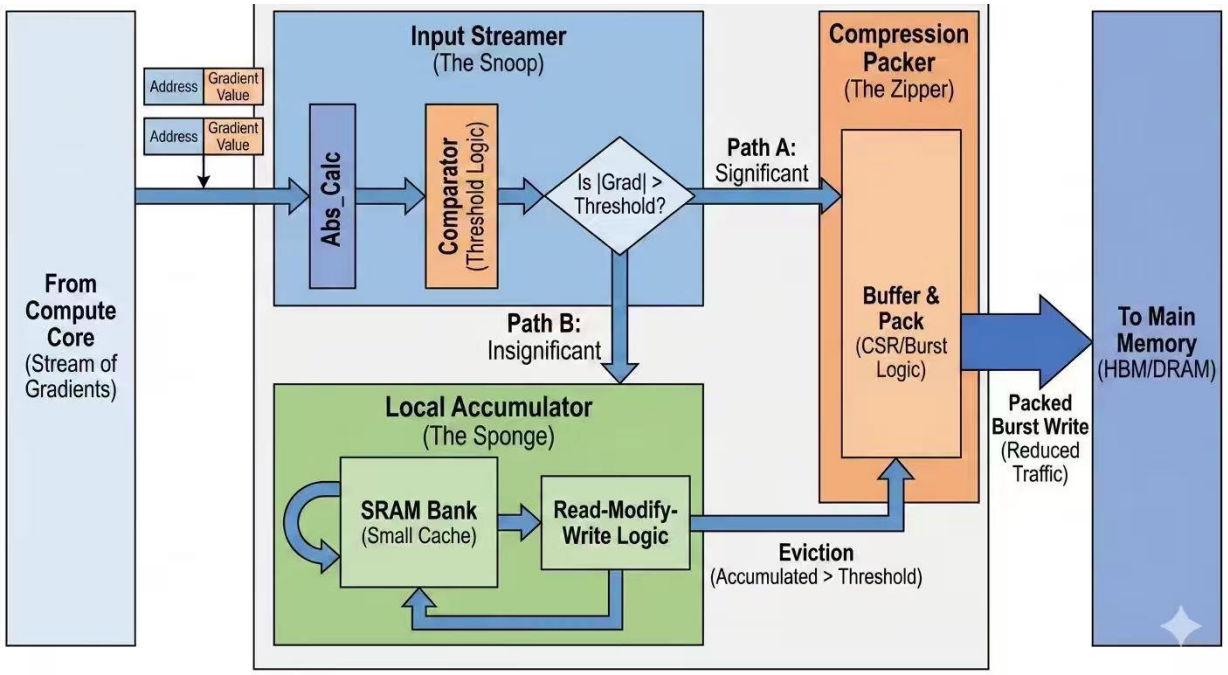
The gradient_accumulator.sv is used for set up interface connection between the whole ALU part and the L2 buffer

The gradient_accumulator_top.sv and streamer dive into the alu logic design which is used for the logic detection and doing data distribution, Writeback Decision and L1 Update Logic

The gradient_accumulator_buffer is used for the data storage and tag storage

The gradient_compressor_top is the main top which used to exposed the interface for the testbench to operate

Gradient Compressor Architecture: Streamer



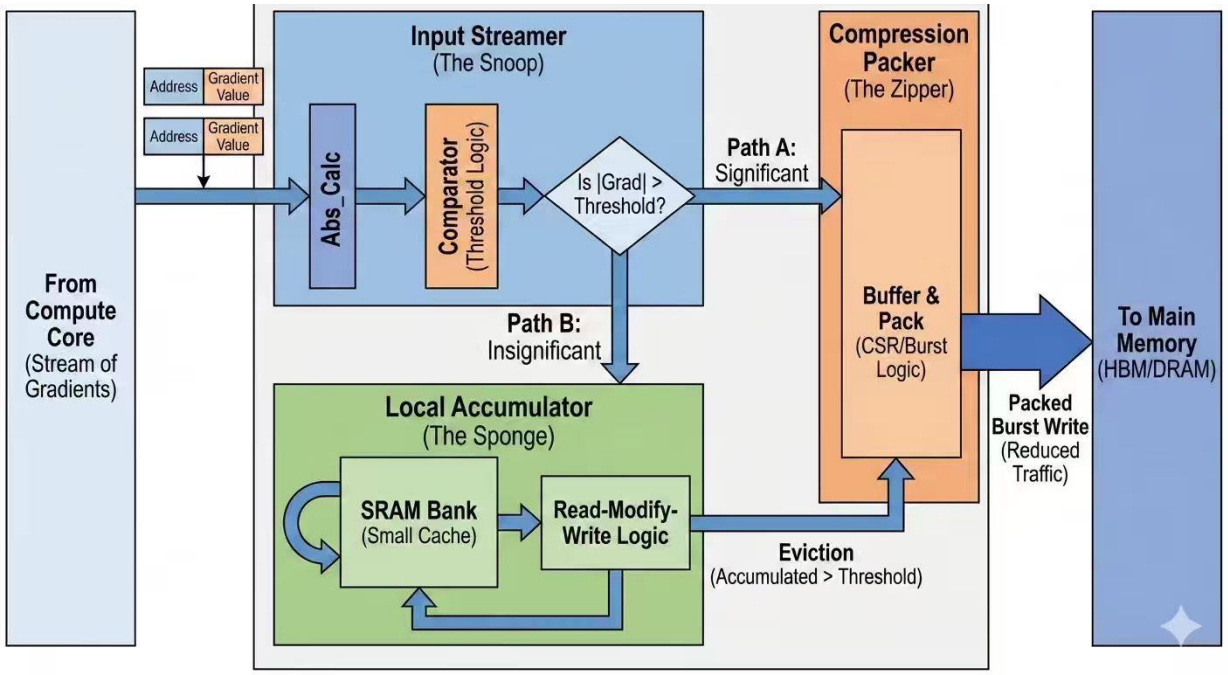
Stage 1: Streamer

The first block, called the **Streamer**, receives gradient inputs from the testbench on every cycle. Its primary function is to perform threshold comparison and classify incoming gradients.

If the magnitude of the gradient exceeds the predefined threshold, it is considered a large update. In this case, the gradient bypasses the L1 Accumulator and is directly forwarded to the L2 buffer. These large updates are temporarily stored in L2 and later transmitted to DRAM in burst mode when the buffer becomes full.

If the gradient magnitude is below the threshold, it is considered a small update and is sent to the Accumulator stage for further processing.

Gradient Compressor Architecture: L1 Accumulator



Stage 2: L1 Accumulator (Set-Associative)

The **Accumulator** is designed to accumulate small gradients efficiently. It is implemented as a set-associative L1 structure.

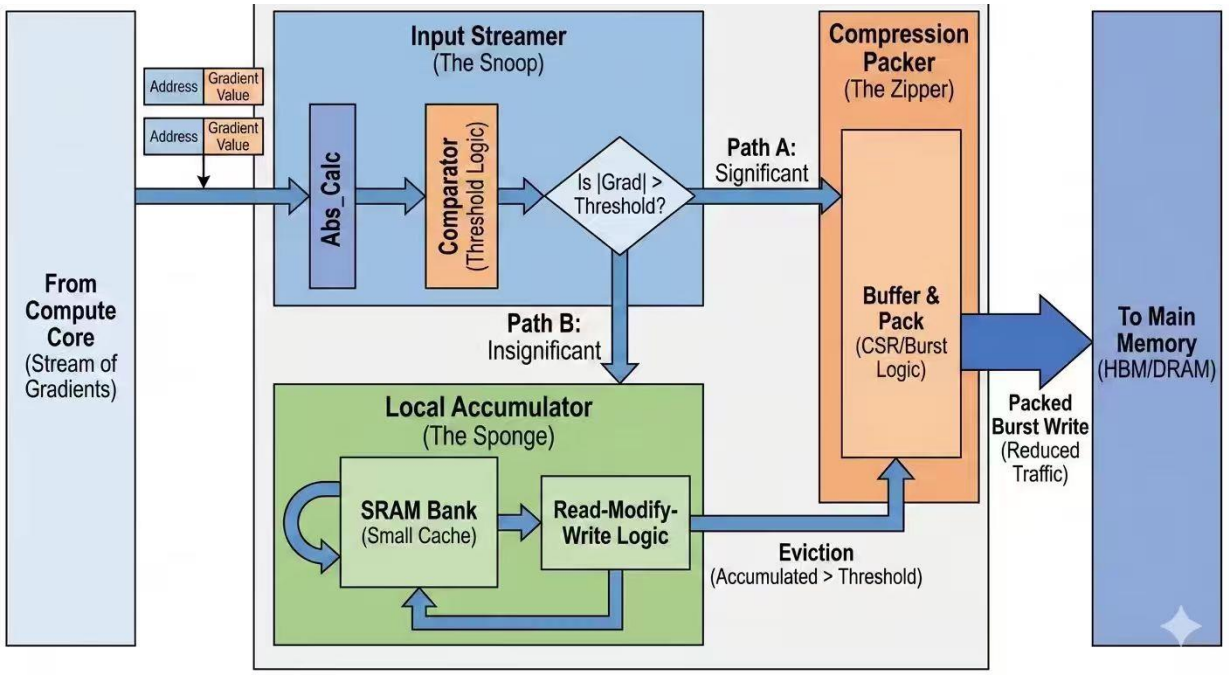
When a small gradient arrives:

- If the tag matches an existing entry (hit), the gradient is accumulated with the stored value.
- If the accumulated value reaches or exceeds the threshold, it is immediately pushed to the L2 buffer.
- If the accumulated value remains below the threshold, it stays in the L1 entry for further accumulation.

Data may also be written back to L2 under the following conditions:

- **Eviction**: This occurs only when there is a tag miss and the corresponding set is full. A victim entry is selected and written back to L2.
- **Maximum Update Count**: If an entry's update counter reaches 255, a forced writeback is triggered to prevent indefinite accumulation.

Gradient Compressor Architecture: L2 Writeback Buffer



Stage 3: L2 Writeback Buffer

The **L2 buffer** stores all writeback data, including:

- Large gradients bypassed from the Streamer
- Accumulated gradients flushed from L1
- Evicted entries

When the L2 buffer becomes full, it transmits its stored data to DRAM in burst mode. This burst-based transfer improves memory bandwidth efficiency and reduces frequent small writes to DRAM.

03

Design Methodology

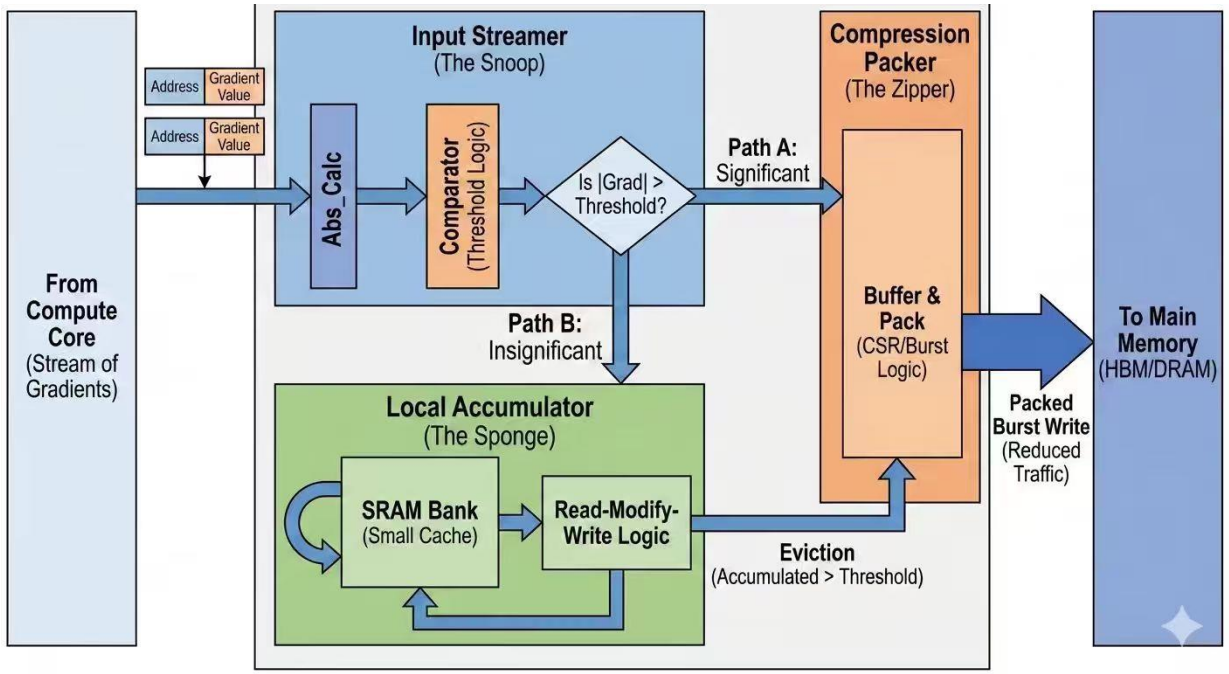
Design Methodology

First Step--Problem Definition

The hardware design describes a Gradient Compressor and Accumulator, which is typically used in machine learning accelerators to reduce memory bandwidth by locally caching and accumulating weight gradients before writing them back to main memory (DRAM).

The system implements a hierarchical, two-level write-combining architecture (L1 cache and L2 FIFO) to decouple compute from memory latency.

Second Step--Design Structured Circuits



Design Methodology

Third Step--Generate Submodule Prompts & Summary

Module 1: The Input Streamer (The Snoop)

Function: Filters noise. It decides if the incoming integer gradient is worth processing immediately or if it can be buffered.

Fundamental Units:

1. **Input Bus:**
   - Address_In (32-bit): The memory address of the weight.
   - Gradient_In (16-bit Signed Integer): The value to update.
2. **Absolute Value Logic (ABS):**
   - *Unit:* A simple logic block that converts negative numbers to positive.
   - *Logic:* if (Gradient_In < 0) return (~Gradient_In + 1); else return Gradient_In; (Two's Complement).
3. **Comparator (Magnitude):**
   - *Unit:* Digital Comparator.
   - *Logic:* if (Abs_Gradient > Threshold_Reg) -> Path A; else -> Path B;
4. **Path Mux (Router):**
   - *Unit:* 1-to-2 Demultiplexer.
   - *Operation:* Routes the {Address, Gradient} pair to either the **Packer (Path A)** or the **Accumulator (Path B)**.

Module 2: The Local Accumulator (The Sponge)

Function: A Direct-Mapped Cache that performs a Read-Modify-Write operation in one cycle.

Fundamental Units:

1. **Memory Array (The Storage):**
   - Implemented as an array of Registers (to support single-cycle Read/Write).
   - **Structure:** Each slot holds {Valid_Bit, Tag, Accumulated_Value}.
   - **Indexing:** We use the lower bits of the address (e.g., Address[9:0]) to find the slot.
2. **Tag Comparator:**
   - *Unit:* Equality Checker (==).
   - *Logic:* Compares the stored Tag in the slot with the upper bits of the incoming Address.
   - *Output:* Hit (Match) or Miss (Mismatch).
3. **Integer Adder (The ALU):**
   - *Unit:* Standard Binary Adder.
   - *Operation:* Sum = Gradient_In + Stored_Value.
4. **Overflow Check:**
   - *Unit:* Comparator.
   - *Logic:* Checks if |Sum| > Threshold. If true, we must evict this value immediately.
5. **Write Logic (Muxes):**
   - **Data Mux:** Decides what to write into the slot.
     - If Miss: Write Gradient_In.
     - If Hit: Write Sum.
   - **Write Enable:** Controls if the memory updates.

Module 3: The Compression Packer (The Zipper)

Function: The exit point. It collects "significant" updates and sends them to main memory.

Fundamental Units:

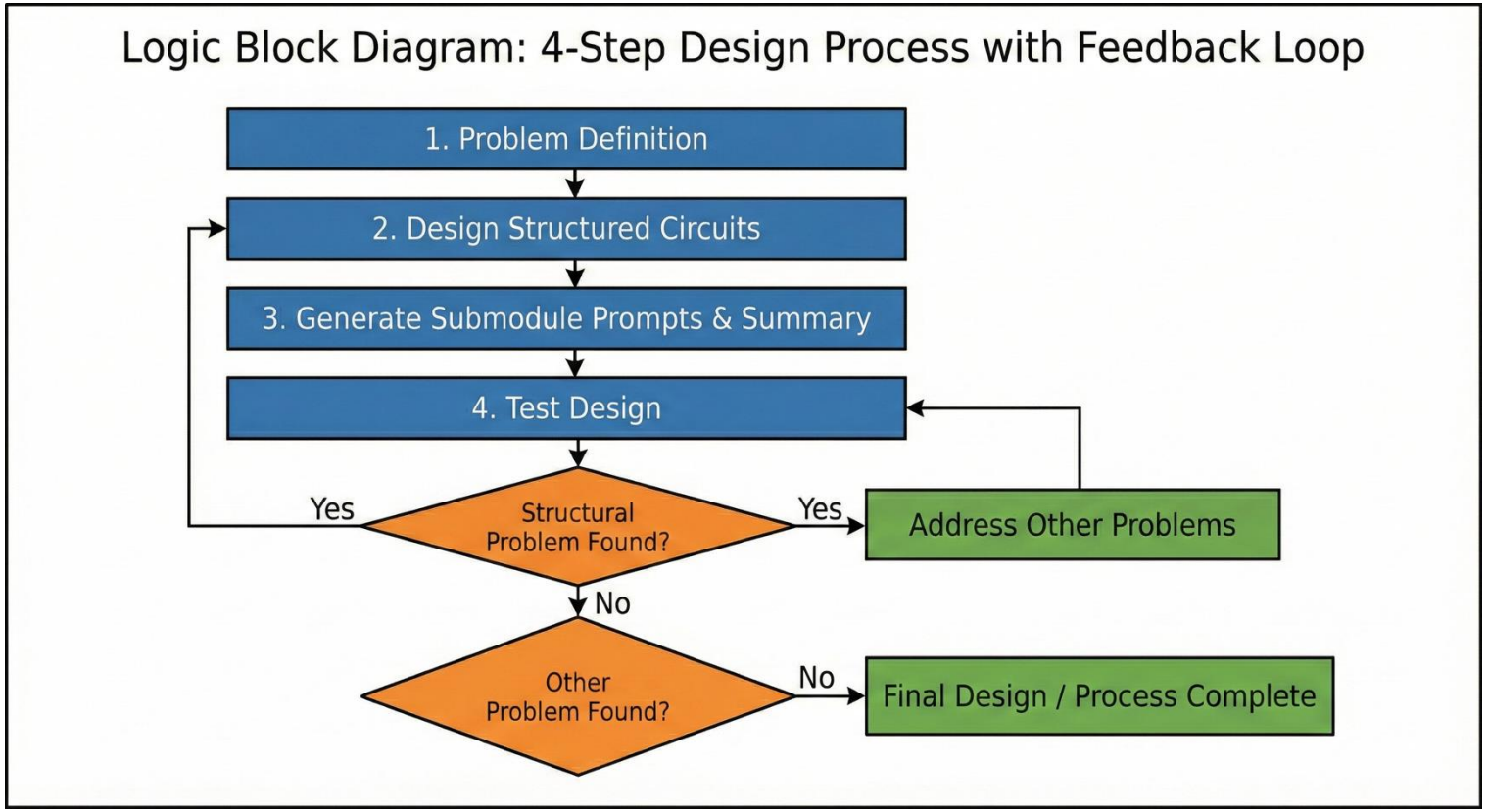
1. **Arbiter (Mux):**
   - *Unit:* 2-to-1 Multiplexer.
   - *Inputs:*
     1. **Path A:** Huge gradients directly from Module 1.
     2. **Eviction Path:** Accumulated values kicked out of Module 2.
   - *Logic:* A simple priority logic (e.g., Path A has priority).
2. **Output Buffer:**
   - *Unit:* A simple Register or small FIFO.
   - *Operation:* holds the {Address, Value} pair and asserts a Valid signal to the memory controller.

First generate the prompt of each submodule and let Cognichip platform generate the .sv file of each submodule. Then generate the top-level file and connect each sub-module.

Design Methodology

Forth Step--Test & Problem Solving

When we run the simulation file and find the problem,
if it is found to be a structural problem, we will redesign the circuits, add or delect some submodules to meet our needs.
Then repeat forth step to test the design.



Design Methodology

Forth Step--Test & Problem Solving

If it's not a structural issue, we just need to patch the files that need modification.

Please APPLY A MINIMAL PATCH to fix four SPEC FAILURES while keeping:
- the SAME module list (no new modules)
- the SAME module interfaces/ports (do NOT add/remove ports)
- the SAME top-level connectivity
- the SAME overall behavior for direct/threshold/max_updates/eviction pushes and atomic commit

You must modify only internal logic (signals/register widths/conditions) inside existing modules.
Output ONLY the patched SystemVerilog code (full file). No explanations.

==========
FAILURE #1: L1 flush scan never terminates (counter wrap)
==========
Problem: In gradient_accumulator_top, `flush_set` is INDEX_BITS wide and `flush_way` is WAY_W wide. Both wrap before reaching termination, so conditions like `flush_set < NUM_SETS` and `flush_way < NUM_WAYS` are always true and flush loops forever.

PATCH REQUIREMENTS:
(A) Widen flush scan counters so they can represent the terminal value:
- Replace:
  logic [INDEX_BITS-1:0] flush_set;
  logic [WAY_W-1:0] flush_way;
  With:
  logic [INDEX_BITS:0] flush_set; // +1 bit
  logic [WAY_W:0] flush_way; // +1 bit

(B) Correct the flush FSM exit conditions so flush finishes deterministically without wrap:
- Interpret flush scan as nested loops over sets and ways:
  for flush_set in [0 .. NUM_SETS-1]:
    for flush_way in [0 .. NUM_WAYS-1]:
      if entry valid -> push then clear
      else advance
- Use explicit checks:
  if (flush_set == NUM_SETS) -> FLUSH_DONE
  else if (flush_way == NUM_WAYS) -> flush_set++, flush_way=0
  else -> process (flush_set, flush_way)

(C) Keep all backpressure rules: if wb_push_ready=0 when a flush push is needed, stall and keep wb_push_* stable until it fires; clear entry ONLY on wb_push_fire.

==========
FAILURE #2: l1_idle is incorrect (has_valid_entries hardcoded 0)
==========
Problem: l1_idle currently ignores actual L1 contents; it becomes true in IDLE even if entries are still valid.

PATCH REQUIREMENTS (minimal restructure):
(A) Add an internal counter `valid_count` in gradient_accumulator_top that tracks number of valid L1 entries:
- Width: enough for DEPTH, e.g.
  localparam int VC_W = $clog2(DEPTH+1);
  logic [VC_W-1:0] valid_count;

(B) Update valid_count ONLY on committed L1 state changes (i.e., on in_fire for no-push updates, and on wb_push_fire for push-related actions):
- Increment when allocating an empty way on miss.
- Decrement when clearing an entry due to:
  - hit flush (threshold/max_updates) after wb_push_fire
  - flush scan clearing after wb_push_fire
- For eviction overwrite (victim replaced by new entry), valid_count does NOT change (valid stays 1).
- For direct trigger, valid_count does NOT change.

(C) Define l1_idle accurately:
- l1_idle = (state == IDLE) && (valid_count == 0) && (flush_state != FLUSHING) && (no pending push)
  (Use your existing state/flags; ensure it is false if flushing or pending.)

(D) Remove the fake has_valid_entries loop or leave it unused; l1_idle must be driven by valid_count.

04

Simulation Results

Simulation: Testbench Architecture & Connectivity

1. Architecture & Connectivity

Testbench (Top-level): tb_unified_gradient_compressor_top
(tb_gradient_compressor_top)

DUT Hierarchy:
L1 Cache: gradient_accumulator_top
L2 Buffer: gradient_writeback_buffer (FIFO)

Interface Handshake: Uses valid/ready protocol for both Core input (core_valid) and Memory output (dram_valid/dram_ready).

```
DEPS.yml > {} sim_unified
1   # 新增的统一测试 target
2   sim_unified:
3     deps:
4       - gradient_buffer.sv
5       - gradient_accumulator_top.sv
6       - gradient_writeback_buffer.sv
7       - gradient_accumulator.sv
8       - gradient_compressor_top.sv
9       - bandwidth_perf_monitor.sv
10      - tb_unified_gradient_compressor.sv
11    top: tb_unified_gradient_compressor
```

2. Monitoring & Logging Logic

Performance Monitor: bandwidth_perf_monitor captures raw_input_tx_count and compressed_output_tx_count.

Data Integrity: Real-time CSV logging via $display to track address/value pairs at both input and output ports.

Simulation: Testbench Architecture & Connectivity

Functional Test Scenarios: Phases 1 to 3

01

Phase 1: Pure Accumulation

Repeatedly sends small gradients (value: 4) to 32 different addresses to check L1 hit and accumulation logic.

02

Phase 2: Outlier Bypass

Sends values (value: 100) exceeding the THRESHOLD (50) to verify the "Direct Trigger" path that bypasses the cache.

03

Phase 3: Address Conflict

Targets specific sets with different tags to test way allocation and initial eviction logic.

Capacity and Limit Testing: Phases 4 to 6

04

Phase 4: Accumulation Overflow

Forces a flush by reaching the THRESHOLD through repeated additions.

05

Phase 5: Full Capacity & Forced Eviction

Fills all 4 ways in all 32 sets, then adds a 5th address to force eviction.

06

Phase 6: MAX_UPDATES Flush

Sends 260 updates to a single address to verify the safety mechanism that flushes data when the update counter (8-bit) hits 255.

Simulation: Performance Metrics and Success Criteria

```
================================================
 BANDWIDTH PERFORMANCE REPORT
================================================
Raw Input Transactions  : 1448 (5792 Bytes)
Output Writes to Memory : 484 (1936 Bytes)
------------------------------------------------------------
Bandwidth Reduction     : 66.57 %
Compression Ratio       : 2.99 x
================================================

- S i m u l a t i o n   R e p o r t: Verilator 5.038 2025-07-08
- Verilator: $finish at 16us; walltime 0.074 s; speed 214.579 us/s
- Verilator: cpu 0.074 s on 1 threads; alloced 203 MB
```

Key Indicators

High Ratio: Expected during Phase 1 (heavy accumulation).

Low Ratio (1.0x): Expected during Phase 2 (all outliers bypass).

FIFO Integrity: dram_valid should stay high during burst periods without losing data.

Compression Ratio

The ratio of raw input transactions to compressed memory writes.

Bandwidth Reduction

Calculated as 1−(Output Tx/Input Tx).

Simulation Results

Final Writeback Statistics:

Direct Trigger (|grad| ≥ THRESHOLD): 19 events

Accumulation Threshold (accum ≥ THRESHOLD): 6 events

MAX_UPDATES Force Flush: 0 events

Eviction (tag conflict): 6 events

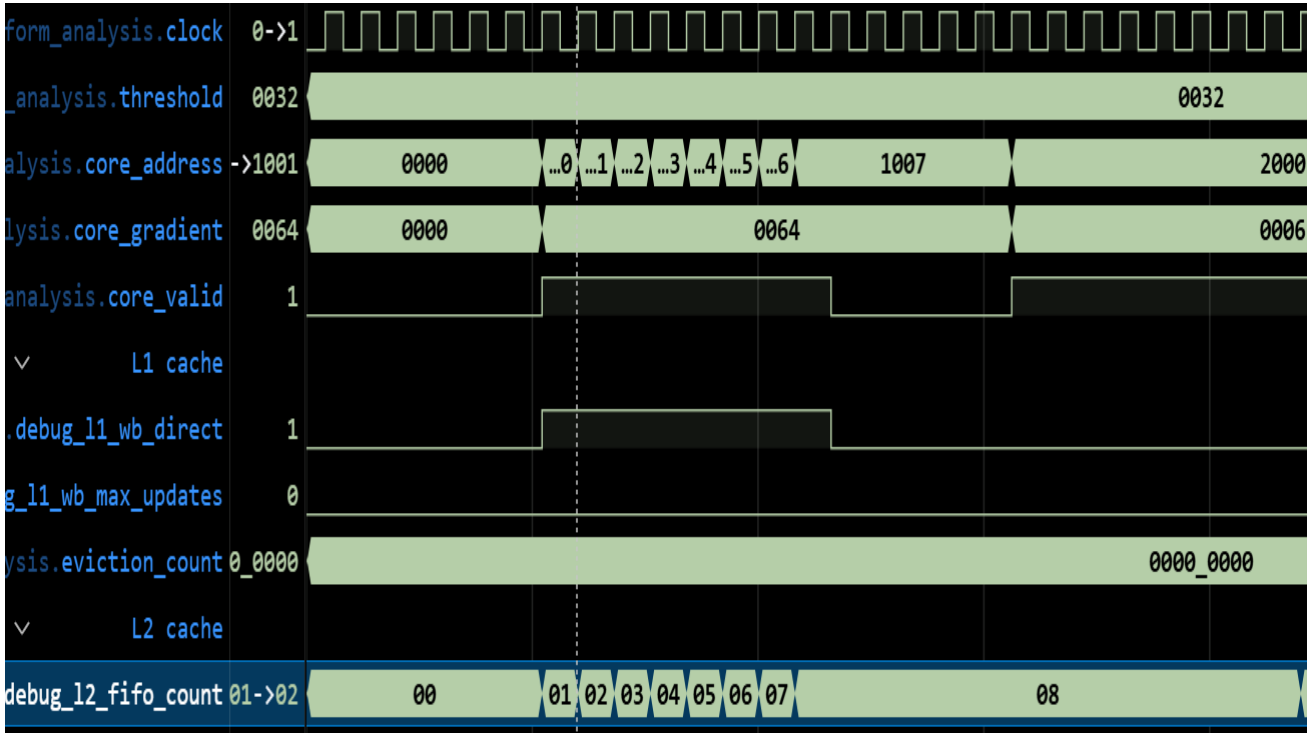
DRAM Writes (FIFO → DRAM): 31 events

Total L1 → FIFO pushes: 31

Total FIFO → DRAM writes: 31

The perfect match between FIFO pushes and DRAM writes confirms full data integrity.

Testbench: Direct Trigger Path



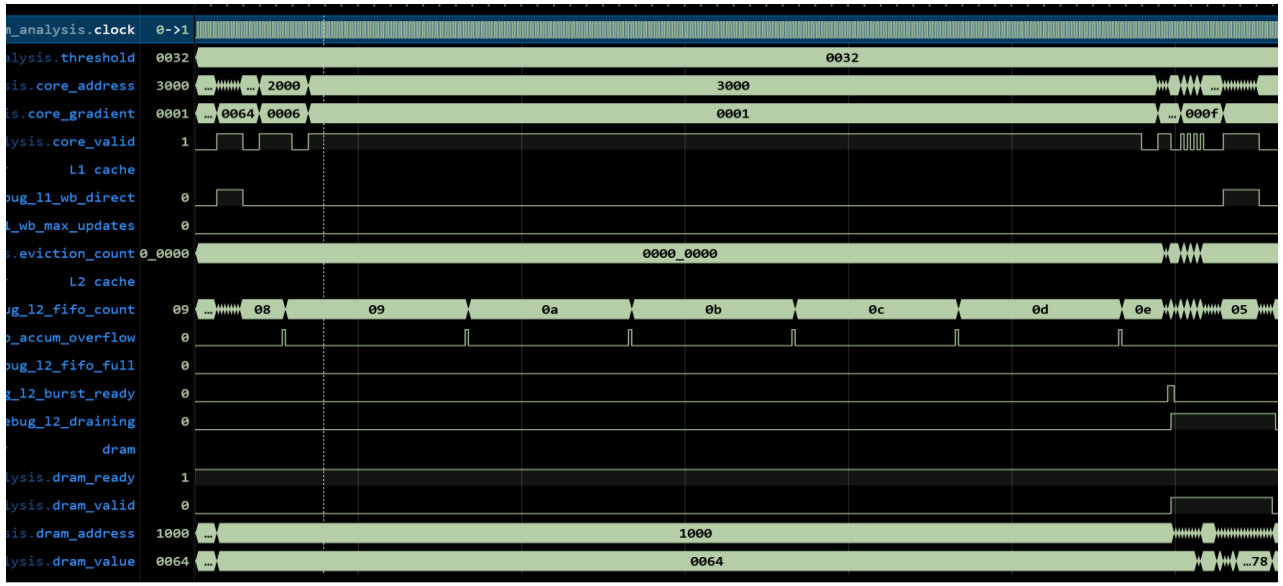
Nineteen large gradients (≥ threshold) correctly bypassed the L1 accumulator and were directly pushed into the L2 FIFO.

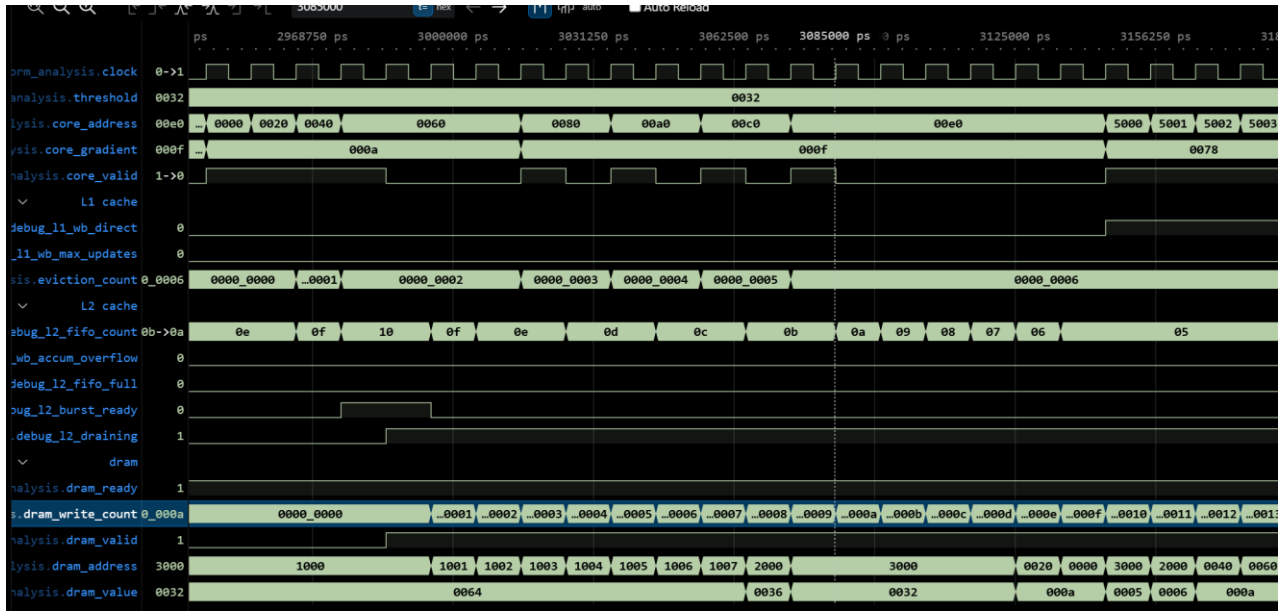
Verified behaviors:

- No L1 allocation
- Immediate FIFO push
- Proper debug signal activation

You can see from here when the fifo2 which is the second stage is always increases due to 64 (hex)
Is bigger than the threshold and keep its stages and stops when the core_valid pulls down to zero.
Which means the fifo2 is always been written data directly from the Streamer.

Testbench: FIFO Burst Writeback





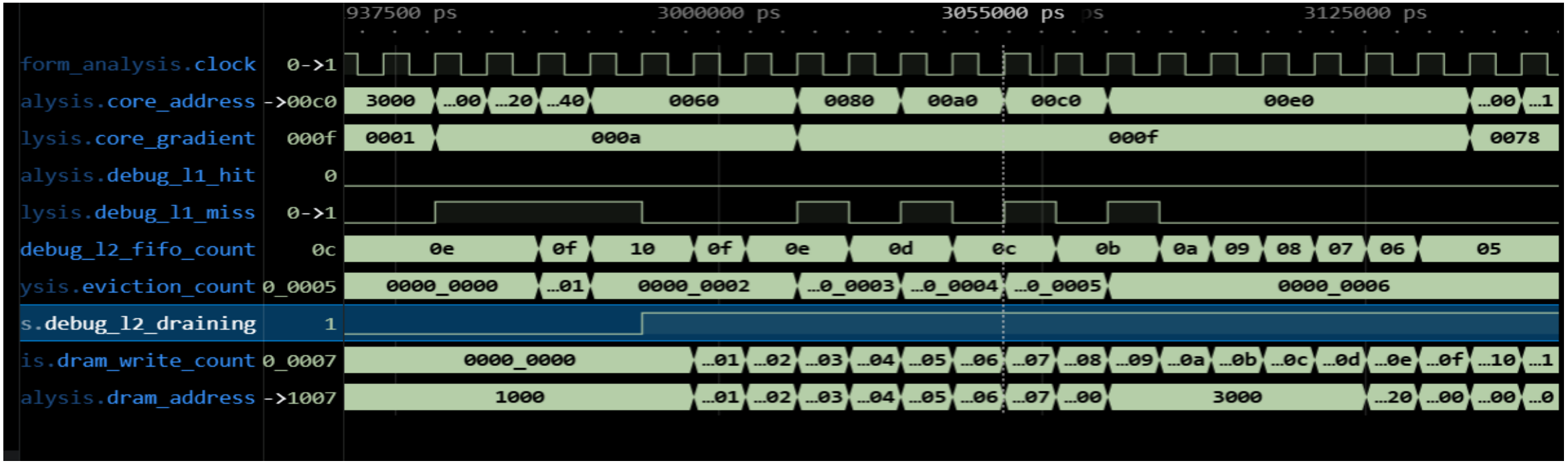
The FIFO triggered burst writeback once the occupancy reached 16 entries.

Verified behavior:

- FIFO count decreased from 16 to 0
- Continuous DRAM write for 31 cycles
- Proper draining state control
- No data loss

You can see from here when the L2 cache draining is happen which means the L2 is writing its data to dram . So obviously , the dram_count is increasing and the fifo count is decreasing at the same time which corresponding to my thinking. The burst_ready signal is where the signal processing start.

Testbench: Eviction Mechanism



Six tag-conflict eviction events were successfully detected and processed.

Each eviction correctly pushed the victim entry into the L2 FIFO.

This is the eviction , when the core address change from 2000-3000-0000-0020-0040-0060-0080,it occurs during the 0040, because the set_full signal which directly assigned to the eviction_count. So, after the 0040 the evicted data flow to the fifo , you can see the fifo count increases from 0f to 10 and then it stops because the fifo is full and is about to output the data to the dram.

Simulation Conclusion

Key Achievements:

Eviction mechanism fully operational

Correct writeback hierarchy behavior

Proper priority handling between flush mechanisms

Complete debug visibility across L1/L2/L3

The simulation completed successfully and all critical mechanisms were verified. The system now functions as intended, with verified correctness across all writeback paths.

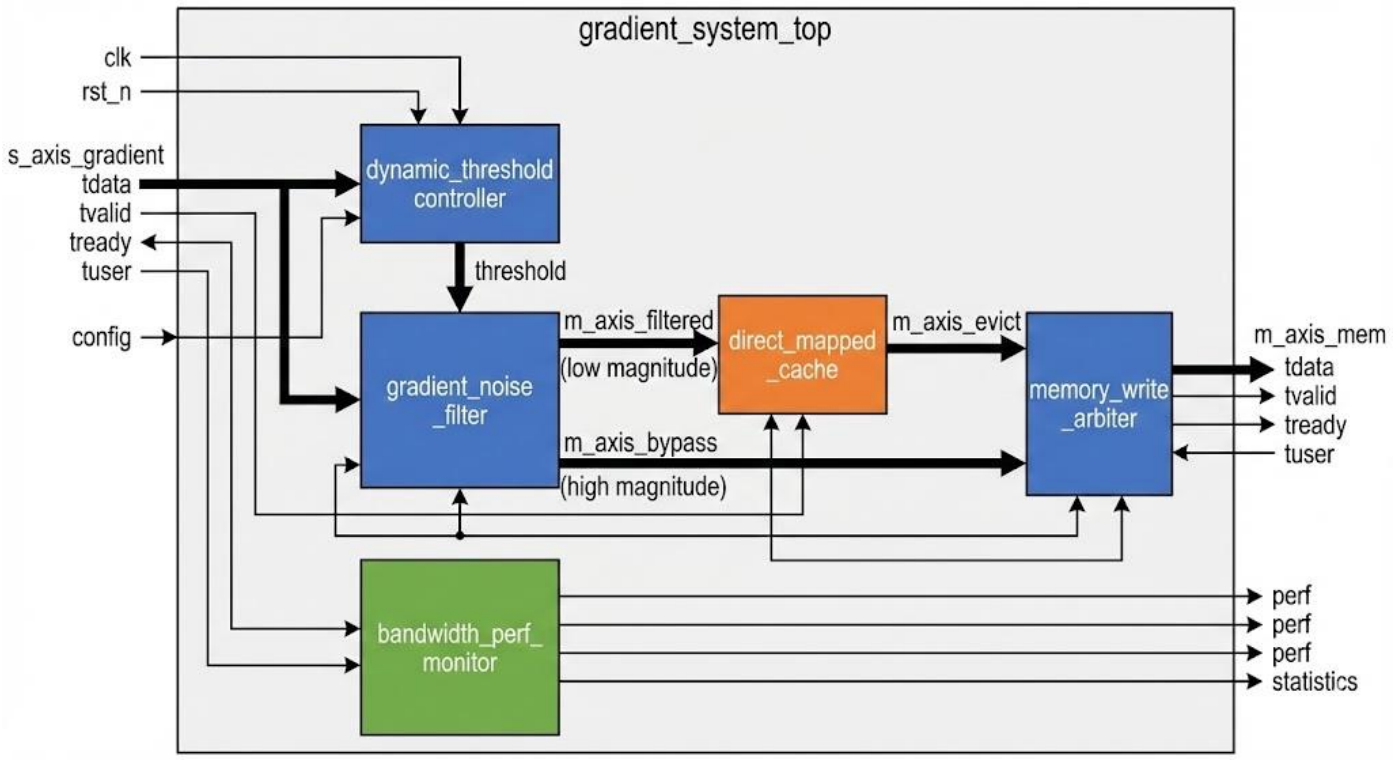
```
================================================
 BANDWIDTH PERFORMANCE REPORT
================================================
Raw Input Transactions  : 1448 (5792 Bytes)
Output Writes to Memory : 484 (1936 Bytes)
----------------------------------------------------------------
Bandwidth Reduction     : 66.57 %
Compression Ratio       : 2.99 x
================================================

- S i m u l a t i o n   R e p o r t: Verilator 5.038 2025-07-08
- Verilator: $finish at 16us; walltime 0.074 s; speed 214.579 us/s
- Verilator: cpu 0.074 s on 1 threads; alloced 203 MB
```

04-5

Some Sub-Designs

I. Dynamic Adaptive Gradient Compressor



Core Module

The system employs a modular and pipelined design, where data flows through four core modules

- **Dynamic Threshold Controller**

  This module computes a dynamic threshold using the Exponential Moving Average (EMA) of incoming gradient magnitudes.

- **Gradient Noise Filter**

- **Direct-Mapped Cache**

- **Memory Write Arbiter**

Simulation Testbench

Pure Accumulation

Target: Verify Cache RMW (Read-Modify-Write).
Action: Multiple small-magnitude updates.
Goal: Ensure correct accumulation, no data loss.

High-Magnitude Bypass

Target: Threshold Decision Logic.
Action: Large gradients bypass cache.
Goal: Immediate update of critical gradients.

Dynamic Convergence

Target: Direct-Mapped Cache vulnerabilities.
Action: Rotate Tag, lock Index (Force Miss & Evict).
Goal: Observe conflict resolution, prevent data corruption.

Mixed Testing

Target: Write-back trigger logic.
Action: Minor fluctuations → Threshold breakthrough.
Goal: Validate memory write upon reaching critical point.

Cache Conflicts

Target: EMA adaptive threshold stability.
Action: Massive random positive/negative gradients.
Goal: Simulate real-world ML gradient distributions.

Simulation Results

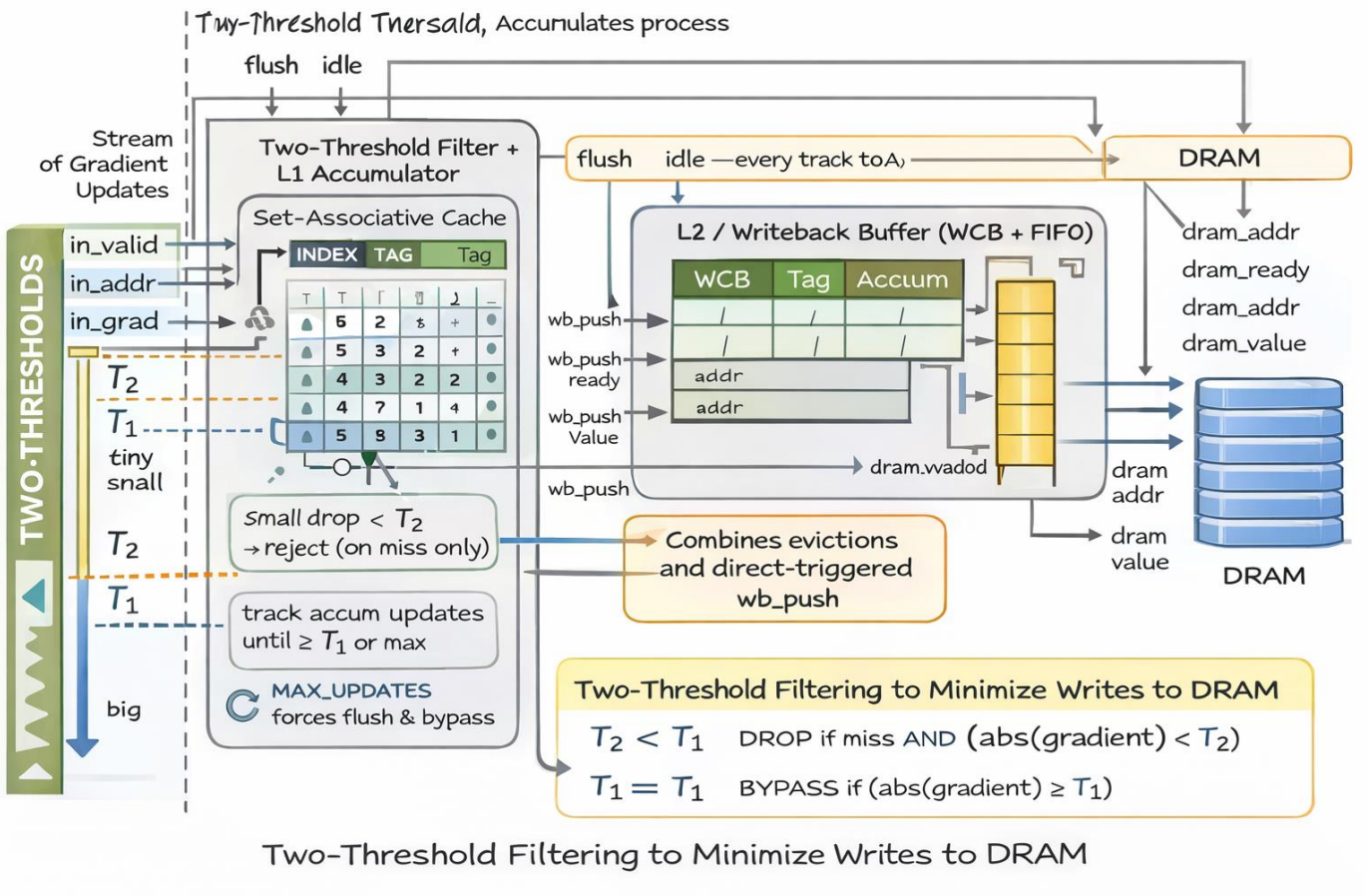
Exceptional Bandwidth Optimization:

The final Performance Report is the standout highlight. The system successfully compressed **1,276 raw input transactions (5,104 Bytes)** down to a mere **614 output writes (2,456 Bytes)**.

Core Metrics:

It achieved **a 51.88% bandwidth savings** and **a 2.08x compression ratio**. This means your accelerator successfully slashed memory/bus write traffic by more than half, which is an outstanding achievement for a gradient processing system.

II. Dual Threshold Gradient Compressor



Two-Threshold Filtering to Minimize Writes to DRAM

Core Module

This design a second (smaller) threshold before updates enter the L1 accumulator, so tiny gradients are dropped early, reducing L1 pollution / eviction

- **L1 accumulation buffer (set-associative)**
- **L2 writeback buffer (WCB + FIFO)**
- **Two-threshold policy**

  T1 = THRESHOLD: big gradients bypass L1 and go directly to L2.
  T2 = SMALL_THRESHOLD (smaller): **drop tiny gradients on MISS only** to prevent L1 pollution/evictions.

- **Flush barrier semantics**

Simulation Testbench

Tiny-Miss Drop Storm

Proves (your new feature): the second threshold filter works as intended: tiny gradients on MISS are dropped, preventing L1 pollution/eviction pressure.

Conflict Eviction

Proves: eviction path is correct and lossless (victim pushes occur, no corruption), including set-conflict worst case.

Direct-Trigger Flood

Proves: large gradients don’t touch L1 and are safely buffered through L2 under heavy backpressure—no packet drop, stable handshakes.

Max-Updates Hammer

Proves: “no small-gradient gets stuck forever” — your MAX_UPDATES forward-progress guarantee is functional.

Training-Like Steps

Proves: flush/idle barrier semantics are correct across repeated step boundaries (LLM training behavior).

Simulation Result

Exceptional Bandwidth Optimization:

The final Performance Report is the standout highlight. The system successfully compressed **1,206 raw input transactions (4824 Bytes)** down to a mere **549 output writes (2196 Bytes)**.

Core Metrics:

It achieved **a 54.48% bandwidth savings** and **a 2.20x compression ratio**. This means your accelerator successfully slashed memory/bus write traffic by more than half, which is an outstanding achievement for a gradient processing system.

05

Challenges & Lessons Learned

Challenges & Lessons Learned: Prompts Engineering

Target-Output Mismatch

Use structured, terminological prompts to avoid semantic confusion and model misunderstandings

Testbench Functionality

In addition to the testbench generated by the platform, add additional testbench to pass specific and stressed tests

Structure Granularity

When meticulous revision is needed, use more fine-grained prompt descriptions, such as unit level and variable name level to identify the modification

Challenges & Lessons Learned: Architecture Design

Pipeline Design

If the streamer and the accumulator could be pipelined, there will be more hazards and forward mechanism to be used. There is a trade-off, more speed or less complexity.

Single Cycle

Using single cycle might be bad for CPI but it would be better for the accuracy and might be good for LLM training because most accelerator doesn't require all algorithm. Guarantee accuracy is much more important than increasing such slightly changing speed.

Cache Transmission

Try to use the wright back and set associative memory instead of direct map and write through because the latter one might cause more address tag miss.

06

Future Work

Future Work

Future work will focus on validating the Gradient Compressor and Accumulator beyond synthetic traffic patterns and manual testbenches, and more fine-grained architecture optimization.

01 Simulation with Real Gradient Traces

02 Architectural Compatibility and integration

03 End-to-End Performance

04 3-stage pipelined structure for the threshold calculation

05 Fune-tuned threshold for FPGA and tape out

06 Meticulous revision for clock cycle corresponding to the logic functionality

TRAIN NEO BIT

THANK U

FOR WATCHING

By Feiyu Jia, Heng Pu, Lixuan Xu, Yuhan Jiang

https://github.com/fjia-xu/Cognichip-Hackathon-by-Train-Neo-Bit.git

Reference

[1] A. Gholami, Z. Yao, S. Kim, C. Hooper, M. W. Mahoney, and K. Keutzer, "AI and Memory Wall," IEEE Micro, vol. 44, no. 3, pp. 33-39, May-June 2024.

[2] B. R. Bartoldson, B. Kailkhura, and D. Blalock, "Compute-Efficient Deep Learning: Algorithmic Trends and Opportunities," Journal of Machine Learning Research, vol. 24, no. 122, pp. 1-77, 2023.

[3] A. Rebai and M. Canini, "OptimusNIC: Offloading Optimizer State to SmartNICs for Efficient Large-Scale AI Training," in Proceedings of the 5th Workshop on Machine Learning and Systems (EuroMLSys '25), 2025, pp. 176–182, doi: 10.1145/3721146.3721960.

[4] Y. Lin, S. Han, H. Mao, Y. Wang, and W. J. Dally, "Deep Gradient Compression: Reducing the Communication Bandwidth for Distributed Training," in Proceedings of the International Conference on Learning Representations (ICLR), 2018. (Note: This covers both the 4th and 7th lines in your prompt, as they are the exact same paper).

[5] D. Alistarh, T. Hoefler, M. Johansson, S. Khirirat, N. Konstantinov, and C. Renggli, "The Convergence of Sparsified Gradient Methods," in Advances in Neural Information Processing Systems (NeurIPS), vol. 31, 2018.

[6] I.-H. Li and T.-S. Chang, "Dynamic Gradient Sparse Update for Edge Training," arXiv:2503.17959, Mar. 2025. https://arxiv.org/abs/2503.17959.