

Coding test 1: DLLM KV Sparsity test

Background: Recent works have observed that diffusion LLMs break the strict dependency from autoregressive LLMs and can outperform them, emerging as a promising future paradigm for language modeling. Unlike autoregressive models that generate text sequentially, diffusion LLMs employ a noise-to-text transformation process that enables parallel token generation and bidirectional context modeling.

However, challenges remain in computational efficiency and training complexity. In this project, we will observe behavioral patterns and extract insights from recent diffusion LLMs to better understand their capabilities, limitations, and potential applications compared to traditional autoregressive approaches.

Implementation Details:

Step 1: Read the following papers:

- FastdLLM-v1: <https://arxiv.org/abs/2505.22618>
- FastdLLM-v2: <https://arxiv.org/pdf/2509.26328>
- Sparse-dLLM: <https://arxiv.org/pdf/2508.02558>
- dLLM-Cache: <https://arxiv.org/abs/2506.06295>

Step 2: Get familiar with [fastdLLM-v2](#) codebase, [dLLM-Cache](#) and [Sparse-dLLM](#) codebase.

Step 3: Apply training free dynamic cache eviction with sparse attention in Sparse-dLLM to [fastdLLM-v2-1.5B](#) and [fastdLLM-v2-7B](#)

Step 4: Produce results similar to Table 1 in Sparse-dLLM, comparing models (fastdLLM-v2-1.5B and fastdLLM-v2-7B) with and without the Sparse-dLLM techniques in terms of accuracy, throughput (tokens per second), and memory usage (GB) on the GSM8K and MATH datasets.

Step 5: After steps 3 and 4 warm up, conduct an ablation study on the delay step for the GSM8K benchmark, similar to Table 3 in Sparse-dLLM. (Experiment on the first 100 samples for 1.5B model and 7B model)

Step 6: Conduct an ablation study on the retention ratio for the GSM8K benchmark and report the accuracy under different retention ratios. (Experiment on the first 100 samples for 1.5B model and 7B model)

Step 7: Perform a sweep over the different retention ratio and delay steps across all layers, and report the accuracy, throughput, and memory usage. (Experiment on the first 100 samples for 1.5B model and 7B model)

Step 8: Apply dLLM-Cache techniques (V-verify) to [fastdLLM-v2-1.5B](#) and [fastdLLM-v2-7B](#) and report the accuracy. (Experiment on the first 100 samples for 1.5B model and 7B model)

Step 9: Given the observations from Steps 5, 6, 7, and 8, propose a method that can combine Sparse-dLLM and dLLM-Cache while preserving high accuracy. Compare the proposed method with the baseline introduced in Sparse-dLLM. (Experiment on 1.5B model)

Deliverables:

1. Code implementing Sparse-dLLM techniques and dLLM-Cache on Fast-dLLM-v2, and the experiments for following plots.
2. Report answering the following
 1. What are the differences between Fast-dLLM-v2 and LLaDA (as used in Sparse-dLLM)?
 2. What techniques are proposed in Sparse-dLLM? Do they make sense? Why or why not?

3. What techniques are proposed in dLLM-Cache? Do they make sense? Why or why not?
4. Results with and without Sparse-dLLM techniques in terms of accuracy, throughput, and memory on the GSM8K and MATH datasets, along with summarized observations.
5. Ablation study on the delay step for the GSM8K dataset, with summarized observations.
6. Ablation study on the retention ratio for the GSM8K dataset, with summarized observations.
7. Report Step 8 results
8. Report Step 9 results

Coding test 2: DLLM Quantization test

Background: Recent works have observed that diffusion LLMs break the strict dependency from autoregressive LLMs and can outperform them, emerging as a promising future paradigm for language modeling. Unlike autoregressive models that generate text sequentially, diffusion LLMs employ a noise-to-text transformation process that enables parallel token generation and bidirectional context modeling.

However, challenges remain in computational efficiency and training complexity. In this project, we will observe behavioral patterns and extract insights from recent diffusion LLMs to better understand their capabilities, limitations, and potential applications compared to traditional autoregressive approaches.

Implementation Details:

Step 1: Read the following papers:

- Fastdilm-v1: <https://arxiv.org/abs/2505.22618>
- Fastdilm-v2: <https://arxiv.org/pdf/2509.26328>
- QDLM: <https://arxiv.org/abs/2508.14896>

Step 2: Get familiar with [fastdilm-v2](#) codebase, and [QDLM](#) codebase.

Step 3: Apply in GPTQ, AWQ, SmoothQuant, QuaRot, DuQuant to [fastdilm-v2-1.5B](#) and [fastdilm-v2-7B](#)

Step 4: Produce results similar to Table 3 and Table 4 in QDLM (same experiment settings but with fastdilm-v2 models, 1.5B and 7B)

Step 5: Perform weight-only FP8 NVFP4 quantization (with FP16 activations) on GSM8K and report the accuracy (1.5B and 7B).

Step 6: Perform weight-activation FP8/FP8 and NVFP4/NVFP4 quantization on GSM8K and report the accuracy (1.5B and 7B).

Step 7: Visualizations of activation outliers in Fastdilm-v2-1.5B and Fastdilm-v2-7B (similar to Fig. 2 in QDLM)

Step 8: Given the observations from Steps 5, 6, and 7, propose a method that can dynamically determine, for each layer, the best bit precision to maintain the accuracy while increasing the throughput. The method can be either training-free or based on fine-tuning. Compare the proposed method with the baseline introduced in QDLM. (Experiment on 1.5B model)

Deliverables:

1. Code implementing QDLM techniques on Fast-dLLM-v2, and plots.
2. Report answering the following

9. What are the differences between Fast-dLLM-v2 and LLaDA (as used in Sparse-dLLM)?
10. What techniques are used in QDLM? Do they make sense? Why or why not?
11. Report Step 4 results
12. Report Step 5 results
13. Report Step 6 results
14. Report Step 7 results
15. Report Step 8 results

Coding test 3: FFN CUDA optimization

- **Overview:** FFN is often used in LLMs and is very important. Design a CUDA kernel that optimizes FFN computation (see [this](#)).
- **Requirements:**
 - The hidden size (4096) and intermediate size (12288) are fixed in this coding test.
 - Write a CUDA kernel that takes an input x of dimension $[B, 4096]$, where B is the batch size and can be one of $\{4, 8, 16, 32, 64, 128\}$, and performs a GEGLU_FFN as shown in [this](#).
 - The output should be verified for correctness.
 - Achieve at least a 3x speedup across all batch sizes on A5000 through kernel [optimizations](#).
- **Deliverables:**
 - A complete test suite, including correctness tests, the CUDA kernel, and a speedup comparison.
 - A report describing how you approached and completed the coding test.

Coding Test 4: NxN Systolic Array Design & Evaluation

- **Overview:** Design a NxN Systolic Array in Verilog, write the testbench and evaluate the PPA (power, performance, and area)
- **Requirements:** Your design should contain the following parts: a systolic array that supports fix-point 16-bit multiplication & accumulation, two memory blocks that feed the data into the array, a controller that controls the memory and the systolic array, one memory block to store the output data, and one memory block to store the instruction.
 - The top IO of the design should be
 - clk, rst (rst_n if targeting ASIC)
 - addrA, enA, dataA (for write data memory A)
 - addrB, enB, dataB (for write data memory B)
 - addrI, enI, dataI (for write instruction memory I)
 - addrO, dataO (for read result memory O)
 - ap_start (pulse signal)
 - ap_done (level signal)
 - Instruction format
 - The instruction stream is a sequence of integers. Each group of three consecutive nonzero integers specifies one matrix–matrix multiplication (MMM): $[a, b, c, \dots]$ corresponds to:
 - Input matrix: $a \times b$
 - Weight matrix: $b \times c$

- Output matrix: $a \times c$
- The integer 0 marks the end of the instruction sequence (no more MMMs).
- Example. Given the instruction sequence: [32, 16, 64, 8, 16, 0]. This should be interpreted as three chained MMMs:
 - First MMM
 - Input: 32×16
 - Weights (MLP1): 16×64
 - Operation:
 - $(32 \times 16) \times (16 \times 64) \text{ (MLP1)} \rightarrow 32 \times 64$
 - Second MMM
 - Input: 32×64 (result of previous step)
 - Weights (MLP2): 64×8
 - Operation:
 - $(32 \times 64) \times (64 \times 8) \rightarrow 32 \times 8$
 - Third MMM
 - Input: 32×8 (result of previous step)
 - Weights (MLP3): 8×16
 - Operation:
 - $(32 \times 8) \times (8 \times 16) \rightarrow 32 \times 16$
- The final result of this instruction sequence is a matrix of size 32×16 , which must be stored back to data memory.
- For the data transfer between the memory and array, both the input and output data should be systolically propagated, i.e., you can not use a large mux just to pick which PE among all the PEs to send the data directly to.
- You should design your own test data generator. One recommended way is to use MATLAB to generate random matrices and use the “fi” command to generate a fixed representation.
- **Testbench guidelines:** In the Verilog testbench, you should first write data to the data memory and the instruction memory. **You should at least test the instruction sequence [32, 16, 64, 8, 16, 0] in the instruction memory and run it on the systolic array with N=4,8.** After the input data and memory are loaded, use ap_start to start your design. After getting the ap_done signal, print the output data from the memory to a file and compare the results.
- **Bonus:** Design an FP16 PE and do the flow again.
- **Deliverable:** After the task, the source code and the documentation, which have enough information to explain the results and implementation flow, should be provided for assessment.

Design source:

- Python / C / MATLAB file to generate fixed point inputs
- Verilog design & testbench
- Result evaluation program

Documentation

Coding Test 5: SOTA AI accelerator arch-level reproduction

- **Overview:** This test involves reproducing a state-of-the-art AI accelerator using Verilog, aiming to achieve the closest possible resemblance.

- **Requirements:**
 - **Design Approach:**
 - Adopt a top-down approach in designing the system architecture.
 - Start with a block diagram to outline the system and define specifications for each module.
 - Proceed with detailed implementation for each module, before integrating them into the complete system.
 - **Technical Specifications:**
 - Use Hardware Description Languages (HDL) such as Verilog or SystemVerilog, or opt for hardware performance modeling using C/Python.
 - Your design must include the following:
 - Algorithm compilation that involves mapping and scheduling tasks on the hardware.
 - RTL design or performance modeling.
 - Testing and evaluation of the design.
 - **Project Scope:**
 - Completion of the project before the meeting is not required.
 - Emphasis is on your thought process, architectural approach, and future completion plans.
 - **Extendability requirements:**
 - If we were to use the design to profile some customized workloads, e.g., another transformer model not benchmarked by the work, we would have the ways to know the hardware performance
 - If we were to make modifications to the original hardware in the paper, we could base the modifications on your design
- **Choices of AI Accelerators for Reproduction:**
 - [Ditto: Accelerating Diffusion Model via Temporal Value Similarity](#)
 - [S-DMA: Sparse Diffusion Models Acceleration via Spatiality-Aware Prediction and Dimension-Adaptive Dataflow](#)
 - [SpAtten: Efficient Sparse Attention Architecture with Cascade Token and Head Pruning](#)
 - [SIGMA: A Sparse and Irregular GEMM Accelerator with Flexible Interconnects for DNN Training](#)
- **Deliverable:**
 - **Source Code and Documentation:**
 - All source files for design, testing, simulation, and evaluation.
 - Comprehensive documentation that includes:
 - Detailed implementation plan and process.
 - Instructions for potential users on how to run or modify the design.
 - Clear delineation of completed and pending tasks.
 - A roadmap for future work to complete the design.