CalHacks Write Up

Rithwik Nukala, Kinjal Govil, Anikait Paliwal, Achyut Chebiyum

October 2025

1 Abstract

As LLMs become widely used for various tasks ranging from writing papers to solving complex math problems, it becomes more important to ensure the user gets accurate responses as quickly as possible as well as context being effectively passed from the beginning to end of the conversations. In this write up, we propose various strategies for both inference time speed ups such as a combination of pruning+SVD and quantization as well as Chain-Of-Thought memory improvements.

2 Inference Time Improvements

2.1 Dynamic Pruning + SVD

We created a novel pruning algorithm which first chunks the weight vector into 15 parameter pieces. We then randomly generate 500 to 1000 pruning masks which we will apply onto each chunk one at a time.

Each mask is applied *temporarily* to its corresponding chunk, and the model's performance (e.g., perplexity or loss) is immediately evaluated on a small, held-out calibration dataset. This evaluation yields a performance score for each of the 500-1000 candidate masks.

After all masks for a given chunk have been scored, we identify the 'Top-K' (e.g., Top 20) best-performing masks. A final, permanent mask for that chunk is then generated by finding the *consensus* among this top cohort. Specifically, we take the *intersection* of all weights that were pruned (set to zero) by this top group. A weight is only permanently pruned if all 20 of the best-performing masks agreed it should be removed.

This "stochastic consensus" process is repeated independently for every parameter chunk, and the resulting chunk-specific masks are combined into a single global pruning mask for the entire model.

Finally, after the model is pruned, we apply Singular Value Decomposition (SVD) to the newly sparse weight matrices. Each pruned layer's weight matrix is replaced by a dense, low-rank approximation (e.g., using a rank of 4). This SVD step acts as a "re-densification," aiming to recover performance by finding

the most optimal low-rank representation of the pruned layer, effectively "filling in the blanks" left by pruning in a structured and efficient manner.

2.2 Quantization

The quantization is done through the HardwareOptimizer class, which provides a unified framework for hardware-aware neural network optimization, facilitating efficient model deployment across diverse computational backends, including GPUs, CPUs, ONNX Runtime, and TensorFlow. The framework automatically detects the target device's capabilities to determine optimal numerical precision, applying mixed-precision techniques (FP16 or BF16) on supported architectures to enhance computational throughput and memory efficiency. It integrates multiple compression strategies, such as structured channel pruning and quantization, to reduce model size and improve inference performance, although we obtained better results with a custom pruning algorithm mentioned in the previous section. The quantization module supports dynamic, static post-training, and quantization-aware training (QAT) schemes, leveraging backend-specific kernels such as FBGEMM and QNNPACK to ensure compatibility and accuracy. Furthermore, the framework supports cross-platform deployment by exporting models to ONNX format and provides an interface for TensorFlow or TFLite conversion, promoting interoperability across edge and cloud environments. The HardwareOptimizer also offers model introspection tools to report key metrics, including parameter count, model size, and precision type, thus serving as a comprehensive system for adaptive, hardware-informed model compression and deployment.

3 Chain-Of-Thought Management + Improvement

We take inspiraton from *Less is More: Recursive Reasoning with Tiny Networks*. The basic idea is to train a smaller model that recursively improves the prompt and feeds that back into the larger model so that the model generates a more informed response. We started by training a 70M parameter multi-head attention model with KV-caching from scratch using a dataset of 1.4B tokens which we got from the TinyStories and SlimPajamas datasets. The small model improves generation by managing model memory during Chain of Thought generation. Every few reasoning steps in Chain of Thought generation the model takes in the prompt and the reasoning and condense them into a more refined prompt. Repeatedly doing this allows the model to refine it owns prompt while also reducing context size. This reduction in context size improves inference performance. Additionally the prompt refinement boosts response quality to compensate for the aggressive model compression techniques we implemented to improve inference performance. While the pruning and quantization discussed earlier is optimized to maintain model performance there is no such thing as

perfect compression, so this Chain of Though Context Management makes up for the lost response quality.

Algorithm 1 Hybrid Model Inference Loop

Require: $P_{initial}$: The initial prompt text.
Require: M_L, T_L: The large model and its tokenizer.
Require: M_S, T_S: The small model and its tokenizer.
Require: N_{loops}: Number of hybrid iterations (e.g., 4).
Require: G_{max}: Max CoT groups to generate (e.g., 3).
Ensure: P_{final}: The final generated text.

procedure HybridInference($P_{initial}, M_L, T_L, M_S, T_S, N_{loops}, G_{max}$)
  $P_{current} \leftarrow P_{initial}$
  for $i = 1 \rightarrow N_{loops}$ **do**
  — *Step 1: Large Model CoT Generation* —
    $I_L \leftarrow$ Tokenize($T_L, P_{current}$)
    $T_{cot\_full} \leftarrow$ GenerateWithCoT(M_L, I_L, G_{max})
    $T_{cot\_only} \leftarrow$ ExtractNewText($T_{cot\_full}, P_{current}$)
  — *Step 2: Small Model Response Generation* —
    $P_{combined} \leftarrow P_{current} + T_{cot\_only}$
    $I_S \leftarrow$ Tokenize($T_S, P_{combined}$)
    $T_{small\_full} \leftarrow$ GenerateResponse($M_S, I_S, 100$)
  — *Step 3: Update Context for Next Loop* —
    $P_{current} \leftarrow T_{small\_full}$
  end for
  — *Final Generation* —
  $I_{final} \leftarrow$ Tokenize($T_L, P_{current}$)
  $P_{final} \leftarrow$ GenerateResponse($M_L, I_{final}, 512$)
  return P_{final}
end procedure

3.1 Pre Training

Since this model is very small we need to very carefully choose our datasets so it does not just output gibberish. The TinyStories dataset was generated using a very limited vocabulary with simple words and structures. This stops the model from being overloaded with large volumes of complex data it is not scaled to handle. It has been shown to make even models in the 1-10M parameter range coherent and gramatically sound. The second dataset we used is SlimPajamas a carefully curated subset of the RedPajamas dataset the Llama models are trained on. Focusing on the highest quality data only. This dataset is till over 600B tokens. We were looking to follow the Chinchilla Scaling Law so for our 70M parameter model we wanted 1.4B tokens. TinyStories is roughly 400M

tokens so we randomly sampled 1B tokens from SlimPajama and used that to add general knowledge to the language cohesion TinyStories would provide.

3.2 Post Training

In LLM chatbots their post training involves fine tuning to make the model properly respond as if in conversation and properly format responses to questions. We need to do fine tuning so the model learns how to structure its generation to effectively condense and optimize the model context. To do this we used the Open Thoughts datatset. It contains Qwen3-32B responses to various STEM focused prompts. The model was also given a system prompt to make very large reasoning chains making it ideal for us to fine tune our model which needs to learn how to process Chain of Throught reasoning into memory. Since the model we are doing backprop on is separated from the model producing the output tokens, we used REINFORCE policy gradient to evaluate the loss for weight updates.

4 Results

4.1 Inference Time Optimizations

We noticed a difference between when plugging in our laptop versus not. The laptop that we ran the results on is a M2 Mac book Air. We attribute these differences to the power optimizations built into the M2 chip that force less power to go to certain computations when not plugged in.

4.2 Table with Results

Optimization	**Gemma3-270M Speedup**	**Qwen3-8B Speedup**
Quantization	1.74x	0.98x
Pruning+SV	1.65x	1.07x
Q $\Rightarrow$ P+SV	1.66x	N/A
P+SV $\Rightarrow$ Q	1.31x	N/A

5 Future Steps

Currently, the focus is on training the QWEN model; however, there is an ongoing effort to expand toward a broader and more diverse dataset to enable native handling of multiple data sources. At present, quantization relies on PyTorch's built-in CUDA device detection, but future work aims to incorporate an external API for querying detailed hardware specifications to achieve more accurate and adaptive optimization decisions. Additionally, a more rigorous statistical framework is planned for evaluating and validating the performance of pruning algorithms, ensuring that compression techniques are not only hardware-efficient but also statistically robust across different model architectures and datasets.