Large-Scale AI Engineering Project Report

Mamba, DeltaNet and Torch DDP

Samuel Stante, Martin Wertich, Jonas Lill

{sstante, mwertich, jolill}@ethz.ch

December 2025

1 Introduction

The goal of this project was to expand the second assignment of the Large-Scale AI course at ETH Zurich with implementations of the DeltaNet [4, 6] and Mamba [3, 2]. Besides implementation, we also compared these two architectures with the standard Transformers [5], which served as our baseline. Additionally, we expanded the assignment with distributed data parallel (DDP) and gradient checkpointing options to fully utilize the available compute. Our code and instructions to reproduce the results on Clariden are available on GitHub: `https://github.com/Timisorean/large-scale-ai-project`.

We started with the assignment code and did some cleanup and refactoring first. For example, we integrated Weights & Biases[1] to monitor and evaluate runs. We also set up Hydra[2] configurations to easily adapt hyperparameters and to start runs quickly over the command line. Regarding Hydra, we also tried to integrate the Submitit plugin[3] to optimize Slurm job submission, for example, for sweeps. Unfortunately, it is non-trivial to do so in the distributed setting, and it is also difficult when benchmarking with NVIDIA Nsight Systems[4].

In addition to these changes, we also switched from the provided container image to a virtual environment managed by uv[5]. This makes managing dependencies easier, enables better debugging, and improves integration when developing remotely with Visual Studio Code. For example, when requiring NVIDIA Nsight Systems, we can just use an existing uenv (like `prgenv-gnu/25.6:v2`) of the cluster.

2 Implementations

2.1 Torch DDP

For integrating DPP support, we mostly relied on the code of the other assignments. Additionally, we had to adjust the dataset iterator, implement printing logic that supports multi-rank setups, and ensure deterministic initialization across all ranks.

Even though we strictly followed the other assignments, our distributed setup did not work with multiple nodes initially. Debugging this was very hard as we did not get a real error message.

[1] `https://wandb.ai/`

[2] `https://hydra.cc`

[3] `https://hydra.cc/docs/plugins/submitit_launcher`

[4] `https://developer.nvidia.com/nsight-systems`

[5] `https://docs.astral.sh/uv/`

Only after toggling several debug flags did we understand that the underlying distributed logic timed out. However, this only helped that much. After many hours, we figured out that changing the uenv resets some important environment variables required for distributed computing. The CSCS documentation[6] has a small entry about this.

2.2 Mamba

Initially, we implemented Mamba using a simple approach based on [3, 2] in pure PyTorch. However, the sequential execution of each update step took a long time without the use of fused kernels or other optimization tricks. We tried to get a comparison benchmark using the official library[7], which contains a highly optimized version of the model. Despite reasonable efforts, we were not able to link the required libraries for the Triton kernels. The issue was a combination of the AArch64 architecture and the available packages in the uenv.

In order to further improve our pure PyTorch implementation, we implemented a chunked variant of the Mamba used in our sequence mixer. The underlying recurrence remains unchanged. To improve memory locality and enable long-sequence streaming, we divide the sequence into fixed-size chunks of length C and process them sequentially. Within each chunk, we perform the same timestep updates. Chunking reorganizes computation without altering the SSM itself, providing a more memory-efficient execution path, and enabling future integration of fused kernels or streaming operation.

2.3 DeltaNet

We implemented the DeltaNet attention based on [4, 6], using the ELU + 1 function as the kernel function ϕ and using simple normalization. However, this implementation was very slow due to the sequential updates. To speed this up, we again tried a chunked version, where we divided our inputs into chunks of size C and computed the sequential state updates on each chunk. The idea was to improve memory locality, and it did speed up the runtime, but not by a significant enough margin.

Unfortunately, we were not able to implement the exact DeltaNet computations at a reasonable speed. Therefore, we decided to implement an approximation of the DeltaNet Attention by using the concept of Dual Chunked Attention [1]. We again split the input into chunks of size C. Working on one chunk at a time, we split the output into two parts: the inter-chunk attention and the intra-chunk attention. For inter-chunk attention, we compute the DeltaNet state updates for each chunk in parallel. In order to capture long-range dependencies, we also calculate the inter-chunk attention using standard attention between different chunks. The final output is the sum of the intra and inter-chunk attention.

3 Experiments

Since DDP and the different model architectures are somewhat orthogonal in what they try to accomplish, we first evaluate the models without considering DDP and then evaluate DDP separately. We fix all model configurations to ensure a fair comparison (e.g., roughly the same model size, . . .). All used configurations are listed in table 1. Model weights are in `bf16`.

[6]`https://docs.cscs.ch/software/communication/nccl/#using-nccl`
[7]`https://github.com/state-spaces/mamba`

Table 1: Model configurations for our experiments were chosen to ensure a fair comparison between the different models. **Notation:** L: number of layers; d_{model}: embedding dimension; H, H_{kv}: number of query and key-value heads (GQA); d_{state}: SSM state dimension; d_{conv}: local convolution width; d_{head}: head dimension; Δ, A: time-step and matrix initialization ranges; FFN $\times$: feed-forward dimension multiplier; Chunk: parallel scan chunk size.

Model	Backbone	Attention / State	MLP / Conv	Model Size
Transformer (Baseline)	$L = 32$ $d_{model} = 2560$	$H = 32, H_{kv} = 8$ RoPE $\theta = 5\text{e}5$	FFN $\times 1.3$	3.5B
DeltaNet	$L = 32$ $d_{model} = 2560$	$H = 32, H_{kv} = 8$ $\beta =$ True, Chunk $= 64$	FFN $\times 1.3$ Conv $= 4$	3.7B
Mamba	$L = 32$ $d_{model} = 2560$	$d_{state} = 64, d_{head} = 64$ $d_{conv} = 4$, Chunk $= 64$ $A \in [1, 16], \Delta \in [1\text{e-}3, 0.1]$	Expand $= 2$	3.2B

We evaluate the models based on their train loss and resource utilization. The goal is to minimize loss as much as possible within a fixed number of training steps for fixed model configurations. This should capture the training efficiency.

Besides the model parameters, we use gradient clipping to 1.0, a `LambdaLR` scheduler with a warm-up of 10 steps, and sequence lengths of 2048 in combination with the data loader implemented in assignment two of the course. We chose a batch size of 16. Using larger batch sizes (e.g., 24) causes out-of-memory errors. Similar to assignment two of the course, we employ the Adam Optimizer. A sweep over the learning rate (1e−3, 1e−4, 1e−5) revealed 1e−4 to work best for all model types.

3.1 Evaluating the models

Figure 1 shows the train loss and the tokens per second of our runs, smoothened with an exponential moving average of 0.3. The models were trained for a total of 1000 steps and converged to similar loss values. Although Mamba and the DeltaNet achieve better theoretical runtime efficiency by avoiding the quadratic runtime of the classical Transformer's attention mechanism, in our experiments, they still fall short.

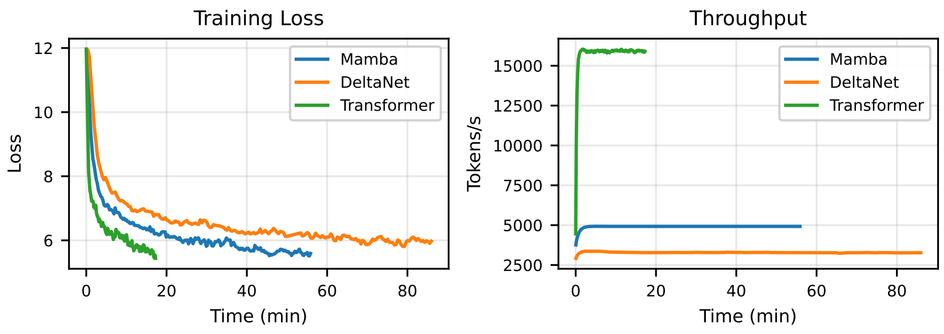

Figure 1: Training loss and throughput comparison of Transformer, Mamba, and DeltaNet architectures on a single GPU with sequence length 2048. Loss is smoothed using an exponential moving average ($\alpha = 0.3$).

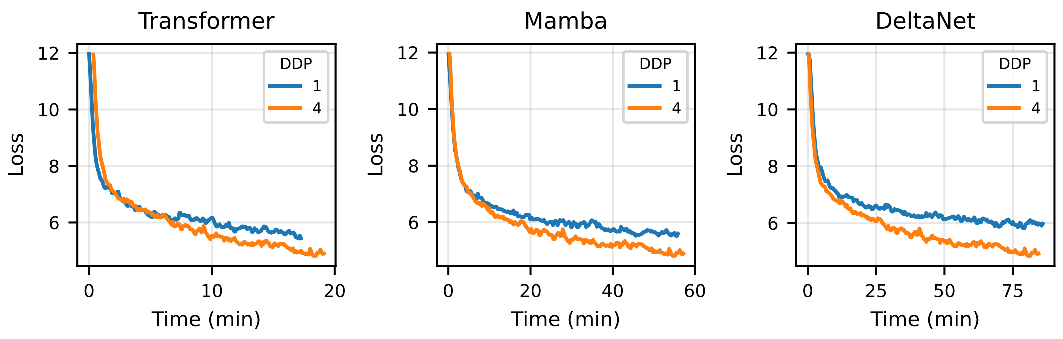

Figure 2: Scaling behavior with DDP for sequence length 2048. Learning rate is scaled linearly with the number of GPUs. Loss is smoothed using exponential moving average ($\alpha = 0.3$).

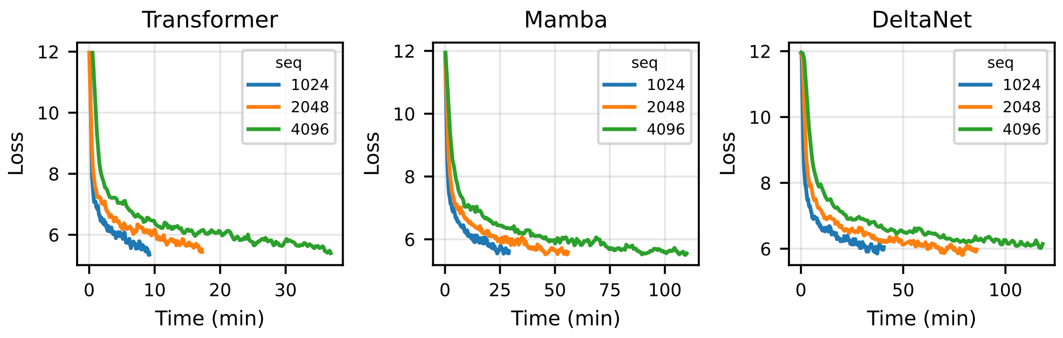

Figure 3: Effect of sequence length on training loss for each model architecture (single GPU). Loss is smoothed using an exponential moving average ($\alpha = 0.3$).

The throughput plot shows that Transformer outperforms the other two models in processing speed. The Mamba model also has a better throughput than the DeltaNet. This result is unexpected and goes against the theory. We address possible reasons in our conclusion.

3.2 Evaluating DDP

The nodes on the Clariden cluster always have four GPUs, which makes their utilization through DPP extremely important. We also evaluated our DDP implementation. Figure 2 shows the loss curves of each model comparing training runs with 1 and 4 GPUs. We verified that DDP works as expected for all three models.

3.3 Ablation Sequence Length

Another interesting point is how the model scales with sequence lengths. Figure 3 shows the loss for all three model types for sequence lengths 1024, 2048, and 4096. They all arrive at approximately the same loss value at 1000 training steps. We can also nicely see how the performance scales.

4 Conclusion

The Transformer is, despite our best efforts with the other model types, still the best-performing model. Possible reasons include missing fused kernels and the need for further optimization to improve Mamba's and DeltaNet's runtime performance. In contrast to the Transformer, the Mamba and DeltaNet implementations are bottlenecked by sequential kernel launches and unoptimized chunking operations. The Transformer benefits from hardware-aligned primitive calls and fewer Python-to-GPU kernel dispatches in the forward/backward calls. Setting up the Trition kernels would allow us to verify whether Mamba and DeltaNet can beat the Transformer. However, further optimizations are currently out of the scope of this project. We verified that DDP works across all three models and improves runtime, allowing us to utilize all 4 GPUs per node.

Note: We used GitHub Copilot for assisted coding and Grammarly for this report.

References

[1] Chenxin An et al. *Training-Free Long-Context Scaling of Large Language Models*. 2024. arXiv: `2402.17463 [cs.CL]`. URL: `https://arxiv.org/abs/2402.17463`.

[2] Tri Dao and Albert Gu. *Transformers are SSMs: Generalized Models and Efficient Algorithms Through Structured State Space Duality*. 2024. arXiv: `2405.21060 [cs.LG]`. URL: `https://arxiv.org/abs/2405.21060`.

[3] Albert Gu and Tri Dao. *Mamba: Linear-Time Sequence Modeling with Selective State Spaces*. 2024. arXiv: `2312.00752 [cs.LG]`. URL: `https://arxiv.org/abs/2312.00752`.

[4] Imanol Schlag, Kazuki Irie, and Jürgen Schmidhuber. *Linear Transformers Are Secretly Fast Weight Programmers*. 2021. arXiv: `2102.11174 [cs.LG]`. URL: `https://arxiv.org/abs/2102.11174`.

[5] Ashish Vaswani et al. *Attention Is All You Need*. 2023. arXiv: `1706.03762 [cs.CL]`. URL: `https://arxiv.org/abs/1706.03762`.

[6] Songlin Yang et al. *Parallelizing Linear Transformers with the Delta Rule over Sequence Length*. 2025. arXiv: `2406.06484 [cs.LG]`. URL: `https://arxiv.org/abs/2406.06484`.