APPENDIX

A PRE-TRAINING RESULTS

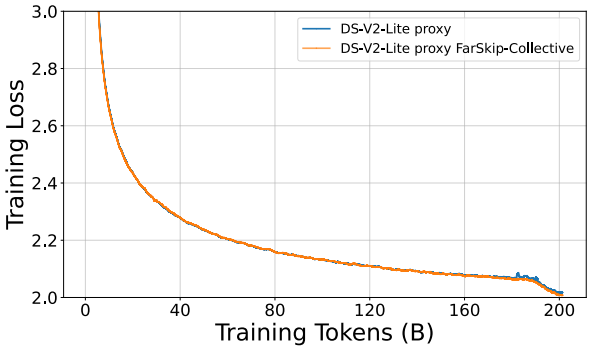

Figure 10: MoE pretraining loss with regular and FarSkip-Collective architectures. We pretrain 16B parameter MoE following the DeepSeek-V2-Lite model architecture for 200B tokens. We observe FarSkip-Collective closely matches the loss of the regular model.

We test the pre-training of the FarSkip-Collective architecture by pre-training an MoE model from scratch using the DeepSeek-V2 Lite model architecture and configuration (64 routed-experts) in Megatron-LM. We train the model on 200B training tokens randomly sampled from a larger corpus of high quality mix of publicly available data using a sequence length of 4096 tokens and a batch size of 4,194,304 tokens. With 3B active parameters, this corresponds to 4x simplistic Chinchilla scaling estimate with respect to the model's active parameters (Hoffmann et al., 2022). We repeat the training with the exact settings and seed but with FarSkip-Collective turned off as the regular MoE baseline. We observe the loss curves of both models matches closely with the FarSkip-Collective model achieving a final training loss of 2.006 as compared with 2.016 averaged over the last 50 steps. For the evaluation we use the same benchmarks as the distillation experiments and observe the two models performance is on par with an average performance gap of FarSkip-Collective of 0.3%, achieving 51.2% vs the regular MoE averaged at 51.5% over the datasets. This result at 200B token scale with a 16B-A3B model further reinforces the viability of the FarSkip-Collective MoE model modification in addition to the larger-scale self-distillation results in the main paper.

B THEORETICAL PERFORMANCE MODELING OF FARSKIP-COLLECTIVE

We develop a theoretical analytical model to analyze the FarSkip-Collective architecture under different scenarios based on hypothetical accelerator performance limits that isolate implementation details such as optimized kernels and fused operations etc. For this we consider a hypothetical setup of a fake accelerator with the following performance specifications. For computation we assume the dense computations run at 500 TFLOPS for attention and 600 TFLOPS for GEMMs. For communication we assume a GByte/s bus bandwidth for all-reduce of 400 and 300 for all-to-all collectives. We choose relatively smaller compute TFLOPS values for main operations to account for non-dense operations that tend to bring down the cumulative compute utilization estimates. With this set-up we model the prefill stage of inference which remains in the "compute-bound" and "communication-bandwidth-bound" regimes making the performance analysis with these specifications possible. For these settings we consider an 8-accelerator node setup and use analytical FLOP usage calculations[2] to compute end-to-end run-times of different pieces of the network execution. We compute the estimated runtimes of different components in the model including computation and communication calls based on the proposed limits and the model dimensions specifications. We then compute the regular MoE architecture and FarSkip-Collective architecture by overlapping parts of the communication runtime with the relevant computation components that it can be overlapped with. We summarize the calculation approach for estimated exposed communication runtime for each communication call in the FarSkip-Collective and Regular settings in Tab. 5. In our analysis we also assume perfect expert load-balancing per GPU and consider TP=8 EP=8 model parallelisms. We run our analysis for a DeepSeek-V3 architecture for which we vary the levels of sparsity. For the prefill request, we assume BS=32 and context size of 2048.

Below we describe how we model different levels of sparsity for a DeepSeek-V3-like architecture. Recall, DeepSeek-V3 activates 8 + 1 (routed + shared) experts out of 256 + 1

[2]We estimate FLOPs using the methodology in github.com/AI-Hypercomputer/maxtext/src/MaxText/maxtext_utils.py#L454 but adapt the calculation for prefill inference (no backward)

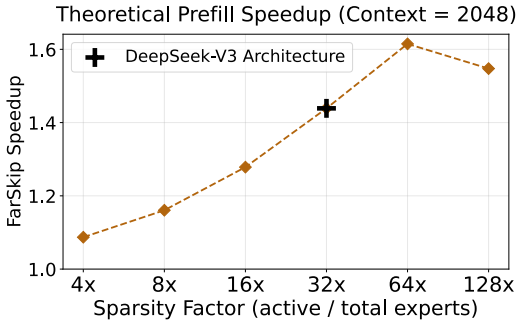

Figure 9: (reproduced from main) Performance model of FarSkip-Collective prefill speedup under different sparsity levels applied to a Deepseek-V3 architecture.

Table 5: Communication-collective operations estimated runtime under FarSkip and Regular MoE settings. Under FarSkip, collectives can be overlapped with other computations and only the exposed communication latency contributes to the runtime.

Communication Collective	Settings	Contributed Runtime	Description
Attention & Shared all-reduce	FarSkip	`min(0, as.comm - routed.runtime)`	Attention and shared-expert all-reduce is combined into a single call, overlapped over the routed-expert computation, needed as input in the next attention layer.
Routed Expert all-reduce	FarSkip	`min(0, routed.comm - attn.runtime)`	Routed-experts are first used in the shared-expert of the next layer and can be overlapped with attention computation.
Attention all-reduce	Regular	`attn.comm`	Attention all-reduce happens sequentially after the attention layer.
MLP all-reduce	Regular	`routed.comm`	The all-reduce of the routed and shared experts activations happens sequentially after the MLP layer.

available experts. To modify the sparsity level we consider the scenario of more refined experts but still using 8 + 1 experts activated for each token. For example to model an architecture that is 2x sparser we double the number of experts, while halving the expert dimension, and maintain the number of active experts 8 + 1 (routed + shared). This is realistic as it enables diversity in expert selection while refining the expert size and improving sparsity as seen in recent works such as Kimi-K2 architecture (Kimi et al., 2025) which has 1.5x more experts than DeepSeek-V3. With this approach to modeling sparsity we can cleanly maintain the total parameter count (neglecting minimal differences due to the router-gating parameter) while halving the FLOP requirements of token processing of the activated experts. We reproduce Fig. 9 below for convenience and observe that FarSkip-Collective provides larger speedups for models that are 2x or even 4x sparser than DeepSeek-V3 which at baseline already independently exhibit larger throughputs due to reduced computations.

We would like to note the limitations of the performance analysis we provide. First we do not properly account for non-dense operations and model all of the dense operations with a simplistic GEMM vs. attention distinction to FLOPS. This does not take into account the potentially-reduced FLOP utilization from smaller GEMMs of more refined experts or non-optimal hardware model dimensions. With regards to FarSkip-Collective speedup, slower GEMMs in the extremely sparse regime will actually lead to continued performance gains as compared to the "drop-off" observed at the 128x sparsity level and we should expect a later drop-off as compared with the simple performance model. Lastly we do not account for potential reduction in compute efficiency that can result from communication overlapping due to competing hardware resources.

In Fig. 11 We consider the effects of different hypothetical hardware/algorithmic settings such as increased or reduced computation or communication speeds. In particular we

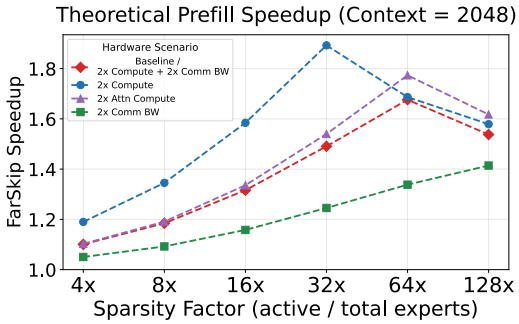

Figure 11: Theoretical speedup of FarSkip-Collective prefill under different compute and communication scenarios for a range of sparsity levels modified from the DeepSeek-V3 architecture (original model @ 32x sparsity factor).

evaluate the scenarios of 1) 2x faster computation, 2) 2x faster attention computation (e.g. suppose in the very sparse settings the small shapes of the MoE multiplication lead to lower utilization compared with attention) 3) 2x faster communication and 4) 2x faster communication and 2x computation. For the compute-bound, communication-bandwidth-bound model the "2x compute + 2x communication" scenario coincides with the baseline scenario as everything scales down proportionally while absolute speeds double in both the FarSkip-Collective and baseline models.

C Optimized Training: Overlapped Backward Call Implementation

In this section we provide more details on our communication-computation overlapped implementation of FarSkip training, specifically how we achieve computation-communication overlapping in the backward pass using two techniques. Recall, in the MoE layer with EP all-to-all communication collectives appear in the Dispatch and Combine operations within the MLP block. The gradients of the

Dispatch and Combine operations are themselves all-to-all collectives which we would like to overlap. The issue is that with autograd, one does not control the node traversal order in the backward pass, risking a race condition or no overlap at all. Instead we propose two innovative techniques that put together, enable us to achieve overlap cleanly. 1) we implement an async-safe all-to-all custom autograd function that triggers a synchronization before future processes access its output tensors which avoids race conditions. and 2) we use the Sequence Numbers in PyTorch's internals API to reprioritize node traversal of autograd processing in order to delay the synchronization triggers for as long as possible to ensure sufficient time for communication to run and overlap.

To implement the async-safe all-to-all custom autograd function we create an autograd class for async-all-to-all with a stateful dictionary that stores both the forward and backward communication handles produced by async communication in PyTorch. During the forward pass, the forward all-to-all handles are being generated by the collective in async mode; while the backward-all-to-all communication handles do not exist yet but they have a dedicated keys in the layer's stateful dictionary. When `backward()` is called on the operator, it runs backward via another all-to-all collective call in async mode and returns a communication handle which we use to populate the state dictionary. This enables us to store and later access the backward handle while not explicitly calling the backward function of the async-all-to-all that gets called implicitly by autograd. We then implement a backward-hook that takes as input the async-safe autograd function object along with its stateful dictionary and will trigger synchronization of the backward handle when the hook fires. We then attach the hooks to the input tensors of the all-to-all operators via PyTorch's `register_full_backward_hook`. At the time we register the hook, the state dictionary is empty but before the hook will be fired async-all-to-all backward will take place and the backward handle keys will be populated. The hook + stateful dictionary approach ensures that we can implicitly pass communication handles to the hook and trigger a synchronization of the handles before the processing of any gradients that will use the tensors produced by the async-all-to-all in the backward call. Overall this makes it possible to run the all-to-all communication in the backward pass asynchronously while ensuring the relevant gradients are ready when accessed.

Nonetheless async-safe all-to-all gradient is not sufficient to achieve significant overlapping of the backward pass. As we are optimizing overlap with explicit ordering in the forward pass, the communication in the forward pass is implemented such that as soon as computation produces all the outputs that are consumed by a communication collective, the communication call will launch. This maximizes the overlapping window in the forward pass by starting the communication as early as possible. In the backward pass, however, this leads to the opposite effect as the inputs to communication calls will now be launched immediately after the backward communication call and the handles are forced to be synchronized immediately after launching which causes a communication bubble by synchronizing too early in the backward pass. To resolve this, we "hijack" the priority ordering of `torch.autograd` via the Sequence Number PyTorch autograd's internal implementation[1] . In torch autograd, the computational graph will be processed according to a topological sorting algorithm of the dependencies between nodes. However, when multiple nodes are ready for processing at the same time, autograd uses Sequence Numbers to select the first node to process which are typically ordered based on node's chronological creation during forward. As we explained, this leads to the undesired effect where async communication is synced as soon as it launches with our optimized implementation of the forward pass. However with the connectivity of FarSkip-Collective, the dependency drops mean that one can actually process an entire sub-block's backward pass autograd nodes before reaching a dependency barrier on the input to the communication call. Harnessing this, we manually reprioritize the autograd priority queue to prioritize nodes in the sub-block's computation and de-prioritize the processing of the computations leading to the input of the communication call by reassigning them custom Sequence Numbers. With this reassignment those nodes will be launched for backward computation only after the non-dependent sub-block computations took place, allowing for large overlap opportunities before the synchronization points get triggered. Overall with this approach we can flexibly implement different overlapped backward settings and control the execution cleanly without handwriting massive backward autograd functions that will require manually computing every gradient of every sub-block of a layer (attention, MLP and MoE weights and activations).

D FarSkip-Collective Layer Traces

We present excerpt layer traces of explicitly overlapped FarSkip Models during training and inference.

[1] See "forward-backward correlation" discussion of Sequence Number autograd internals in paragraph below the anchor https://docs.pytorch.org/docs/stable/autograd.html#torch.autograd.profiler.emit_nvtx

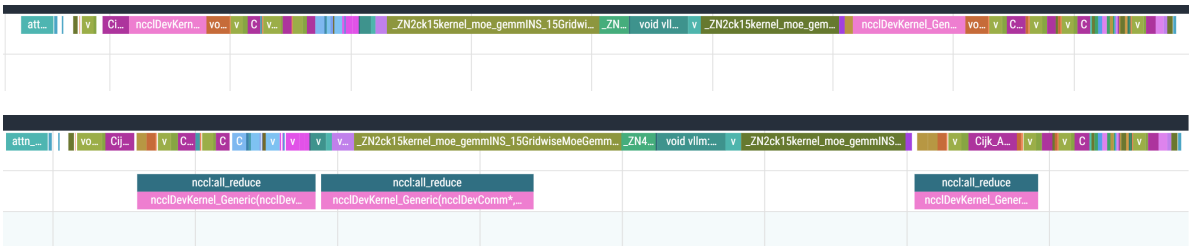

Figure 12: **DeepSeek-V2 vLLM prefill inference layer execution** (Top) regular connectivity (Bottom) FarSkip-Collective. In the bottom figure the all-reduce collectives are overlapped during the attention and MoE sub-blocks by running asynchronously on a 2nd Hardware Queue.

Figure 13: **DeepSeek-V2-Lite pre-training *forward*-pass layer execution** (Top) regular connectivity (Bottom) FarSkip-Collective. In the bottom pane, all-to-all communication is overlapped with computation, the first call corresponds to Dispatch which gets overlapped with the core-attention computation. In the second call, the all-to-all corresponds to Combine and is overlapped with the shared-experts and the next layer's q, k, v computation for attention.

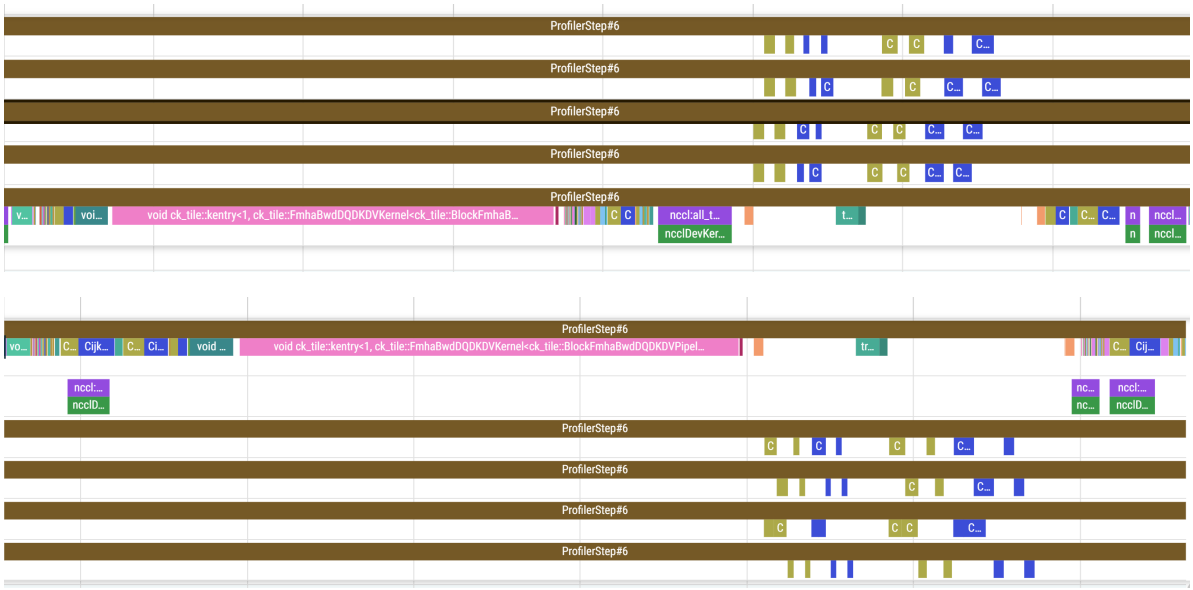

Figure 14: **DeepSeek-V2-Lite pre-training *backward*-pass layer execution** (Top) regular connectivity (Bottom) FarSkip-Collective. The backward-pass operator execution order is "hijacked" from the default `torch.autograd` Sequence Number ordering to re-order operations for overlap. In particular, routed-expert backward computation launches immediately after the finished synchronization point of the Combine all-to-all backwards gradient and the Dispatch gradient launches before the gradient of the first part of attention (q, k, v calculation) to allow for overlapping with it before synchronization.