Attention-Level Speculation

Jack Cai [1 2] **Ammar Vora** [1 2] **Randolph Zhang** [3] **Mark O'Connor** [2] **Mark C. Jeffrey** [1]

Abstract

As Large Language Models (LLMs) grow in size and context length, efficient inference strategies are essential to maintain low-latency token generation. Unfortunately, conventional tensor and data parallelism face diminishing returns when scaling across multiple devices. We propose a novel form—attention-level speculative parallelism (ALSpec)—that predicts self-attention outputs to execute subsequent operations early on separate devices. Our approach overlaps attention and non-attention computations, reducing the attention latency overhead at 128K context length by up to $5\times$ and improving end-to-end decode latency by up to $1.65\times$, all without sacrificing quality. We establish the fundamental pillars for speculative execution and provide an execution paradigm that simplifies implementation. We show that existing attention-approximation methods perform well on simple information retrieval tasks, but they fail in advanced reasoning and math. Combined with speculative execution, we can approximate up to 90% of self-attention without harming model correctness. Demonstrated on Tenstorrent's NPU devices,[1] we scale up LLM inference beyond current techniques, paving the way for faster inference in transformer models.

1. Introduction

State-of-the-art Large Language Models (LLMs) for token generation often rely on decoder-only transformer architectures (Touvron et al., 2023; Dubey et al., 2024; Jiang et al., 2024; Yang et al., 2024; Almazrouei et al., 2023). These models stack repetitive layers of self-attention and feed-forward operations, allowing efficient scaling across layers (Tay et al., 2022; Kim et al., 2023).

As model sizes and context lengths increase (Chitty-Venkata et al., 2024; Shoeybi et al., 2019; Huang et al., 2019), a single accelerator can no longer meet the memory and compute demands. *Tensor parallelism* (splitting the model weights across devices) and *data parallelism* (replicating the model weights on each device) have been widely used for large-scale LLM inference. Despite their adoption, these techniques exhibit the following problems during the causal decode phase, hurting token-generation latency:

1. Tensor parallelism suffers from communication overhead. Latency for operations like *all-gather* grows with more devices, yielding diminishing returns with device scaling.
2. Data parallelism improves throughput with large batch sizes but does not improve latency.
3. As the key-value (KV) caches grow with context length, self-attention becomes a bottleneck and increases overall latency (see Section 5.1 and Appendix D for details).

We observe in Figure 1(left) that during token generation at small context length, the majority of latency comes from the non-self-attention operations, particularly the feed-forward networks. The self-attention latency grows linearly with context length. In contrast, the non-self-attention latency stays constant. This motivates our question: *Can we overlap attention with other operations to hide the increasing cost without compromising model correctness?*

We propose *attention-level speculation (ALSpec)*, replacing some full self-attentions with cheaper approximations (e.g., windowed attention with attention sink (Xiao et al., 2024)) and verifying the approximation against the true attention in parallel. When the approximation is accurate, ALSpec commits and skips expensive calculations. This form of speculation, when successful, allows non-self-attention operations to overlap in time with the self-attention operations. We find that *up to 90%* of the self-attention operations can be safely speculated without harming output quality. In contrast to decode-level speculation (Leviathan et al., 2023), ALSpec can be applied to emerging model architectures with recurrent hidden states, such as layer-looping (Eyuboglu et al., 2024) and continuous chain-of-thoughts (Hao et al., 2024). When computing resources are abundant, ALSpec could be coupled with tensor parallelism and decode-level speculation to deliver even lower latency. Figure 1 shows that ALSpec delivers up to $5\times$ reduction in

[1]University of Toronto [2]Tenstorrent [3]University of Waterloo. Correspondence to: Mark C. Jeffrey <mcj@ece.utoronto.ca>.

Proceedings of the 42^{nd} International Conference on Machine Learning, Vancouver, Canada. PMLR 267, 2025. Copyright 2025 by the author(s).

[1]Our code is at github.com/mcj-group/alspec

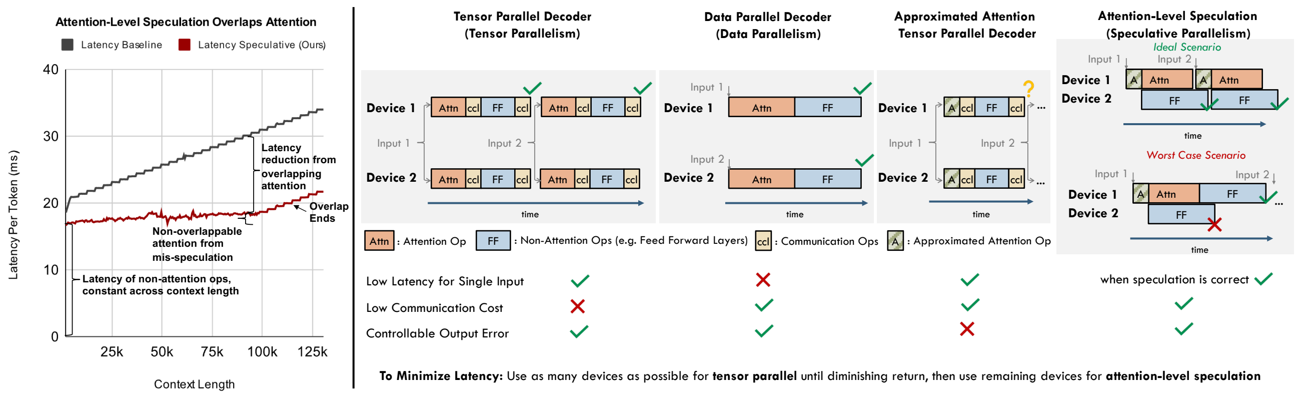

Attention-Level Speculation Overlaps Attention
Latency Baseline
Latency Speculative (Ours)
Latency Per Token (ms)
Context Length
Latency reduction from overlapping attention
Overlap Ends
Non-overlappable attention from mis-speculation
Latency of non-attention ops, constant across context length
Tensor Parallel Decoder (Tensor Parallelism)
Data Parallel Decoder (Data Parallelism)
Approximated Attention Tensor Parallel Decoder
Attention-Level Speculation (Speculative Parallelism)
Ideal Scenario
Worst Case Scenario
Device 1
Device 2
Input 1
Input 2
time
Attn : Attention Op
FF : Non-Attention Ops (e.g. Feed Forward Layers)
ccl : Communication Ops
A : Approximated Attention Op
Low Latency for Single Input
Low Communication Cost
Controllable Output Error
when speculation is correct
To Minimize Latency: Use as many devices as possible for tensor parallel until diminishing return, then use remaining devices for attention-level speculation

Figure 1. (*Left*) Our key result: Llama 3.1 8B decode-stage latency across context lengths on 8 devices, comparing the 8-device full tensor-parallel baseline with our 4-device tensor parallel plus 4-device speculative parallel at 87.5% speculation hit rate (see Section 5.1). (*Right*) Comparing tensor parallel (TP), data parallel (DP), attention approximation, and ALSpec when executing 2 independent inputs on 2 devices (inputs are tensor representations of previous output tokens). TP reduces per-input latency but suffers communication overheads. DP gives highest throughput but also high per-input latency. Approximation can suffer from output errors, despite latency improvements. ALSpec keeps correctness and reduces latency relative to TP when speculations are correct, with latency no worse than DP.

attention latency overhead for long-context decode at 128K. It also contrasts ALSpec with tensor-parallel, data-parallel, and approximated attention, which we detail in Section 2.

We further contribute a *speculative flash decode* kernel that computes approximate and exact attention in one pass, adding little to no overhead for speculation. Our new execution paradigm, *static graph, dynamic concurrency (SGDC) with priority gating*, maintains an op-by-op host view, while running approximate and exact attention paths concurrently. This mechanism delivers up to 1.65× speedup with 2× devices at long context lengths, even in cases where tensor parallelism fails to scale.

Our insight builds on prior observations that sparse attentions are effective (Kitaev et al., 2020; Beltagy et al., 2020; Xiao et al., 2024), but static approximate methods often fail on advanced reasoning or math tasks. By only accepting certain approximations *at run time*, we preserve accuracy across a broad set of benchmarks including reasoning, math, and information retrieval (Figure 2).

Inspired by instruction-level value prediction in CPUs, ALSpec exploits attention sparsity in a dynamic and robust manner. Specifically, we demonstrate that a subset of challenges in transformer—safely exploiting the sparsity of self-attention—can be effectively addressed through dynamic execution with speculation. In addition to accelerating transformer-based models, it points toward new opportunities for applying generic speculation paradigms to deep learning inference, unlocking new avenues for scaling.

2. Motivation

Self-attention tends to dominate the latency of LLM inference at larger context lengths, as every new token attends to an increasingly large key-value (KV) cache. Various parallelization strategies use more computing resources for lower latency or higher throughput, including *tensor parallelism* (Shoeybi et al., 2019), *data parallelism* (Huang et al., 2019), and *pipeline parallelism* (DeepSeek-AI et al., 2024). Figure 1(*right*) qualitatively compares these methods when running two inputs on two devices. Tensor parallelism splits the model weights across multiple devices to accelerate attention and feed-forward layers. Although it reduces latency for a single input, it suffers from communication overhead (e.g., *all-gather* of partial results), which grows as the number of devices increases. This yields diminishing returns when scaling past a handful of devices. Data parallelism replicates the model weights on all devices and processes multiple inputs (or tokens) simultaneously for higher throughput. However, with interactive LLM inference, the latency per input is paramount; data parallelism provides minimal improvement because each input runs on one device. Pipeline parallelism (not shown) partitions the model layers so that each device is responsible for one portion of the forward pass. Once the pipeline fills, it can achieve the same overall throughput as data parallelism. However, each input still traverses all pipeline stages, yielding a similar latency profile to data parallelism. Hence, pipeline parallelism, like data parallelism, does not fundamentally reduce inter-token decode latency.

Orthogonal to scaling up device count, *sub-quadratic algorithms* (Gu & Dao, 2024; Katharopoulos et al., 2020; Poli et al., 2023) and sparsity-exploiting *attention approximation* methods cut attention computation and memory demands. Reformer (Kitaev et al., 2020) replaces the standard dot-product attention with locality-sensitive hashing (LSH) to reduce the per-token computational complexity from linear to logarithmic in context length. SCFA (Pagliardini et al.,

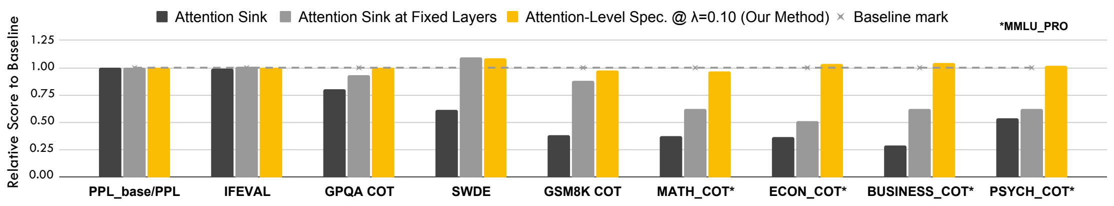
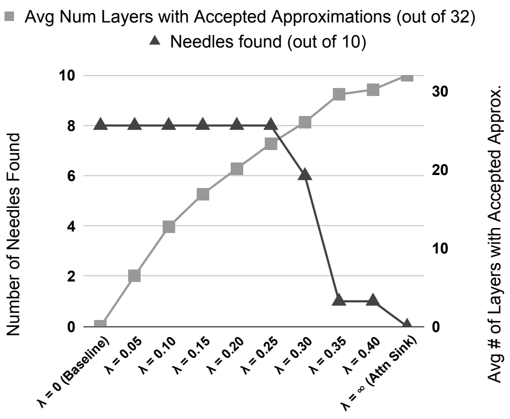


Figure 2. Correctness evaluation of attention approximation methods on various benchmarks using Llama 3.1 8B. Baseline is the model with unmodified attention. Attention sink fails for tasks requiring math and reasoning on specific subject area, such as MMLU_PRO with chain-of-thoughts (COT). Statically allowing attention sink only at specific layers reduces degradation, but still causes a noticeable gap. Dynamically using attention sink with ALSpec gives consistent correctness to the baseline for all tasks at acceptance threshold $\lambda = 0.10$.

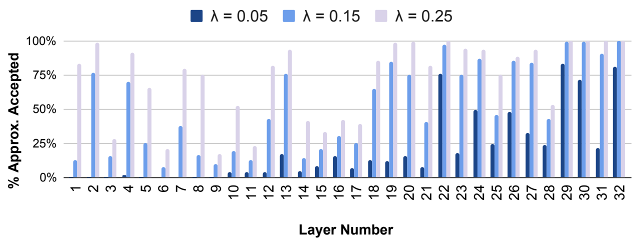
2023) further reduces the computational cost of Reformer. Longformer (Beltagy et al., 2020) relies on sliding-window (local) attention plus a small number of global tokens. These approaches typically exhibit *static* sparsity in the sense that the approximation pattern is fixed prior to execution. Although effective in some cases (e.g., perplexity on large-context text) or with retraining, the static nature of these methods means the model execution graph is fixed prior to execution, inevitably causing quality loss in scenarios where the model needs full attention for correct results (Figure 1).

Among the approximation techniques, attention sink (Xiao et al., 2024) is simple yet surprisingly powerful. It focuses the self-attention on two sets of tokens: the first few tokens (e.g., 4) and a rolling window of S recent tokens from the KV cache. This makes the effective context length of attention drastically smaller than the true context length. Attention sink achieves good performance on certain tasks (e.g., perplexity and information retrieval), even with a small S. However, any global information beyond the S-sized window or the first few tokens is inaccessible, causing attention sink to fail on tasks that require distant context or advanced reasoning. Figure 2 illustrates how attention sink on Llama 3.1 8B struggles on tasks like advanced math or reasoning (e.g., MMLU_PRO with chain-of-thought), where it yields a noticeable quality drop compared to the baseline (see Section 5.1 for methodology).

The success or failure of approximation appears to be *layer-specific*: some layers can tolerate an approximate self-attention output, while others require exact attention. Prior work shows that certain layers can be pruned or skipped without significantly impacting model quality (Elbayad et al., 2020). Consequently, one might wonder if we can *statically* choose a subset of layers to apply attention sink. To concretize this hypothesis, we design a synthetic *Needles in a Haystack* test on Llama 3.1 8B. The model inputs a 16K-token context containing 10 randomly inserted “secret keys.” It must retrieve all keys from the entire text. Figure 3a shows that a model with pure attention sink ($S = 128$) recovers *none* of the secret keys. In contrast, the baseline model finds 8 of 10. Between these configurations, we create models that



(a) Using attention sink at select layers does not harm correctness



(b) Average percentage of approximation accepted per layer

Figure 3. Finding needles in a haystack with Llama 3.1 8B.

selectively use attention sink with criteria that ensure only good approximations are used: a layer accepts attention sink if, for the given token, the L_2 distance between the attention sink output $\tilde{A}$ and full attention output A is within $\lambda \cdot L_2\_\text{norm}(A)$. A full attention sink model corresponds to $\lambda = \infty$ and the baseline model corresponds to $\lambda = 0$. Figure 3a shows that for $\lambda < 0.25$, up to $2/3$ of the layers can be approximated with attention sink without sacrificing the score. Figure 3b further shows that approximations are layer-specific: certain layers, especially the final layers, are more likely to accept approximation without harm, corroborating prior observations that deeper layers can be pruned with limited effect (Elhoushi et al., 2024).

Figure 2 shows that purely relying on a fixed layer-based approximation pattern—although better than applying attention sink to *all* layers—exhibits noticeable quality degradation when extended to a variety of tasks. For the middle bars, we create fixed layer-based approximated models, where attention sink is always used at a layer iff the likelihood to approximate that layer is greater than $p = 0.8$ in Figure 3b. The static approximation models do not balance performance gains with accuracy. Some inherently harder inputs or tasks require additional global context; even for the same task, the best approximation policy may vary from token to token or layer to layer. We therefore conclude that *no single static approximation* suffices; the key is to make decisions *dynamically* at run time.

We need a simple yet powerful way to realize dynamic execution (Barad et al., 2024). *Speculative parallelism* involves a guess about future outcomes, a verification of that guess, and error recovery when the guess is wrong. By speculating, systems can proceed optimistically and exploit opportunities for parallelism, even in the face of run-time uncertainty. Speculation is a long-standing technique in computer architecture, spanning branch, dependence, and value prediction (Rau & Fisher, 1993; Lipasti & Shen, 1996; Sazeides & Smith, 1997) to task-level speculation (Sohi et al., 1995; Herlihy & Moss, 1993; Jeffrey et al., 2015). Speculation enables concurrent or parallel execution of instructions or blocks of code, but discards results when the guesses fail. In LLMs, speculation thus far has appeared only at the decode (or token) level via speculative decoding (Leviathan et al., 2023; Hooper et al., 2023; Spector & Re, 2023; Li et al., 2024; Miao et al., 2024) which drafts several tokens in parallel. This includes decode-level speculation that employs attention sink (Sun et al., 2024).

3. Attention-Level Speculation Overview

Attention-level speculation (ALSpec) speculates on the self-attention output within each transformer layer. Specifically, ALSpec approximates the self-attention with a technique like attention sink and subsequently computes the exact self-attention. Once the exact result arrives, ALSpec *verifies* how close the approximate output is. If sufficiently close, ALSpec accepts it; if not, it reverts to the exact output. The "ideal scenario" of Figure 1 shows that speculation can overlap attention with feed-forward layers, greatly reducing latency when the prediction is correct. The "worst case" matches data parallelism latency; if the speculation fails, the computation must fall back to the exact path. This approach harnesses parallel devices (or threads) to overlap computation, hiding some of the attention latency while preventing quality degradation when approximation fails. We show that a simple approximation—namely attention sink—serves effectively as the "predictor." For each layer, we apply it to self-attention, generate a speculated attention output $\tilde{A}_i$ in layer i, and concurrently compute the ground-truth output A_i. Since attention sink can fail for certain tasks, we design a *verification* step to ensure correctness. In practice, we find that over half of the layers can skip full self-attention by verifying that $\tilde{A}_i \approx A_i$. Meanwhile, ALSpec maintains baseline accuracy across tasks (Figure 2).

3.1. Low-Cost Predictor

We use a coarser version of attention sink that inputs the first and last S tokens from the KV cache. We choose S to be small relative to the context length (e.g., $S \in \{128, 256, 512\}$ for a $128K$ context length). In Section 4.1, we choose S as the chunk size of speculative flash decode.

3.2. Low-Cost Verification

Relaxed Verification via Lipschitz Continuity: Exact equality between $\tilde{A}_i$ and A_i is not mandatory to preserve correctness in LLMs; we only require that their difference remains bounded so that final outputs have small deviations. Lipschitz continuity (Scaman & Virmaux, 2018) provides a theoretical tool to relate small perturbations in one layer's output to eventual changes in final logits. Although self-attention alone is not Lipschitz continuous (Kim et al., 2021; Castin et al., 2024; Geshkovski et al., 2023), feed-forward networks, layer norms, and residual connections possess (or can be bounded by) Lipschitz constants. In addition, Geshkovski et al. show that self-attention's Lipchitz constant is bounded by $CR^2e^{CR^2}$, where R is the input magnitude bound. Combining these, we derive an upper bound on the final output's deviation in terms of per-layer differences $\|\tilde{A}_i - A_i\|$. Concretely, if each approximate attention output is within δ_i in ℓ_p-norm of the ground-truth A_i, the overall output deviation, ϵ, is bounded by

$$\epsilon \leq \Sigma_{i=1}^{N}(1+\alpha)^{N-i+1}(1+f(R)\beta)^{N-i}\delta_i \tag{1}$$

where α, β, and $f(R)$ are bounded constants for the Lipschitz behavior of residuals, feed-forward networks, and self-attention inputs, respectively. A detailed proof and discussion appear in Appendix A.

Verification Algorithm: Motivated by the above bounds we adopt a simple threshold-based metric to accept or reject $\tilde{A}_i$. Concretely, we measure $d(\tilde{A}_i, A_i)$ in the ℓ_2 norm and accept if: $\|\tilde{A}_i - A_i\|_2 < \lambda \cdot \|A_i\|_2$, where λ is a small scalar hyperparameter. Algorithm 1 sketches the speculative execution procedure. For each layer, the algorithm *(i)* computes $\tilde{A}_i$ using attention sink, *(ii)* launches a new thread to continue the "post-attention" operations, *(iii)* computes A_i in the main thread, then *(iv)* verifies if $\tilde{A}_i$ is sufficiently close. If yes, it commits $\tilde{A}_i$; otherwise, it discards the new thread and re-runs the next operations on the main thread's A_i. A simple thresholding strategy effectively ensures the algorithm does not propagate large attention errors. In addition, we show a high probability error bound on ϵ in the

Algorithm 1 Speculative Execution with Attention Sink

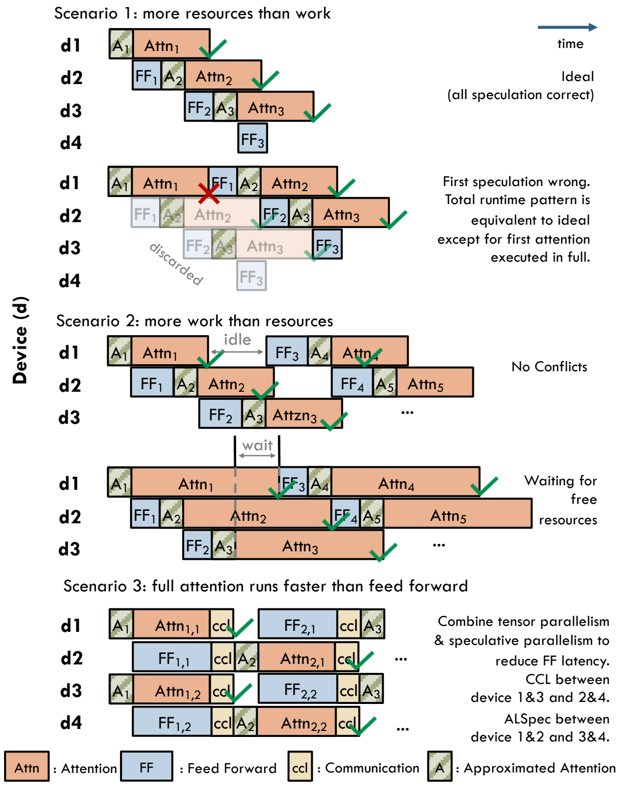
```
Input: activation $x$, N layers, threshold $\lambda$
Start execution as the $main\_thread$.
for $i = 1$ to $N$ do
    $x$ = ops_before_self_attn($x$) // e.g. LayerNorm
    $x_{spec}$ = attention_sink($x$)
    $new\_thread$ = Thread(ops_after_self_attn, $x_{spec}$)
    $x$ = regular_attention($x$)
    if $L_2$_distance($x, x_{spec}$) $< \lambda \cdot L_2$_norm($x$) then
        $main\_thread = new\_thread$
    else
        Kill $new\_thread$
        $x$ =ops_after_self_attn($x$) // e.g. Feed Forward
    end if
end for
```

order of $\sqrt{N}$ (Appendix A.3). This is consistent with our experimental results on deeper models (Section 5.2), achieving similar results as shallower models across benchmarks.

3.3. Example Speculative Executions

Figure 4 shows three high-level scenarios that can arise when applying Algorithm 1 on multiple devices in parallel. *Scenario 1: More resources than work.* With sufficiently many devices, every layer's approximate and exact attention can run in parallel without contention. In the *ideal* case, all speculations are correct, so each device can immediately continue with post-attention operations (e.g., feed-forward (*FF*) layers), yielding maximum speedup by overlapping all FF within attention. In the *mis-speculation* case, when the first approximation fails, the system reverts to full attention for that step. Subsequent layers resume the normal speculative pattern. The total latency impact of one mis-speculation remains bounded by the time needed to execute a single attention pass in full. *Scenario 2: More work than resources.* In practice, we often have fewer devices than are needed to run every speculation fully in parallel. A device may need to *wait* until a prior layer finishes using the hardware before starting its own speculation. *Scenario 3: Full attention runs faster than feed-forward.* In this scenario, speculation only uses 2 devices as the prior device will be free when FF finishes. In such cases, we combine tensor parallelism with speculative parallelism: Figure 4 illustrates a 4-device system using 2 devices for tensor parallelism and 2 devices for speculative parallelism with 1:1 mapping.

In most real-world LLM usage, FF dominates run time while attention latency grows with context length (Dao, 2024). For context lengths up to 128K tokens, FF remains the primary bottleneck, leading to *Scenario 3*, whereas *Scenario 1* or *Scenario 2* only become relevant at extremely large context lengths. Moreover, *Scenario 1 & 2* require fully pipelining or even speculating on still-speculative data (Jeffrey et al.,



Scenario 1: more resources than work
time
Ideal (all speculation correct)
First speculation wrong. Total runtime pattern is equivalent to ideal except for first attention executed in full.
discarded
Device (d)
d1
d2
d3
d4
Scenario 2: more work than resources
idle
No Conflicts
wait
Waiting for free resources
Scenario 3: full attention runs faster than feed forward
Combine tensor parallelism & speculative parallelism to reduce FF latency.
CCL between device 1&3 and 2&4.
ALSpec between device 1&2 and 3&4.
Attn : Attention
FF : Feed Forward
ccl : Communication
A : Approximated Attention

Figure 4. Algorithm 1 run-time patterns. (*Scenario 1*) *More resources than work:* Each approximate and exact path can run concurrently on separate devices. If all speculations are correct, total latency is minimized; a mis-speculation simply triggers the exact attention for one layer. (*Scenario 2*) *More work than resources:* Some computations must wait for free devices, adding stalls. Otherwise, the overlap pattern is similar to Scenario 1. (*Scenario 3*) *Feed-forward dominates:* When FF is slower than attention, we can pipeline or combine tensor parallelism with speculative parallelism (mapping devices 1:1) to further reduce FF latency.

2018), complicating a static execution graph. In contrast, *Scenario 3* naturally fits an *op-by-op* framework: one device is used for an approximate path and another for an exact path. No further pipelining is required. Hence, in the remainder of this paper, we focus on implementing *Scenario 3* and extending it with tensor parallelism in real systems.

3.4. Overheads

On the communication side, combining ALSpec and TP (*Scenario 3*) for n devices reduces latency vs. full TP on n devices. In each TP operation, the bandwidth-optimal all-reduce algorithm (Thakur et al., 2005) has each device send and receive $\frac{n-1}{n}$ of the tensor per all-reduce. While the total bandwidth demands per device remain nearly constant as TP device count increases, the latency of all-reduce operations during decode are often exposed on the critical path due to small activation sizes (Narayanan et al., 2021). This latency is highly dependent on the underlying network topology. For example, in a ring topology (Sergeev & Balso, 2018), all-reduce requires $n-1$ hops of exposed latency. In contrast, ALSpec+TP on n devices performs all-reduce across $\frac{n}{2}$ devices, with $\frac{n}{2}-1$ hops of exposed latency and adds 1 hop of exposed latency for the approximating devices to send speculated activations to the speculative FF path.

On the computation side, the ALSpec verification step introduces two L_2 norm calculations of the ground truth and speculated activations. This L_2 norm cost is similar to LayerNorm. For decode with small activations, the L_2 norm computation is negligible compared to FF and attention.

3.5. Summary

Speculative parallelism offers a simple, *dynamic* extension to static approximation. Rather than retraining or modifying model architecture, we only overlay a prediction–verification mechanism. Speculative parallelism extends naturally to other sparse attention approaches, creating a versatile framework for dynamic execution. This dynamic viewpoint transforms approximate-attention methods like attention sink from "always approximate" to "approximate only when safe," mitigating their inherent latency–quality tradeoff. By combining run-time verification with parallel computation, we overlap approximate and full-attention paths to reduce the critical path latency.

4. Implementation

Implementing *Scenario 3* poses significant challenges. Traditional frameworks assume an *op-by-op* execution pattern over a static dataflow graph that maps onto well-known scaling paradigms like tensor and data parallelism. By contrast, attention-level speculation requires dynamic conditional execution over the graph. An approximate path runs concurrently with computing the exact attention, and only upon verification do we commit to one.

To make the implementation feasible, we introduce two key optimizations. First, we design a custom *Speculative Flash Decode kernel* that fully hides the cost of the approximate attention path to incur minimal penalty if a speculation is rejected. Second, we introduce *static graph, dynamic concurrency (SGDC) with priority gating*: a mechanism to preserve op-by-op host code and a static computation graph, while dynamically forking and merging the speculative path under the hood. Verification then determines which result to commit, controlled by a lightweight priority gating mechanism.

4.1. Speculative Flash Decode

In Algorithm 1, the attention sink runs before the regular attention. In the worst case, where all speculations are rejected, the latency would be worse than pipelining. To mitigate this, we modify the read pattern of the flash decode kernel and implement an all-fused kernel that speculates for *free* while computing full attention and verification at once. Appendix B shows the full algorithm.

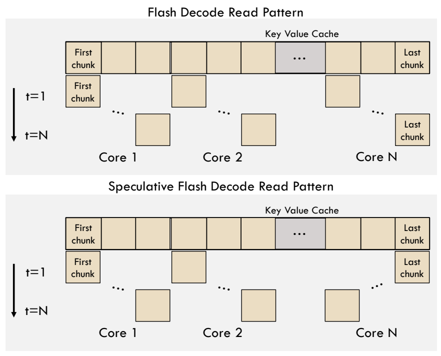

Flash Decode Read Pattern
Key Value Cache
First chunk
Last chunk
t=1
t=N
Core 1
Core 2
Core N
Speculative Flash Decode Read Pattern

Figure 5. Flash decode and speculative flash decode KV cache read patterns. Speculative flash decode reads the first and last chunks first and computes a partial result as speculation.

In short, the intuition of speculative flash decode is shown in Figure 5. The original flash decode algorithm (Dao, 2024; Dao et al., 2023) processes self-attention by chunks of KV cache from the start to the decoding position, where groups of chunks are distributed across the available cores. The results are then aggregated using the statistics. *We exploit the order invariance in which the chunks are processed.* In speculative flash decode, we set the chunk size to be S, assign the first and last chunk to be read first, and send the aggregated partial result immediately to the next device to execute the speculated path. The partial result is equivalently the attention sink on the first and last S tokens, and can be used later towards calculating the full self-attention result. We then combine the verification step into the same single kernel, using it as a drop-in replacement for PyTorch's `scaled_dot_product_attention`.

4.2. SGDC with Priority Gating

Algorithm 2 implements *Scenario 3* of Algorithm 1 on two groups of devices by using the speculative flash decode kernel and a small change to the runtime, namely SGDC with Priority Gating. It uses the following methods:

- **skip_compute($\mathbf{p}[k] == 0$):** Skips a subsequent operation's execution when $p[k] == 0$, where k indicates the device index of itself.
- **all_gather($\mathbf{p}$):** Gathers $\mathbf{p}$ across d_1, d_2 (d with the highest priority holds the correct x).
- **speculative_flash_decode(x, $\mathbf{p}$):** Executes a speculative decoding operation and determines the sender and receiver based on $\mathbf{p}$. Sender sets $p_k = 2$ if verification fails and otherwise $p_k = 0$. Receiver sets $p_k = 1$.

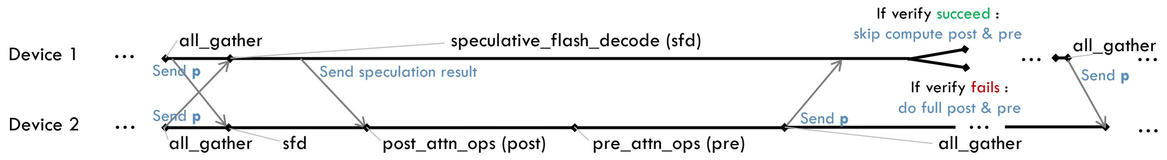


Figure 6. Timing diagram of speculative execution on two devices, based on Algorithm 2. We show a segment of execution timeline from the all-gather op in the first layer to the all-gather op in the second layer. When speculation succeeds, Device 1 skips the computation in post- and pre-attention ops. When speculation fails, it executes them fully.

Algorithm 2 SGDC with Priority Gating

```
Input: replicated activation $x$, N layers, threshold $\lambda$,
replicated priority vector $p$, device $d_1$, $d_2$, device id $k$.
Initialize $\mathbf{p} = [p_1 = 2, p_2 = 0]$, indicating $d_1$ holds the
correct $x$ with higher priority.
for $i = 1$ to $N$ do
    with skip_compute($\mathbf{p}[k] == 0$):
        $x$ = ops_before_self_attn($x$)
    $\mathbf{p}$ = all_gather($\mathbf{p}$)
    $x$ = speculative_flash_decode($x$, $\mathbf{p}$, $\lambda$)
    with skip_compute($\mathbf{p}[k] == 0$):
        $x$ =ops_after_self_attn($x$)
end for
return $x$ from the correct device based on $\mathbf{p}$
```

Figure 6 illustrates Algorithm 2. We maintain static host code where model and input activations are replicated across two devices, effectively data parallelism. Each device maintains a priority vector to track which of them holds the correct activations. Initially, both devices hold the correct activations; once speculations and computation skipping happen, only one device holds it. We perform an all-gather of the priority vector before the speculation point to determine which device has highest priority. At the speculation point, the sender device with highest priority (indicating correct activations) will execute the compute side of the speculative flash decode op, and the receiver device waits for the speculative result, then continues subsequent ops. When the full self-attention finishes, the sender will either skip the computation of the subsequent ops if the verification succeeds, or proceed executing the ops normally if the verification fails. We maintain the same ops across devices and the host dispatches all the ops asynchronously. This method can be extended to tensor + speculative parallelism: when the first group comprises > 1 tensor-parallel devices, the second group has the same number of devices with 1-1 mapping.[2]

[2]Appendix E validates SGDC with a micro-benchmark. In our real implementation, we also sync the residual stream at attention to ensure the KV cache and residual are calculated correctly. We overlap it with compute so that it does not cause extra overhead.

5. Experiments

We evaluate the implementation of Algorithm 2 with the speculative flash decode kernel to answer the following questions: *(i)* Does dynamic execution really improve performance on real device implementation? *(ii)* What rate of speculation hit rate do we see across benchmarks? *(iii)* Why is dynamic sometimes superior to static approximation?

5.1. Methodology

Model & Hardware: We answer the previous questions with a case study of parallelizing Llama 3.1 8B model onto 8 Tenstorrent N150 chips. The model architecture and weights are taken from the official Llama codebase without modification and implemented on Tenstorrent N150 chips, using the Tenstorrent Metalium (TT-Metalium) kernel library.[3] The model is executed with default mixed precision configuration,[4] with BF16 activations, BF8 KV cache, and BF{16,8,4} model weights. For correctness evaluations, we run the model in BF16 precision using an NVIDIA A100 or H100 GPUs; we simulate speculation by running both full and approximated attention ($S = 128$) and choosing the approximated attention output if verification succeeds.

Benchmarks: We evaluate the correctness and speculation hit rate measurement at various choices of λ. We group the evaluations into categories of {Question Answering (QA), Information Retrieval (IR), Reasoning, Long Context, and Math}. We use the LM Evaluation Harness (Gao et al., 2024), a framework for few-shot language model evaluation, with the default prompts and few-shot templates. As we focus on how ALSpec outputs diverge from the baseline, we do not explicitly optimize for baseline model scores.

[3]We conducted our experiments on Tenstorrent N150 devices using the pre-release version of the TT-Metalium v0.55.0-rc13 software stack. Tenstorrent has active ongoing software optimizations, therefore the performance results presented here are preliminary and do not reflect the latest optimized capabilities of Tenstorrent devices. Rather, they serve to illustrate the relative performance advantage provided by ALSpec. See details about TT-Metalium (Tenstorrent, 2024b), speculative flash decode, and SGDC with priority gating implementation in Appendix C, D & E.

[4]See Appendix F for TT-Metalium Llama implementation and performance measurement with tensor parallelism and ALSpec.

Table 1. Evaluation of attention-level speculation at various speculation verification threshold λs for Llama 3.1 8B model across benchmarks, grouped by categories. Each entry contains (evaluated scores (0-1)) / (speculation hit rate (0-100%)). Higher λ corresponds to more relaxed verification, allowing higher speculation hit rates, while lower λ keeps outputs closer to the baseline.

TASK	QA		IR		MATHEMATIC		
	IFEVAL	GPQA COT	SWDE	FDA	GSM8K COT	MATH*	MGSM COT SW
BASELINE	0.798	0.237	0.359	0.210	0.823	0.326	0.580
$\lambda = 0.05$	0.799 / 69.5%	0.257 / 38.0%	0.371 / 57.8%	0.205 / 37.5%	0.806 / 49.4%	0.332 / 21.6%	0.592 / 53.6%
$\lambda = 0.10$	0.812 / 89.8%	0.237 / 65.7%	0.389 / 83.2%	0.205 / 69.3%	0.799 / 78.9%	0.315 / 49.6%	0.584 / 77.6%
$\lambda = 0.15$	0.807 / 94.8%	0.214 / 81.1%	0.391 / 92.9%	0.190 / 85.7%	0.799 / 90.2%	0.331 / 69.3%	0.592 / 86.0%
$\lambda = 0.20$	0.805 / 98.0%	0.225 / 90.4%	0.384 / 97.0%	0.172 / 94.4%	0.795 / 94.6%	0.323 / 79.4%	0.564 / 95.4%
$\lambda = 0.25$	0.794 / 99.3%	0.208 / 95.4%	0.359 / 98.8%	0.158 / 98.2%	0.719 / 96.2%	0.266 / 86.3%	0.516 / 96.6%

TASK	LONG CONTEXT		REASONING		
	HOTPOTQA	REPOBENCH-P	ECON*	BUSINESS*	PSYCH*
BASELINE	0.920	0.756	0.328	0.470	0.289
$\lambda = 0.05$	0.920 / 18.5%	0.736 / 33.1%	0.341 / 18.9%	0.477 / 22.8%	0.232 / 23.3%
$\lambda = 0.10$	0.915 / 42.8%	0.768 / 59.6%	0.339 / 45.8%	0.490 / 50.0%	0.296 / 52.3%
$\lambda = 0.15$	0.920 / 56.8%	0.812 / 75.5%	0.321 / 63.0%	0.473 / 68.5%	0.294 / 67.5%
$\lambda = 0.20$	0.910 / 71.6%	0.756 / 85.0%	0.316 / 77.5%	0.437 / 80.7%	0.292 / 80.9%
$\lambda = 0.25$	0.705 / 80.1%	0.742 / 91.8%	0.273 / 87.7%	0.359 / 88.3%	0.281 / 91.3%

* MMLU_PRO BENCHMARK WITH CHAIN OF THOUGHTS DECODING.

Measurement: We use the TT-Metalium profiler to measure the device kernel time of every operation during decode at specific context lengths and speculation hit rates. This performance measurement shows an ideal scalability pattern because it does not account for host and op dispatch latency, which are subject to the host configuration. We report latency per token, which is estimated by taking the measured kernel run time per layer and multiplied by the total number of layers (32). The metric token/s/user is calculated by 1/(latency per token).

5.2. Speculation Hit Rates and Performance Results

Table 1 shows the output quality and speculation hit rate for $\lambda \in \{0.05, 0.10, 0.15, 0.20, 0.25\}$ compared to baseline. ALSpec with $\lambda \in \{0.05, 0.10\}$ achieves on par or better correctness on all evaluated tasks, with speculation hit rates ranging from 18% (HotpotQA) to 90% (IFEval) depending on the task, and most speculation hit rates exceeding 50% for $\lambda = 0.10$. Relaxing λ to 0.15 allows the model to speculate up to 95% of attentions successfully while doing slightly worse on GPQA and FDA. Adjusting λ beyond 0.15 allows flexibly trading off model accuracy for computation latency, which could be helpful depending on the use case. Table 2 shows a subset of benchmarks with Llama 3 70B (80 decoder layers), confirming that a larger and deeper model achieves similar speculation hit rate with limited correctness degradation at $\lambda \in \{0.05, 0.10\}$.

Leveraging the previous experiments, we estimate performance for Llama 3 8B on N150 chips based on commonly observed ranges of the speculation hit rate: {50, 62.5, 75, 87.5}%. Our results show that ALSpec provides additional scaling when tensor parallelism scaling diminishes. Figure 7 shows that full tensor parallelism provides almost no improvements when scaling from 4 to 8 devices at long context length, whereas speculative + tensor parallelism continues to scale. In the best case (96k context length with 87.5% speculation hit rate), ALSpec provides $1.65\times$ improvement in per-user throughput compared to full tensor parallelism.

Table 2. Evaluation of speculative execution at various verification thresholds λ for Llama 3.1 70B model across selected benchmarks.

TASK	GPQA COT	GSM8K COT	HOTPOTQA
BASELINE	0.518	0.958	0.940
$\lambda = 0.05$	0.529 / 55.1%	0.950 / 64.5%	0.940 / 32.3%
$\lambda = 0.10$	0.507 / 80.0%	0.951 / 86.6%	0.945 / 59.0%
$\lambda = 0.15$	0.458 / 89.9%	0.946 / 93.9%	0.935 / 74.6%
$\lambda = 0.20$	0.446 / 95.1%	0.936 / 96.8%	0.935 / 83.3%
$\lambda = 0.25$	0.379 / 97.6%	0.897 / 98.3%	0.935 / 89.4%

5.3. Reproducibility and Estimation on GPUs

We corroborate the diminishing returns of tensor parallelism previously seen on GPUs and estimate ALSpec performance on GPUs. Chitty-Venkata et al. find that the Llama 3 8B model on A100s with DeepSpeed-MII achieves around 75, 90, and 115 tokens/s per-user throughput when scaling across 1, 2, and 4 GPUs. This scaling is similar to what we observe on N150 devices. We measure the Llama 3 8B decoding latency on 4 vs. 8 H100 GPUs doing TP using the SGLang (Zheng et al., 2024) serving framework and the FlashInfer (Ye et al., 2025) attention backend. Although we currently lack an ALSpec CUDA implementation, we estimate the ALSpec performance using SP@65% + TP4. Table 3 shows our estimation that ALSpec cuts latency on Llama 8B by $1.28\times$ vs. $1.09\times$ for full TP at context length 128K. In this case, the attention latency is only 63% of non-attention latency, suggesting greater gains for context lengths beyond 128K. Although ALSpec introduces a new attention kernel and SGDC, it does not fundamentally change the op-by-op and static graph execution style of a GPU runtime. We expect that ALSpec is implementable via CUDA graphs and conditionals (Gaiser et al., 2024).

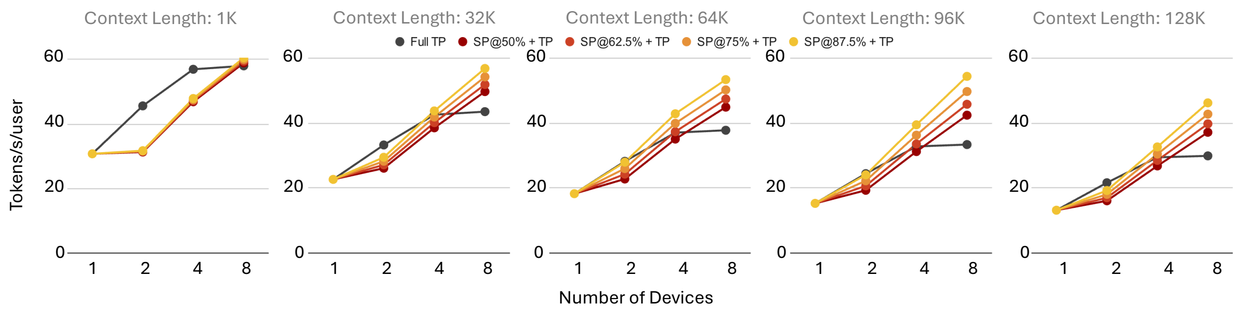
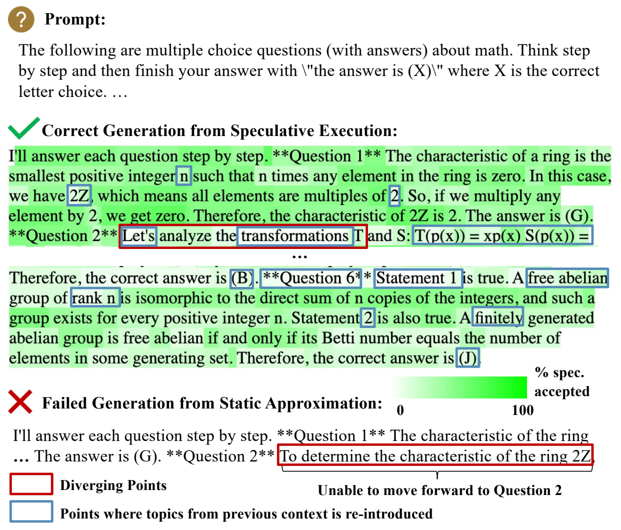


Figure 7. Scalability patterns of Llama 3.1 8B with full tensor parallelism (TP) and speculative parallelism at {50, 62.5, 75, 87.5}% speculation hit rate with tensor parallelism (SP@X% + TP) on Tenstorrent N150 chips. Each plot shows the scalability at a particular decode context length (1k, 32k, 64k, 96k, and 128k). Attention is speculative parallel using the speculative flash decode kernel with chunk size 256 and tensor parallel by head. Feed-forward layers are tensor parallel by inner dimension. Performance estimation is obtained by measuring the device kernel duration of every single kernel, agnostic of the dispatch and host time. At long context lengths, SP+TP (our method) continues to scale with 8 devices while full tensor parallelism scaling diminishes.

Table 3. Latency and throughput scaling for Llama 8B inference on H100s across various context lengths. Latencies are measured per decoder layer, and throughput is in tokens per second (tok/s). Projected ALSpec results assume 65% speculation hit rate.

Context	4xH100 Attn Latency/Layer (us)	4xH100 Non-Attn Latency/Layer (us)	4xH100 TP tok/s	8xH100 TP tok/s	TP Scaling	Projected ALSpec @65% Hit Rate	ALSpec Scaling
1k	13	95	244.6	249.3	1.9%	262.0	7.1%
32k	29	100	214.2	231.3	8.0%	246.0	14.8%
64k	49	100	191.4	209.7	9.6%	237.8	24.2%
96k	56	100	178.2	194.2	9.0%	224.9	26.2%
128k	63	101	169.3	184.8	9.2%	217.5	28.5%


Prompt:

The following are multiple choice questions (with answers) about math. Think step by step and then finish your answer with \"the answer is (X)\" where X is the correct letter choice. …

Correct Generation from Speculative Execution:

I'll answer each question step by step. **Question 1** The characteristic of a ring is the smallest positive integer n such that n times any element in the ring is zero. In this case, we have 2Z, which means all elements are multiples of 2. So, if we multiply any element by 2, we get zero. Therefore, the characteristic of 2Z is 2. The answer is (G). **Question 2** Let's analyze the transformations T and S: T(p(x)) = xp(x) S(p(x)) =

…

Therefore, the correct answer is (B). **Question 6** Statement 1 is true. A free abelian group of rank n is isomorphic to the direct sum of n copies of the integers, and such a group exists for every positive integer n. Statement 2 is also true. A finitely generated abelian group is free abelian if and only if its Betti number equals the number of elements in some generating set. Therefore, the correct answer is (J).

Failed Generation from Static Approximation:

I'll answer each question step by step. **Question 1** The characteristic of the ring … The answer is (G). **Question 2** To determine the characteristic of the ring 2Z,

% spec. accepted
0
100
Unable to move forward to Question 2
Diverging Points
Points where topics from previous context is re-introduced

Figure 8. Example of output divergence in static approximation model. Divergence points corresponds to tokens with low speculation accuracy, where the static approximation model fails to adjust.

5.4. When Does Static Approximation Go Wrong?

In some cases, static approximation is unable to generate key tokens that influence the direction of the conversation, indicating that full attention to the context is required. This usually happens at a change of topic token. Figure 8 illustrates a specific example in MMLU_PRO_MATH where the model is given a math problem with 5-shot examples. We visualize each token, shaded by its speculation rate, generated by the speculative execution model. The static approximation model is unable to switch to the second question during generation, hence diverging. The diverging point is a token with a low speculation rate, where it changes the topic from Question 1 to 2. Similarly, we mark the points where topics from previous contexts were introduced and they overlap with tokens with small speculation rates. Overall, dynamic execution via ALSpec detects large approximation errors and adaptively recalibrates to maintain generation quality.

6. Conclusion

We presented ALSpec as a new paradigm for scaling LLM inference, addressing the shortcomings of existing tensor and data parallelism methods. By leveraging predictions of attention outputs, our approach overlaps computations to reduce latency while maintaining model accuracy. The successful implementation on TT-Metalium underscores the practicality of our method. While this work focuses on speculating attention, the underlying principles of speculative parallelism are general and hold potential for wider applicability. Future research will explore the generalization of this technique, particularly in mechanisms to repair and eliminate error accumulation, as well as generic op-level speculative parallelism, which extends these principles beyond attention to optimize a broader range of ops in LLMs.

Acknowledgments

We sincerely thank Utku Aydonat, Steven Hill, Brian Liu, Aster Plotnik, Gilead Posluns, Aditya Saigal, Angus Wu, and the anonymous reviewers for their helpful feedback. This work was supported in part by Tenstorrent through the use of N150 and GPU devices and funding from Fujitsu.

Impact Statement

This paper presents work whose goal is to advance the field of Machine Learning. There are many potential societal consequences of our work, none which we feel must be specifically highlighted here.

References

Almazrouei, E., Alobeidli, H., Alshamsi, A., Cappelli, A., Cojocaru, R., Debbah, M., Étienne Goffinet, Hesslow, D., Launay, J., Malartic, Q., Mazzotta, D., Noune, B., Pannier, B., and Penedo, G. The Falcon series of open language models. *arXiv preprint*, 2023. doi:10.48550/arXiv.2311.16867.

Barad, H., Achterberg, J., Chou, T. P., and Yu, J. Accelerated AI inference via dynamic execution methods. *arXiv preprint*, 2024. doi:10.48550/arXiv.2411.00853.

Beltagy, I., Peters, M. E., and Cohan, A. Longformer: The long-document transformer. *arXiv preprint*, 2020. doi:10.48550/arXiv.2004.05150.

Cai, J. Flashdecode on tenstorrent's wormhole architecture. Tech Report, Tenstorrent, 2024. URL https://github.com/tenstorrent/tt-metal/blob/main/tech_reports/FlashAttention/FlashDecode.md.

Castin, V., Ablin, P., and Peyré, G. How smooth is attention? In *Proc. International Conference on Machine Learning (ICML)*, volume 235, pp. 5817–5840. PMLR, 2024. URL https://proceedings.mlr.press/v235/castin24a.html.

Chitty-Venkata, K. T., Raskar, S., Kale, B., Ferdaus, F., Tanikanti, A., Raffenetti, K., Taylor, V., Emani, M., and Vishwanath, V. LLM-Inference-Bench: Inference benchmarking of large language models on AI accelerators. In *Proc. Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC24-W)*, pp. 1362–1379, 2024. doi:10.1109/SCW63240.2024.00178.

Dao, T. FlashAttention-2: Faster attention with better parallelism and work partitioning. In *Proc. International Conference on Learning Representations (ICLR)*, 2024. URL https://openreview.net/forum?id=mZn2Xyh9Ec.

Dao, T., Haziza, D., Massa, F., and Sizov, G. Flash-decoding for long-context inference. Stanford University Center for Research on Foundation Models Blog, Oct 2023. URL https://crfm.stanford.edu/2023/10/12/flashdecoding.html.

DeepSeek-AI, Liu, A., Feng, B., Xue, B., Wang, B., Wu, B., Lu, C., Zhao, C., Deng, C., Zhang, C., Ruan, C., Dai, D., Guo, D., Yang, D., Chen, D., Ji, D., Li, E., Lin, F., Dai, F., Luo, F., Hao, G., Chen, G., Li, G., Zhang, H., Bao, H., Xu, H., Wang, H., Zhang, H., Ding, H., Xin, H., Gao, H., Li, H., Qu, H., Cai, J. L., Liang, J., Guo, J., Ni, J., Li, J., Wang, J., Chen, J., Chen, J., Yuan, J., Qiu, J., Li, J., Song, J., Dong, K., Hu, K., Gao, K., Guan, K., Huang, K., Yu, K., Wang, L., Zhang, L., Xu, L., Xia, L., Zhao, L., Wang, L., Zhang, L., Li, M., Wang, M., Zhang, M., Zhang, M., Tang, M., Li, M., Tian, N., Huang, P., Wang, P., Zhang, P., Wang, Q., Zhu, Q., Chen, Q., Du, Q., Chen, R. J., Jin, R. L., Ge, R., Zhang, R., Pan, R., Wang, R., Xu, R., Zhang, R., Chen, R., Li, S. S., Lu, S., Zhou, S., Chen, S., Wu, S., Ye, S., Ye, S., Ma, S., Wang, S., Zhou, S., Yu, S., Zhou, S., Pan, S., Wang, T., Yun, T., Pei, T., Sun, T., Xiao, W. L., Zeng, W., Zhao, W., An, W., Liu, W., Liang, W., Gao, W., Yu, W., Zhang, W., Li, X. Q., Jin, X., Wang, X., Bi, X., Liu, X., Wang, X., Shen, X., Chen, X., Zhang, X., Chen, X., Nie, X., Sun, X., Wang, X., Cheng, X., Liu, X., Xie, X., Liu, X., Yu, X., Song, X., Shan, X., Zhou, X., Yang, X., Li, X., Su, X., Lin, X., Li, Y. K., Wang, Y. Q., Wei, Y. X., Zhu, Y. X., Zhang, Y., Xu, Y., Xu, Y., Huang, Y., Li, Y., Zhao, Y., Sun, Y., Li, Y., Wang, Y., Yu, Y., Zheng, Y., Zhang, Y., Shi, Y., Xiong, Y., He, Y., Tang, Y., Piao, Y., Wang, Y., Tan, Y., Ma, Y., Liu, Y., Guo, Y., Wu, Y., Ou, Y., Zhu, Y., Wang, Y., Gong, Y., Zou, Y., He, Y., Zha, Y., Xiong, Y., Ma, Y., Yan, Y., Luo, Y., You, Y., Liu, Y., Zhou, Y., Wu, Z. F., Ren, Z. Z., Ren, Z., Sha, Z., Fu, Z., Xu, Z., Huang, Z., Zhang, Z., Xie, Z., Zhang, Z., Hao, Z., Gou, Z., Ma, Z., Yan, Z., Shao, Z., Xu, Z., Wu, Z., Zhang, Z., Li, Z., Gu, Z., Zhu, Z., Liu, Z., Li, Z., Xie, Z., Song, Z., Gao, Z., and Pan, Z. Deepseek-v3 technical report. *arXiv preprint*, 2024. doi:10.48550/arXiv.2412.19437.

Dubey, A., Jauhri, A., Pandey, A., Kadian, A., Al-Dahle, A., Letman, A., Mathur, A., Schelten, A., Yang, A., Fan, A., Goyal, A., Hartshorn, A., Yang, A., Mitra, A., Sravankumar, A., Korenev, A., Hinsvark, A., Rao, A., Zhang, A., Rodriguez, A., Gregerson, A., Spataru, A., Roziere, B., Biron, B., Tang, B., Chern, B., Caucheteux, C., Nayak, C., Bi, C., Marra, C., McConnell, C., Keller, C., Touret, C., Wu, C., Wong, C., Ferrer, C. C., Nikolaidis, C., Allonsius, D., Song, D., Pintz, D., Livshits, D., Esiobu, D., Choudhary, D., Mahajan, D., Garcia-Olano, D., Perino, D., Hupkes, D., Lakomkin, E., AlBadawy, E., Lobanova, E., Dinan, E., Smith, E. M., Radenovic, F., Zhang, F., Synnaeve, G., Lee, G., Anderson, G. L., Nail, G., Mialon, G., Pang, G., Cucurell, G., Nguyen, H., Korevaar, H., Xu, H., Touvron, H., Zarov, I., Ibarra, I. A., Kloumann, I., Misra, I., Evtimov, I., Copet, J., Lee, J., Geffert, J., Vranes, J., Park, J., Mahadeokar, J., Shah, J., van der Linde, J., Billock, J., Hong, J., Lee, J., Fu, J., Chi, J., Huang, J., Liu, J., Wang, J., Yu, J., Bitton, J., Spisak, J., Park, J., Rocca, J., Johnstun, J., Saxe, J., Jia, J., Alwala, K. V., Upasani, K., Plawiak, K., Li, K., Heafield, K., Stone, K., El-Arini, K., Iyer, K., Malik, K., Chiu, K., Bhalla, K., Rantala-Yeary, L., van der Maaten, L., Chen, L., Tan, L., Jenkins, L., Martin, L., Madaan, L., Malo, L., Blecher, L., Landzaat, L., de Oliveira, L., Muzzi, M., Pasupuleti, M., Singh, M., Paluri, M., Kardas, M., Oldham, M., Rita, M., Pavlova, M., Kambadur, M., Lewis, M., Si, M., Singh, M. K., Hassan, M., Goyal, N., Torabi, N., Bashlykov, N., Bogoychev, N., Chatterji, N., Duchenne, O., Çelebi, O., Alrassy, P., Zhang, P., Li, P., Vasic, P., Weng, P., Bhargava, P., Dubal, P., Krishnan, P., Koura, P. S., Xu, P., He, Q., Dong, Q., Srinivasan, R., Ganapathy, R., Calderer, R., Cabral, R. S., Stojnic, R., Raileanu, R., Girdhar, R., Patel, R., Sauvestre, R., Polidoro, R., Sumbaly, R., Taylor, R., Silva, R., Hou, R., Wang, R., Hosseini, S., Chennabasappa, S., Singh, S., Bell, S., Kim, S. S., Edunov, S., Nie, S., Narang, S., Raparthy, S., Shen, S., Wan, S., Bhosale, S., Zhang, S., Vandenhende, S., Batra, S., Whitman, S., Sootla, S., Collot, S., Gururangan, S., Borodinsky, S., Herman, T., Fowler, T., Sheasha, T., Georgiou, T., Scialom, T., Speckbacher, T., Mihaylov, T., Xiao, T., Karn, U., Goswami, V., Gupta, V., Ramanathan, V., Kerkez, V., Gonguet, V., Do, V., Vogeti, V., Petrovic, V., Chu, W., Xiong, W., Fu, W., Meers, W., Martinet, X., Wang, X., Tan, X. E., Xie, X., Jia, X., Wang, X., Goldschlag, Y., Gaur, Y., Babaei, Y., Wen, Y., Song, Y., Zhang, Y., Li, Y., Mao, Y., Coudert, Z. D., Yan, Z., Chen, Z., Papakipos, Z., Singh, A., Grattafiori, A., Jain, A., Kelsey, A., Shajnfeld, A., Gangidi, A., Victoria, A., Goldstand, A., Menon, A., Sharma, A., Boesenberg, A., Vaughan, A., Baevski, A., Feinstein, A., Kallet, A., Sangani, A., Yunus, A., Lupu, A., Alvarado, A., Caples, A., Gu, A., Ho, A., Poulton, A., Ryan, A., Ramchandani, A., Franco, A., Saraf, A., Chowdhury, A., Gabriel, A., Bharambe, A., Eisenman, A., Yazdan, A., James, B., Maurer, B., Leonhardi, B., Huang, B., Loyd, B., Paola, B. D., Paranjape, B., Liu, B., Wu, B., Ni, B., Hancock, B., Wasti, B., Spence, B., Stojkovic, B., Gamido, B., Montalvo, B., Parker, C., Burton, C., Mejia, C., Wang, C., Kim, C., Zhou, C., Hu, C., Chu, C.-H., Cai, C., Tindal, C., Feichtenhofer, C., Civin, D., Beaty, D., Kreymer, D., Li, D., Wyatt, D., Adkins, D., Xu, D., Testuggine, D., David, D., Parikh, D., Liskovich, D., Foss, D., Wang, D., Le, D., Holland, D., Dowling, E., Jamil, E., Montgomery, E., Presani, E., Hahn, E., Wood, E., Brinkman, E., Arcaute, E., Dunbar, E., Smothers, E., Sun, F., Kreuk, F., Tian, F., Ozgenel, F., Caggioni, F., Guzmán, F., Kanayet, F., Seide, F., Florez, G. M., Schwarz, G., Badeer, G., Swee, G., Halpern, G., Thattai, G., Herman, G., Sizov, G., Guangyi, Zhang, Lakshminarayanan, G., Shojanazeri, H., Zou, H., Wang, H., Zha, H., Habeeb, H., Rudolph, H., Suk, H., Aspegren, H., Goldman, H., Damlaj, I., Molybog, I., Tufanov, I., Veliche, I.-E., Gat, I., Weissman, J., Geboski, J., Kohli, J., Asher, J., Gaya, J.-B., Marcus, J., Tang, J., Chan, J., Zhen, J., Reizenstein, J., Teboul, J., Zhong, J., Jin, J., Yang, J., Cummings, J., Carvill, J., Shepard, J., McPhie, J., Torres, J., Ginsburg, J., Wang, J., Wu, K., U, K. H., Saxena, K., Prasad, K., Khandelwal, K., Zand, K., Matosich, K., Veeraraghavan, K., Michelena, K., Li, K., Huang, K., Chawla, K., Lakhotia, K., Huang, K., Chen, L., Garg, L., A, L., Silva, L., Bell, L., Zhang, L., Guo, L., Yu, L., Moshkovich, L., Wehrstedt, L., Khabsa, M., Avalani, M., Bhatt, M., Tsimpoukelli, M., Mankus, M., Hasson, M., Lennie, M., Reso, M., Groshev, M., Naumov, M., Lathi, M., Keneally, M., Seltzer, M. L., Valko, M., Restrepo, M., Patel, M., Vyatskov, M., Samvelyan, M., Clark, M., Macey, M., Wang, M., Hermoso, M. J., Metanat, M., Rastegari, M., Bansal, M., Santhanam, N., Parks, N., White, N., Bawa, N., Singhal, N., Egebo, N., Usunier, N., Laptev, N. P., Dong, N., Zhang, N., Cheng, N., Chernoguz, O., Hart, O., Salpekar, O., Kalinli, O., Kent, P., Parekh, P., Saab, P., Balaji, P., Rittner, P., Bontrager, P., Roux, P., Dollar, P., Zvyagina, P., Ratanchandani, P., Yuvraj, P., Liang, Q., Alao, R., Rodriguez, R., Ayub, R., Murthy, R., Nayani, R., Mitra, R., Li, R., Hogan, R., Battey, R., Wang, R., Maheswari, R., Howes, R., Rinott, R., Bondu, S. J., Datta, S., Chugh, S., Hunt, S., Dhillon, S., Sidorov, S., Pan, S., Verma, S., Yamamoto, S., Ramaswamy, S., Lindsay, S., Lindsay, S., Feng, S., Lin, S., Zha, S. C., Shankar, S., Zhang, S., Zhang, S., Wang, S., Agarwal, S., Sajuyigbe, S., Chintala, S., Max, S., Chen, S., Kehoe, S., Satterfield, S., Govindaprasad, S., Gupta, S., Cho, S., Virk, S., Subramanian, S., Choudhury, S., Goldman, S., Remez, T., Glaser, T., Best, T., Kohler, T., Robinson, T., Li, T., Zhang, T., Matthews, T., Chou, T., Shaked, T., Vontimitta, V., Ajayi, V., Montanez, V., Mohan, V., Kumar, V. S., Mangla, V., Albiero, V., Ionescu, V., Poenaru, V., Mihailescu, V. T., Ivanov, V., Li, W., Wang, W., Jiang, W., Bouaziz, W., Constable, W., Tang, X., Wang, X., Wu, X., Wang, X., Xia, X., Wu, X., Gao, X., Chen, Y., Hu, Y., Jia, Y., Qi, Y., Li, Y., Zhang, Y., Zhang, Y., Adi, Y., Nam, Y., Yu, Wang, Hao, Y., Qian, Y., He, Y., Rait, Z., DeVito, Z., Rosnbrick, Z., Wen, Z., Yang, Z., and Zhao, Z. The llama 3 herd of models. *arXiv preprint*, 2024. doi:10.48550/arXiv.2407.21783.

Elbayad, M., Gu, J., Grave, E., and Auli, M. Depth-adaptive transformer. In *Proc. International Conference on Learn-*

ing Representations (ICLR), 2020. URL https://openreview.net/forum?id=SJg7KhVKPH.

Elhoushi, M., Shrivastava, A., Liskovich, D., Hosmer, B., Wasti, B., Lai, L., Mahmoud, A., Acun, B., Agarwal, S., Roman, A., Aly, A., Chen, B., and Wu, C.-J. Layer-Skip: Enabling early exit inference and self-speculative decoding. In *Proc. Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 12622–12642, 2024. doi:10.18653/v1/2024.acl-long.681.

Eyuboglu, S., Zinsley, D., Saad-Falcon, J., Arora, S., Rudra, A., Zou, J., and Re, C. Towards smaller language models via layer looping. In *Proc. Workshop on Efficient Systems for Foundation Models II @ ICML2024*, 2024. URL https://openreview.net/forum?id=2N3CtUdoB0.

Gaiser, J., Fontaine, D., Hoffman, H., Jones, S., and Oh, F. Dynamic control flow in CUDA graphs with conditional nodes. NVIDIA Developer Technical Blog, May 2024.

Gao, L., Tow, J., Abbasi, B., Biderman, S., Black, S., DiPofi, A., Foster, C., Golding, L., Hsu, J., Le Noac'h, A., Li, H., McDonell, K., Muennighoff, N., Ociepa, C., Phang, J., Reynolds, L., Schoelkopf, H., Skowron, A., Sutawika, L., Tang, E., Thite, A., Wang, B., Wang, K., and Zou, A. The language model evaluation harness, July 2024.

Geshkovski, B., Letrouit, C., Polyanskiy, Y., and Rigollet, P. The emergence of clusters in self-attention dynamics. In *Proc. Conference on Neural Information Processing Systems (NeurIPS)*, volume 36, pp. 57026–57037, 2023. URL https://openreview.net/forum?id=aMjaEkkXJx.

Gu, A. and Dao, T. Linear-time sequence modeling with selective state spaces. In *Proc. Conference on Language Modeling (COLM)*, 2024. URL https://openreview.net/forum?id=tEYskw1VY2.

Hao, S., Sukhbaatar, S., Su, D., Li, X., Hu, Z., Weston, J., and Tian, Y. Training large language models to reason in a continuous latent space. *arXiv preprint*, 2024. doi:10.48550/arXiv.2412.06769.

Herlihy, M. and Moss, J. E. B. Transactional memory: architectural support for lock-free data structures. In *Proc. International Symposium on Computer Architecture (ISCA)*, pp. 289–300. ACM/IEEE, 1993. doi:10.1145/165123.165164.

Hooper, C., Kim, S., Mohammadzadeh, H., Genc, H., Keutzer, K., Gholami, A., and Shao, S. SPEED: Speculative pipelined execution for efficient decoding. In *Proc. NeurIPS Workshop on Efficient Natural Language and Speech Processing (ENLSP)*, 2023. doi:10.48550/arXiv.2310.12072.

Huang, Y., Cheng, Y., Bapna, A., Firat, O., Chen, D., Chen, M., Lee, H., Ngiam, J., Le, Q. V., Wu, Y., and Chen, z. GPipe: Efficient training of giant neural networks using pipeline parallelism. In *Proc. Conference on Neural Information Processing Systems (NeurIPS)*, volume 32, 2019. URL https://proceedings.neurips.cc/paper_files/paper/2019/file/093f65e080a295f8076b1c5722a46aa2-Paper.pdf.

Jeffrey, M. C., Subramanian, S., Yan, C., Emer, J., and Sanchez, D. A scalable architecture for ordered parallelism. In *Proc. International Symposium on Microarchitecture (MICRO)*, pp. 228–241. IEEE/ACM, 2015. doi:10.1145/2830772.2830777.

Jeffrey, M. C., Ying, V. A., Subramanian, S., Lee, H. R., Emer, J., and Sanchez, D. Harmonizing speculative and non-speculative execution in architectures for ordered parallelism. In *Proc. International Symposium on Microarchitecture (MICRO)*, pp. 217–230. IEEE/ACM, 2018. doi:10.1109/MICRO.2018.00026.

Jiang, A. Q., Sablayrolles, A., Roux, A., Mensch, A., Savary, B., Bamford, C., Chaplot, D. S., de Las Casas, D., Hanna, E. B., Bressand, F., Lengyel, G., Bour, G., Lample, G., Lavaud, L. R., Saulnier, L., Lachaux, M., Stock, P., Subramanian, S., Yang, S., Antoniak, S., Scao, T. L., Gervet, T., Lavril, T., Wang, T., Lacroix, T., and Sayed, W. E. Mixtral of experts. *arXiv preprint*, 2024. doi:10.48550/ARXIV.2401.04088.

Katharopoulos, A., Vyas, A., Pappas, N., and Fleuret, F. Transformers are RNNs: Fast autoregressive transformers with linear attention. In *Proc. International Conference on Machine Learning (ICML)*, volume 119, pp. 5156–5165. PMLR, 2020. URL https://proceedings.mlr.press/v119/katharopoulos20a.html.

Kim, H., Papamakarios, G., and Mnih, A. The Lipschitz constant of self-attention. In *Proc. International Conference on Machine Learning (ICML)*, volume 139, pp. 5562–5571. PMLR, 2021. URL https://proceedings.mlr.press/v139/kim21i.html.

Kim, S., Hooper, C., Wattanawong, T., Kang, M., Yan, R., Genc, H., Dinh, G., Huang, Q., Keutzer, K., Mahoney, M. W., Shao, Y. S., and Gholami, A. Full stack optimization of transformer inference: a survey. In *Proc. Workshop on Architecture and System Support for Transformer Models (ASSYST)*, 2023. URL https://openreview.net/forum?id=GtyQbLUUagE.

Kitaev, N., Kaiser, L., and Levskaya, A. Reformer: The efficient transformer. In *Proc. International Conference on Learning Representations (ICLR)*, 2020. URL https://openreview.net/forum?id=rkgNKkHtvB.

Leviathan, Y., Kalman, M., and Matias, Y. Fast inference from transformers via speculative decoding. In *Proc. International Conference on Machine Learning (ICML)*, volume 202, pp. 19274–19286. PMLR, 2023. URL `https://proceedings.mlr.press/v202/leviathan23a.html`.

Li, Y., Wei, F., Zhang, C., and Zhang, H. EAGLE: Speculative sampling requires rethinking feature uncertainty. In *Proc. International Conference on Machine Learning (ICML)*, volume 235, pp. 28935–28948. PMLR, 2024. URL `https://proceedings.mlr.press/v235/li24bt.html`.

Lipasti, M. and Shen, J. Exceeding the dataflow limit via value prediction. In *Proc. International Symposium on Microarchitecture (MICRO)*, pp. 226–237. IEEE/ACM, 1996. doi:10.1109/MICRO.1996.566464.

Miao, X., Oliaro, G., Zhang, Z., Cheng, X., Wang, Z., Zhang, Z., Wong, R. Y. Y., Zhu, A., Yang, L., Shi, X., Shi, C., Chen, Z., Arfeen, D., Abhyankar, R., and Jia, Z. SpecInfer: Accelerating large language model serving with tree-based speculative inference and verification. In *Proc. International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 932–949. ACM, 2024. doi:10.1145/3620666.3651335.

Naor, A. On the banach-space-valued azuma inequality and small-set isoperimetry of alon–roichman graphs. *Combinatorics, Probability and Computing*, 21(4):623–634, 2012. doi:10.1017/S0963548311000757.

Narayanan, D., Shoeybi, M., Casper, J., LeGresley, P., Patwary, M., Korthikanti, V., Vainbrand, D., Kashinkunti, P., Bernauer, J., Catanzaro, B., Phanishayee, A., and Zaharia, M. Efficient large-scale language model training on GPU clusters using Megatron-LM. In *Proc. of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. ACM, 2021. doi:10.1145/3458817.3476209.

Pagliardini, M., Paliotta, D., Jaggi, M., and Fleuret, F. Fast attention over long sequences with dynamic sparse flash attention. In *Proc. Conference on Neural Information Processing Systems (NeurIPS)*, volume 36, pp. 59808–59831, 2023. URL `https://openreview.net/forum?id=UINHuKeWUa`.

Poli, M., Massaroli, S., Nguyen, E., Fu, D. Y., Dao, T., Baccus, S., Bengio, Y., Ermon, S., and Re, C. Hyena hierarchy: Towards larger convolutional language models. In *Proc. International Conference on Machine Learning (ICML)*, volume 202, pp. 28043–28078. PMLR, 2023. URL `https://proceedings.mlr.press/v202/poli23a.html`.

Rau, B. R. and Fisher, J. A. Instruction-level parallel processing: History, overview, and perspective. *The Journal of Supercomputing*, 7(1):9–50, 1993. doi:10.1007/BF01205181.

Sazeides, Y. and Smith, J. The predictability of data values. In *Proc. International Symposium on Microarchitecture (MICRO)*, pp. 248–258. IEEE/ACM, 1997. doi:10.1109/MICRO.1997.645815.

Scaman, K. and Virmaux, A. Lipschitz regularity of deep neural networks: analysis and efficient estimation. In *Proc. Conference on Neural Information Processing Systems (NeurIPS)*, volume 31, 2018. URL `https://proceedings.neurips.cc/paper_files/paper/2018/file/d54e99a6c03704e95e6965532dec148b-Paper.pdf`.

Sergeev, A. and Balso, M. D. Horovod: fast and easy distributed deep learning in TensorFlow. *arXiv preprint*, 2018. doi:10.48550/arXiv.1802.05799.

Shoeybi, M., Patwary, M., Puri, R., LeGresley, P., Casper, J., and Catanzaro, B. Megatron-LM: Training multi-billion parameter language models using model parallelism. *arXiv preprint*, 2019. doi:10.48550/arXiv.1909.08053.

Sohi, G. S., Breach, S. E., and Vijaykumar, T. N. Multiscalar processors. In *Proc. International Symposium on Computer Architecture (ISCA)*, pp. 414–425. ACM/IEEE, 1995. doi:10.1145/223982.224451.

Spector, B. F. and Re, C. Accelerating LLM inference with staged speculative decoding. In *Workshop on Efficient Systems for Foundation Models @ ICML2023*, 2023. URL `https://openreview.net/forum?id=RKHF3VYjLK`.

Sun, H., Chen, Z., Yang, X., Tian, Y., and Chen, B. Triforce: Lossless acceleration of long sequence generation with hierarchical speculative decoding. In *Proc. Conference on Language Modeling (COLM)*, 2024. URL `https://openreview.net/forum?id=HVK6nl3i97`.

Tay, Y., Dehghani, M., Bahri, D., and Metzler, D. Efficient transformers: A survey. *ACM Computing Surveys*, 55(6), 2022. doi:10.1145/3530811.

Tenstorrent. Llama3 models. GitHub Repository, 2024a. URL `https://github.com/tenstorrent/tt-metal/blob/v0.55.0-rc13/models/demos/llama3/README.md`.

Tenstorrent. TT-Metallium, 2024b. URL `https://tenstorrent.com/software/tt-metalium`.

Tenstorrent. *TT-Metalium™ Guide*, 2024c. URL https://github.com/tenstorrent/tt-metal/blob/v0.55.0-rc13/METALIUM_GUIDE.md.

Thakur, R., Rabenseifner, R., and Gropp, W. Optimization of collective communication operations in MPICH. *The International Journal of High Performance Computing Applications*, 19(1):49–66, 2005. doi:10.1177/1094342005051521.

Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., Rodriguez, A., Joulin, A., Grave, E., and Lample, G. Llama: Open and efficient foundation language models. *arXiv preprint*, 2023. doi:10.48550/arXiv.2302.13971.

Vasiljevic, J., Bajic, L., Capalija, D., Sokorac, S., Ignjatovic, D., Bajic, L., Trajkovic, M., Hamer, I., Matosevic, I., Cejkov, A., Aydonat, U., Zhou, T., Gilani, S. Z., Paiva, A., Chu, J., Maksimovic, D., Chin, S. A., Moudallal, Z., Rakhmati, A., Nijjar, S., Bhullar, A., Drazic, B., Lee, C., Sun, J., Kwong, K.-M., Connolly, J., Dooley, M., Farooq, H., Chen, J. Y. T., Walker, M., Dabiri, K., Mabee, K., Lal, R. S., Rajatheva, N., Retnamma, R., Karodi, S., Rosen, D., Munoz, E., Lewycky, A., Knezevic, A., Kim, R., Rui, A., Drouillard, A., and Thompson, D. Compute substrate for software 2.0. *IEEE Micro*, 41(2):50–55, 2021. doi:10.1109/MM.2021.3061912.

Xiao, G., Tian, Y., Chen, B., Han, S., and Lewis, M. Efficient streaming language models with attention sinks. In *Proc. International Conference on Learning Representations (ICLR)*, 2024. URL https://openreview.net/forum?id=NG7sS51zVF.

Yang, A., Yang, B., Hui, B., Zheng, B., Yu, B., Zhou, C., Li, C., Li, C., Liu, D., Huang, F., Dong, G., Wei, H., Lin, H., Tang, J., Wang, J., Yang, J., Tu, J., Zhang, J., Ma, J., Yang, J., Xu, J., Zhou, J., Bai, J., He, J., Lin, J., Dang, K., Lu, K., Chen, K., Yang, K., Li, M., Xue, M., Ni, N., Zhang, P., Wang, P., Peng, R., Men, R., Gao, R., Lin, R., Wang, S., Bai, S., Tan, S., Zhu, T., Li, T., Liu, T., Ge, W., Deng, X., Zhou, X., Ren, X., Zhang, X., Wei, X., Ren, X., Liu, X., Fan, Y., Yao, Y., Zhang, Y., Wan, Y., Chu, Y., Liu, Y., Cui, Z., Zhang, Z., Guo, Z., and Fan, Z. Qwen2 technical report. *arXiv preprint*, 2024. doi:10.48550/arXiv.2407.10671.

Ye, Z., Chen, L., Lai, R., Lin, W., Zhang, Y., Wang, S., Chen, T., Kasikci, B., Grover, V., Krishnamurthy, A., and Ceze, L. Flashinfer: Efficient and customizable attention engine for llm inference serving. In *Proc. Conference on Machine Learning and Systems (MLSys)*, 2025. URL https://openreview.net/forum?id=RXPofAsL8F.

Zheng, L., Yin, L., Xie, Z., Sun, C., Huang, J., Yu, C. H., Cao, S., Kozyrakis, C., Stoica, I., Gonzalez, J. E., Barrett, C., and Sheng, Y. SGLang: Efficient execution of structured language model programs. In *Proc. Conference on Neural Information Processing Systems (NeurIPS)*, 2024. URL https://openreview.net/forum?id=VqkAKQibpq.

A. A Proof for Verification Error Bound

Error bound on speculation steps is essential to rigorously quantify the trade-offs between efficiency and accuracy. Such bounds ensure that the simplified model maintains acceptable performance levels while providing theoretical guarantees on the maximum potential loss in accuracy.

We accept the speculated attention output when its value is close enough to the ground truth (i.e., within a threshold δ) in some metric space. To find this appropriate metric space and its corresponding δ, however, requires search in an almost infinite space if done empirically.

Instead, we frame the problem as follows:

On a model with M speculation points, given some metric d, for $\epsilon > 0$, for each speculation point $i \in 1...N$, find threshold δ_i such that for every ground truth hidden state H_i and speculated hidden state $\tilde{H}_i$, if $d(H_i, \tilde{H}_i) < \delta_i$ for all i, then the distance between the final output $d(H_N, \tilde{H_N}) < \epsilon$.

To derive the relationship between δ and ϵ, we apply the Lipschitz regularity of neural network (Scaman & Virmaux, 2018). We consider the Llama (Touvron et al., 2023) architecture, consists of layer normalization, self-attention, feed-forward neural networks, and residual connections. When the self-attention output A_i is being speculated as $\tilde{A}_i$, a verifier with threshold δ_i guarantees $d(A_i, \tilde{A}_i) < \delta_i$, otherwise the speculation would be rejected.

A.1. Worst Case Error Bound

Since we are interested in bounding the error in $\mathbb{R}^n$ which is a Banach space, we denoted the distance function $d(A, B)$ as $\|A - B\|_p$ that is the p-norm of $\mathbb{R}^d$. Note that the norm induced metrics are translation invariant in Banach spaces:

$$\|X - Y\| = \|(X + A) - (Y + A)\|, \ \ \forall X, Y, A \in \mathbb{R}^d$$

Those metrics also follow triangular inequality:

$$\|X - Y\| + \|X - Z\| \leq \|X - Z\|, \ \ \forall X, Y, Z \in \mathbb{R}^d$$

Follow the same assumption as in Section 3.2, suppose that for each layer, the self-attention output speculation error is bounded by $\|A_i - \tilde{A}_i\| < \delta_i$. Then we can easily obtain the bound for residual connections:

$$\|(H_i + A_i) - (H_i + \tilde{A}_i)\| = \|A_i - \tilde{A}_i\| < \delta_i$$

Following the residual connection of self-attention is the LayerNorm and MLP. Since both LayerNorm and MLP are Lipschitz continuous, their composite is also Lipschizt continuous.

$$\|NormMLP(A_i) - NormMLP(\tilde{A}_i))\| \leq \alpha_i \|A_i - \tilde{A}_i\|$$

It follows

$$\begin{aligned}
&\left\|(A_i + NormMLP(A_i)) - \left(\tilde{A}_i + NormMLP(\tilde{A}_i)\right)\right\| \\
=&\left\|(A_i - \tilde{A}_i) + \left(NormMLP(A_i)) - NormMLP(\tilde{A}_i)\right)\right\| \\
\leq&\left\|(A_i - \tilde{A}_i)\right\| + \left\|\left(NormMLP(A_i)) - NormMLP(\tilde{A}_i)\right)\right\| \\
<&\delta_i + \alpha_i \delta_i \\
=&(1 + \alpha_i)\delta_i
\end{aligned}$$

At the next speculation point (the self-attention in layer $i + 1$) the non-speculative and speculative inputs are H_{i+1} and $\tilde{H}_{i+1}$ respectively. We denote self-attention outputs A_{i+1}, $\tilde{A}_{i+1}$, and $\tilde{\tilde{A}}_{i+1}$ as the non-speculative path output, speculative input followed by non-speculative output, and speculative input followed by speculative output respectively. We note that to obtain a bound for $d(A_{i+1}, \tilde{\tilde{A}}_{i+1})$, we can once again apply triangular inequality of the metric space and the result from (Castin et al., 2024), which the Lipschitz constant of self-attention, $f(R)$, is a function of the input magnitude R:

$$\begin{aligned}
&\|A_{i+1} - \tilde{\tilde{A}}_{i+1}\| \\
\leq& \|A_{i+1} - \tilde{A}_{i+1}\| + \|\tilde{A}_{i+1} - \tilde{\tilde{A}}_{i+1}\| \\
\leq& f(R)\|H_{i+1} - \tilde{H}_{i+1}\| + \delta_{i+1}
\end{aligned}$$

In general, let H_i and $\tilde{H}_i$ be the non-speculative and speculative input of layer i, and α_i and β_i be Lipschitz constants of layer i, then:

$$\begin{aligned}&\|H_{i+1} - \tilde{H}_{i+1}\| \leq (1+\alpha_i)((1+f(R)\beta_i)\|H_i - \tilde{H}_i\|\delta_i)\\ &\text{where } \|H_2 - \tilde{H}_2\| \leq (1+\alpha_1)\delta_1\end{aligned} \tag{2}$$

Simplifying Equation by assuming the same upper bound α and β across all layers, we obtain the following relationship between $\{\delta_i\}$ and ϵ:

$$\epsilon = \|H_N - \tilde{H}_N\| \leq \Sigma_{i=1}^{N}(1+\alpha)^{N-i+1}(1+f(R)\beta)^{N-i}\delta_i \tag{3}$$

A.2. Free choice of L_p and linear approximation of Equation 1

For any finite dimension Banach space, any two norms on the space are *equivalent* ($\approx$). Here we show a simplified proof for our norms L_p where $1 \leq p \leq \infty$. Let $x \in \mathbb{R}^n$ and $1 < p < r < \infty$.
When $x = \vec{0}$, this obviously holds. Consider $x \neq \vec{0}$. Then

$$\begin{aligned}\|x\|_r =& (\sum_{k=1}^{n}|x_k|^r)^{1/r}\\ \geq& ((\max_{k\in n}|x_k|)^r)^{1/r}\\ =& \max_{k\in n}|x_k|\\ =& \|x\|_\infty\end{aligned}$$

Note that

$$\begin{aligned}\frac{\|x\|_r^r}{\|x\|_\infty^r} =& \frac{\sum_{k=1}^{n}|x_k|^r}{(\max_{i\in n}|x_i|)^r}\\ =& \sum_{k=1}^{n}\frac{|x_k|^r}{(\max_{i\in n}|x_i|)^r}\\ =& \sum_{k=1}^{n}(\frac{|x_k|}{(\max_{i\in n}|x_i|)})^r\\ \leq& \sum_{k=1}^{n}(\frac{|x_k|}{(\max_{i\in n}|x_i|)})^p\\ =& \sum_{k=1}^{n}\frac{|x_k|^p}{(\max_{i\in n}|x_i|)^p}\\ =& \frac{\|x\|_p^p}{\|x\|_\infty^p}\end{aligned}$$

Which gives

$$\|x\|_r = \|x\|_\infty(\frac{\|x\|_r^r}{\|x\|_\infty^r})^{1/r} \leq \|x\|_\infty(\frac{\|x\|_p^p}{\|x\|_\infty^p})^{1/r} \leq \|x\|_\infty(\frac{\|x\|_p^p}{\|x\|_\infty^p})^{1/p} = \|x\|_p$$

Note this holds for $p, r > 0$. We can use the same argument to show that $\|x\|_p \leq \|x\|_1$. Also,

$$\begin{aligned}n\|x\|_\infty =& \sum_{k=1}^{n}\max_{i\in n}|x_i|\\ \geq& \sum_{k=1}^{n}|x_k|\\ =& \|x\|_1\end{aligned}$$

This shows that $\|x\|_\infty \leq \|x\|r \leq \|x\|_p \leq \|x\|_1 \leq n\|x\|_\infty$. Using this fact, the choice of l_p would at most differ by a constant of n that only depends on the dimensionality but not the magnitude of the input vector.

When α and β are small, we can approximate the error bound linearly. We have that

$$\begin{aligned}
\epsilon \leq & \Sigma_{i=1}^{N}(1+\alpha)^{N-i+1}(1+f(R)\beta)^{N-i}\delta_i \\
= & \sum_{i=1}^{N}\left(\sum_{j=0}^{N-i+1}\binom{N-i+1}{j}\alpha^{N-i+1-j}\right)\left(\sum_{j=0}^{N-i}\binom{N-i}{j}(f(R)\beta)^{N-i-j}\right)\delta_i \\
\approx & \sum_{i=1}^{N}\left(\sum_{j=0}^{N-i+1}\binom{N-i}{j}\alpha^{N-i-j}\right)\left(\sum_{j=0}^{N-i}\binom{N-i}{j}(f(R)\beta)^{N-i-j}\right)\delta_i \\
\approx & \sum_{i=1}^{N}\left(1+(N-i)\alpha\right)\left(1+(N-i)f(R)\beta\right)\delta_i \\
\approx & \sum_{i=1}^{N}\left(1+(N-i)(\alpha+f(R)\beta)\right)\delta_i \\
\approx & \left(N+\frac{1}{2}N^2\right)(\alpha+f(R)\beta)\delta
\end{aligned}$$

Which $\delta \geq \delta_i$ is an overall bound. This bound holds for the worst case; all errors are in the same direction. However, this is highly unlike in practice and it has a dependence on N^2. As a result, the threshold chosen using this bound for rejecting speculation is too small and not useful.

A.3. High Probability Error Bound

However, with stronger assumptions, we can obtain a better error bound. Assume that the errors in the speculation rounds are independent and follow some distribution with mean 0. In other words, the direction of the error will not be biased towards any direction. By rejecting errors that exceeding some bound, we ensure that speculated results are bounded by the ball centered on the ground truth x, with radius $\lambda \cdot \|x\|$. This is a martingale:

$$\mathbb{E}[\tilde{H}_{i+1} - H_{i+1} | \tilde{H}_i] = 0$$

With the causal softmax attention

$$H = V\text{softmax}(QK^T)$$

The approximation is obtained as

$$\tilde{A} = QK^T(:, B)$$

And

$$\tilde{H} = V(B, :)\text{softmax}(\tilde{A})$$

Where B is a subset of columns of K^T. With the error being bounded:

$$\|H_{i+1} - \tilde{H}_{i+1}\|_2 \leq \lambda\|\tilde{H}_i\|_2$$

With Azuma-type inequality for Banach space-valued martingales (Naor, 2012), there exists some universal constant $c > 0$ such that.

$$\mathbb{P}(\|\tilde{H}_N - H_N\|_2 \geq \epsilon) \leq e^4 \cdot \exp\left(-\frac{c\epsilon^2}{\sum_{i=1}^{N}\lambda^2\|\tilde{H}_i\|_2^2}\right)$$

Rearranging the equation, with probability $\delta \in (0, 1)$, we have

$$\epsilon \leq \sqrt{\frac{(4-\log\delta)\lambda^2\sum_{i=1}^{N}\|\tilde{H}_i\|_2^2}{c}}$$

Choose $\delta = 0.05$, with some addition assumption that $\|H_i\|_2$ are "on the same scale" due to layer norm, we replace them with a universal constant $\|H\|_2$. Now with probability 95%,

$$\begin{aligned}\|\tilde{H}_N - H_N\|_2 \leq& \sqrt{\frac{(4-\log 0.05)\lambda^2 N\|H\|_2^2}{c}} \\ \frac{\|\tilde{H}_N - H_N\|_2}{\|H\|_2} \leq& \lambda\sqrt{\frac{7N}{c}}\end{aligned}$$

This bound has much better dependence on N which is the total number of speculations than the last bound. However, it relies on unbiased assumption on speculation error that is yet to be proved. In practice, $\lambda = 0.1$ worked well in varies benchmarks and this bound is much closer to 0.1.

B. Speculative Flash Decode Algorithm

The speculative flash decode algorithm builds upon the standard Flash Attention 2 (Dao, 2024). In the scope of this work, we only consider the forward pass for inference application. During the decode stage, the activation query matrix ($\mathbf{Q}$), with sequence length 1, is very small and fits into the on-chip SRAM. The keys ($\mathbf{K}$) and values ($\mathbf{V}$) matices, particularly at large context length, are large and can only live in the slower DRAM. In flash decode, input $\mathbf{Q}$ is replicated across parallel workers and each worker computes partial attention output (with local statistics) using a portion of $\mathbf{K}$ and $\mathbf{V}$ by running the standard flash attention algorithm with chunk size S. Next, the partial outputs are gathered and reduced on the reducer worker using the local statistics by applying the online softmax trick. In the extreme case of having only a single worker, flash decode is the same as flash attention. In the algorithm below, we skip the details of how flash attention is computed and we simply call `O,m,l = flash_attention(Q,K,V)` to obtain the partial output $\mathbf{O}$ and statistics m & l.

Speculative flash decode builds upon the flash decode algorithm with a few modifications. The kernel consists of the sender, which computes the full and speculative attention outputs, and the receiver, which waits for the speculative results to arrive. First, one (or two) of the workers compute the partial output using first and last chunk of $\mathbf{K}$ and $\mathbf{V}$, which the result serves as the speculated attention output. Then, the worker continues to finish the rest of its own assigned $\mathbf{K}$ and $\mathbf{V}$ chunks. Finally, the reducer worker computes the full output using the partial outputs from each worker and the speculative output. We show speculative flash decode in Algorithm 3 below:

Algorithm 3 Speculative Flash Decode (forward pass for inference)

```
1:  **Require:**
2:    $n_{\text{workers}}$: total number of parallel workers
3:    $m$: total number of chunks
4:    $\mathbf{Q}$: query of size $1 \times d$ (fits in on-chip SRAM)
5:    $\{\mathbf{K}_1, \ldots, \mathbf{K}_m\}$: key chunks in DRAM
6:    $\{\mathbf{V}_1, \ldots, \mathbf{V}_m\}$: value chunks in DRAM
7:    Routines:
8:      flash_attention($\mathbf{Q}, [\mathbf{K}], [\mathbf{V}]) \to (\mathbf{O}, m, l)$
9:      send_to_receiver($\cdot$), send_to_output_worker($\cdot$)
10:     online_softmax_combine($\cdot$)
11: **// SENDER (run on each worker)**
12: chunks_per_worker $\leftarrow \lceil m/n_{\text{workers}} \rceil$
13: assigned_chunks $\leftarrow$ range of chunk indices for this worker, excluding 1 and $m$
14: **if** worker_id $= 1$ **then**
15:   $(\mathbf{O}_s, m_s, l_s) \leftarrow$ flash_attention$\big(\mathbf{Q}, [\mathbf{K}_1, \mathbf{K}_m], [\mathbf{V}_1, \mathbf{V}_m]\big)$ {Speculative output}
16:   send_to_receiver($\mathbf{O}_s$)
17:   send_to_output_worker($\mathbf{O}_s, m_s, l_s$)
18: **end if**
19: $(\mathbf{O}^{(i)}, m^{(i)}, l^{(i)}) \leftarrow$ flash_attention$\big(\mathbf{Q}, [\mathbf{K}_j \mid j \in$ assigned_chunks$], [\mathbf{V}_j \mid j \in$ assigned_chunks$]\big)$
20: send_to_output_worker($\mathbf{O}^{(i)}, m^{(i)}, l^{(i)}$)
21: **// OUTPUT WORKER (on sender)**
22: Gather all partial outputs $(\mathbf{O}^{(i)}, m^{(i)}, l^{(i)})$ from each worker
23: Gather speculative $(\mathbf{O}_s, m_s, l_s)$ from worker 1
24: $(\mathbf{O}, m, l) \leftarrow$ online_softmax_combine$\Big(\{(\mathbf{O}^{(i)}, m^{(i)}, l^{(i)})\}_{i=1}^{n_{\text{workers}}} \cup \{(\mathbf{O}_s, m_s, l_s)\}\Big)$
25: **return O**
26: **// RECEIVER**
27: **wait** for speculative output $\mathbf{O}_s$ to arrive from worker 1
```

C. Tenstorrent N150 Chips Specification and Tenstorrent Metalium (TT-Metalium) Framework

C.1. Tenstorrent Overview

Tenstorrent specializes in high-performance computing solutions for artificial intelligence (AI) and machine learning (ML) workloads. Their primary focus is on designing innovative hardware accelerators and software frameworks that enable efficient inference and training of deep learning models. By leveraging custom processor architectures optimized for AI workloads, Tenstorrent aims to push the boundaries of energy efficiency and computational throughput.

C.2. N150 Wormhole Chips

The N150 Wormhole chip is an AI accelerator designed for High Performance Computing (HPC) applications such as high-throughput and low-latency machine learning inference. These chips are designed to be highly flexible and scalable. Each chip contains a mesh of 72 Tensix^{TM} compute cores connected via 2 network-on-chip (NOCs), offering 262 Tera-FLOPs of FP8 performance. Moreover, with on-board ethernet, N150 chips can be interconnected in various topologies consisting of 2, 4, 8, and 32 N150 devices.

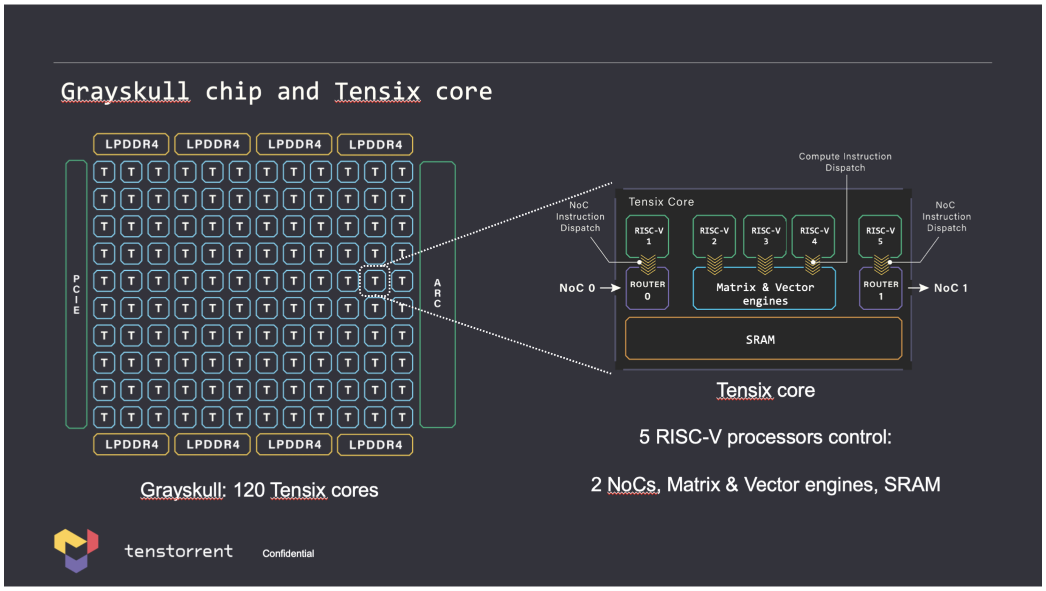

Grayskull chip and Tensix core

Figure 9. Architecture of a GraySkull^{TM} device, the predecessor to the Wormhole^{TM}. Image taken from (Tenstorrent, 2024c).

C.3. Tensix^{TM} Core

The Tensix^{TM} core is a crucial building block on the Wormhole device. It consists of 5 small RISC-V processors (also known as Baby RISCVs), that handle compute, data movement, and dispatch of instructions. For compute, a Matrix Engine (FPU) is used to perform various binary operations on small 32x32 matrices (known as tiles), such matrix-multiplication, dot-product, and other element-wise operations. Alongside, a Vector engine (SPU) is used for special operations such as GELU, exponential, and sqrt. For memory, each core has access to 1.5MB of high-bandwidth SRAM (L1) as well as 12GB of GDDR6 off-chip DRAM. A Data movement engine is also available on each core, connected to 2 NOCs. By creating a mesh of cores via these NOCs, the N150 provides access to a high capacity of 108 MB of SRAM, allowing for the Tensix^{TM} cores to operate at "silicon peak" performance. To allow users to leverage the powerful capabilities of the N150 chip, Tenstorrent has developed a low-level kernel library: TT-Metalium (Tenstorrent, 2024c).

C.4. TT-Metalium

TT-Metalium, also known as TT-Metalium, is an open-source low-level kernel operations library that allows users to enable high-performance machine learning applications on Tenstorrent hardware, such as the N150 chip. At a fundamental level, TT-Metalium assigns a 1:1 mapping between each Tensix^{TM} core and a thread, discarding the need for complex thread scheduling. Using TT-Metalium, users can leverage bare-metal programming to write kernels in C/C++ that run directly on each Tensix™ core and perform computation on tilized input tensors. With the fine-grained control enabled by the TT-Metalium framework, the mesh of Tensix™ cores on the N150 chip can be leveraged in a robust and powerful manner, overall supporting highly intensive machine learning workloads.

C.5. Kernel programming with TT-Metalium

Programming a Tensix™ core on TT-Metalium mainly consists of writing three kernels – a reader, writer, and compute. There are other advanced types of kernels available, such as dispatch and ethernet data movement kernels, however, they are not relevant to the user when bringing up general operations, and can be accessed via an abstraction layer. To implement an operation, the reader kernel is programmed to read data from L1 or DRAM memory and pass it off to the compute kernel via local buffers in L1. After the compute kernel has completed the operation (using the FPU/SFU), it packs out the data into another local buffer, where it is then accessed by the writer kernel. The writer kernel is responsible for formatting the data as required and providing outputs either in L1 or DRAM. As these three kernels run in parallel, there is native support for overlapping data movement with computation, where the reader/writer kernel can transfer data while the compute kernel is busy. It is also important to note that the compute kernel only supports tilized inputs to maximize utilization, and as such, the reader/writer kernels must format input tensors in this manner. Moreover, a set of data movement APIs, that leverage the NOC, are available inside each reader/writer kernel, allowing each core to access the L1 memory of any other core. Putting all of this together, TT-Metalium can be used to create efficient data patterns across the N150 chip, resulting in powerful implementations of operations used in machine learning.

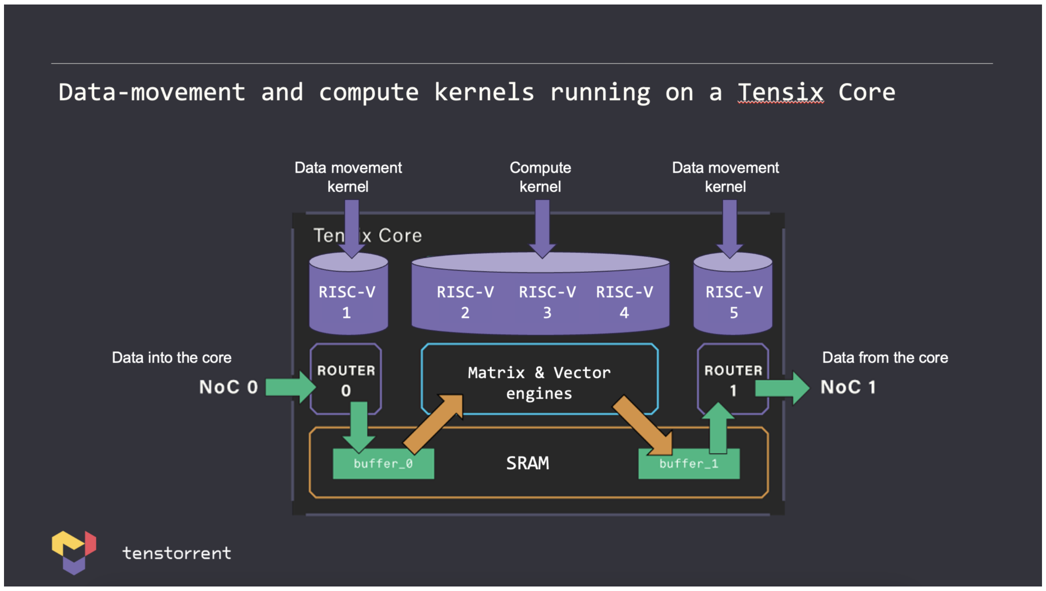


Figure 10. The workflow of a Tensix™ core. Image taken from (Tenstorrent, 2024c).

C.6. TT-NN and Execution Paradigm

Powered by TT-Metalium in the background, TT-NN is a user-friendly library implemented in Python that enables fast bring-up of machine learning models without sacrificing performance. It provides easy-to-use APIs to perform machine learning operations on N-dimensional tensors, similarly to PyTorch. Additionally, TT-Metalium objects such as a Multi-Device tensor are exposed in TT-NN, allowing users to easily implement algorithms and paradigms that span multiple N150 chips and leverage distributed computing. When running models implemented in TT-NN, operations are dispatched by the host and scheduled to run on Tenstorrent hardware via the PCIe link. Once results are ready on device, they are sent back to the host.

D. Speculative Flash Decode Implementation on TT-Metalium

We implement the speculative flash decode kernel on TT-Metalium based on the existing flash decode kernel implementation (Cai, 2024). All kernels on TT-Metalium are programmed directly on the Tensix$^{\text{TM}}$ cores to take advantage of the parallelism and tile-based granularity offered by the hardware. Each N150 chips consists of 64 worker cores and math is done on tiles with size 32×32. In the case of flash decode, we parallelize the batch and key/value heads across the cores, and put the query heads (usually ≤ 32) within a tile.

We illustrate this idea using Llama 3.1 8B's group query attention, which contains 32 query heads and 8 k/v heads, with head dimension 128. Assuming batch 1, during decode, query has sequence length of 1 so the shape of $\mathbf{Q}$ is $[S = 1, B = 1, H = 32, D = 128]$, and the shape of shape of $\mathbf{K}$ and $\mathbf{V}$ are $[B = 1, H = 8, S, D = 128]$. In this case, we parallelize the k/v heads across the 64 cores, with 8 cores per head. Each of the 8 cores then each takes a chunk of $\mathbf{K}$ and $\mathbf{V}$ fractured in the sequence length dimension and run the speculative flash decode algorithm (Appendix B). We illustrate the implementation in Figure 11:

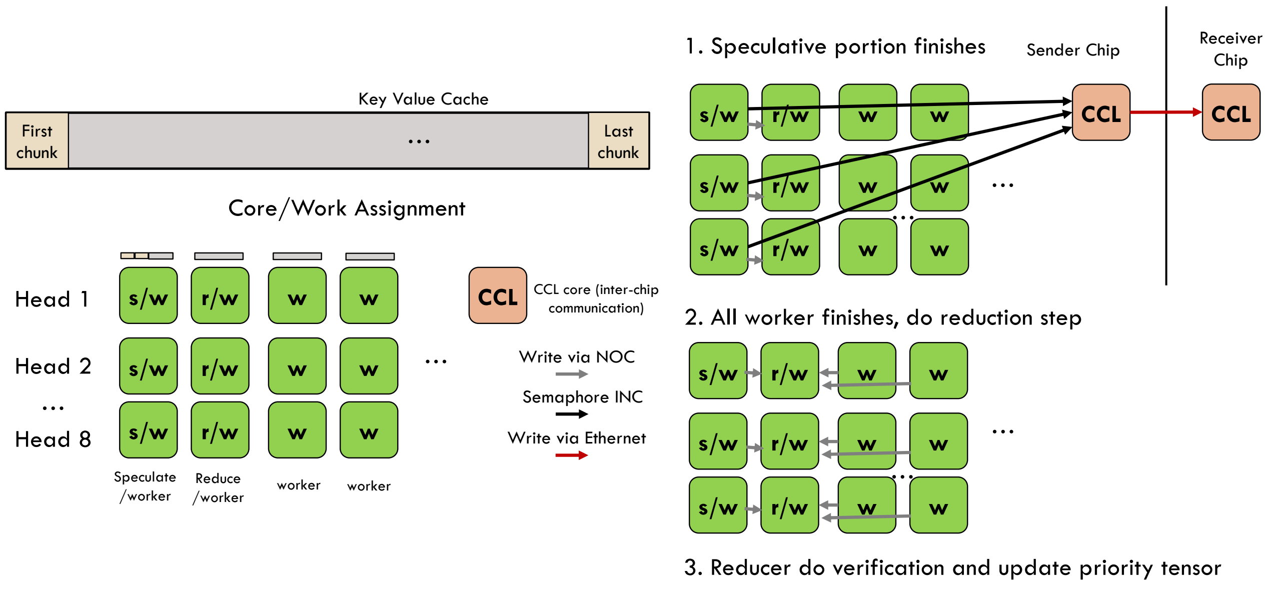


Figure 11. Speculative flash decode implementation on TT-Metalium. Each k/v head is split across 8 cores in the sequence length dimension. For each head, one core computes speculative and partial attention output, one core computes partial attention output and reduction, and the rest of the cores computes partial attention output only. When speculative results is finish, the worker core signals the CCL core to send result to the receiver chip via ethernet link. The rest follows the standard flash decode implementation, where partial outputs are gathered and reduced on the reduction core. In the end, we fused the verification step and update the priority vector.

Overall, speculative flash decode yields almost no overhead compared to standard flash decode in our performance measurement. The speculative portion computes towards the final output, and communication across chip is done on separate core via ethernet. When parallelizing attention across multiple devices using tensor parallelism, we split the k/v heads across the chip so that each head gets more workers per head. For example, splitting onto 2 chips gives 4 k/v heads per chip, and split across 8 chips gives 1 k/v head per chip. In practice, parallelizing across the heads yields diminishing returns when each chip receives "too little" work. In the case of having all 64 cores computing a single head, the bottleneck is NOC rather than compute due to the NOC transactions at the reduction step. For this reason, we see diminishing returns when scaling to 8 devices doing full tensor parallelism, and a better alternative would be doing 4 device tensor parallelism + ALSpec for scaling up to 8 devices. Figure 12 demonstrates speculative flash decode, sender & receiver side kernel durations, across context length for tensor parallel Llama 3.1 8B attention across 1,2,4 & 8 devices with chunk size $\{128, 256, 512\}$.

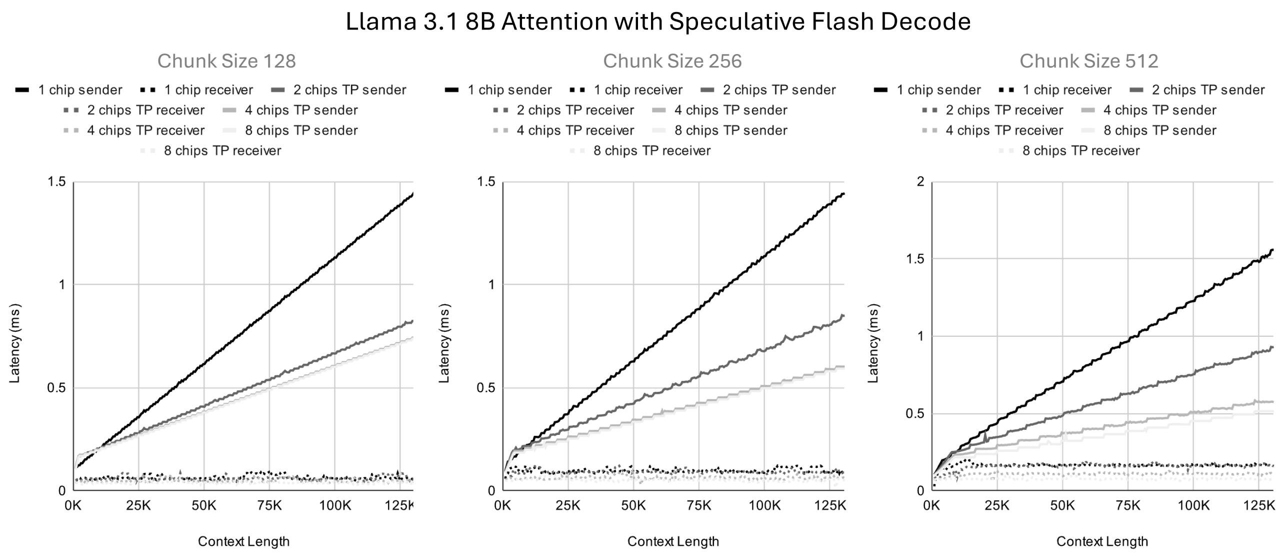


Figure 12. Speculative flash decode kernel duration for Llama 3.1 8B shape with chunk size $\{128, 256, 512\}$.

E. SGDC with Priority Gating Implementation on TT-Metalium

The static graph dynamic concurrency (SGDC) paradigm, refers to having a static operation graph generated by the host but having the devices execute concurrently based on a function, which in our case is the priority gating mechanism. For the speculative flash decode algorithm, the dynamic concurrency comes in the form of % speculation acceptance and the resulting overlap between attention and post-attention operations, which varies layer-to-layer and token-to-token. To support this paradigm, we made a change to the run-time behavior of the TT-Metalium framework.

We leverage the fact that any kernel for any operation (data movement, compute, ethernet) in TT-Metalium is launched through a small set of C++ kernel source code files. By passing a flag at the build stage of these kernel source code files, we can control the behaviour of the kernel launch stage. At the start of model execution, we create a multi-device priority vector ‘p‘, where each device tensor contains the device’s priority. This tensor will have the same lifetime as the entire model and is used as the gating mechanism. Then, we pass the base memory address of this tensor to the kernel source code files using a C++ define, *SKIP_COMPUTE*. When a kernel is launched, it first passes through the kernel source code, where the address specified by *SKIP_COMPUTE* is used to check whether the kernel should be launched. As described in the paper, if the priority of the device is 0, then the operation will be skipped. Note, the ability to use the *SKIP_COMPUTE* can be toggled by adding/removing the define from the build arguments of the kernel files. We make sure to remove the define before running the *speculative_flash_decode* kernel, so as to not skip this operation.

Using this mechanism, we can enable the behaviour described by scenario 3 in Figure 5. When device 1 is a sender device and finds that the speculated result is correct within a certain threshold, it can completely skip the post-attention operations using the *SKIP_COMPUTE* flag, relying on the concurrent computation performed on device 2. When viewed from the host, this results in a static operation graph, however at the device level, we have achieved dynamic concurrency.

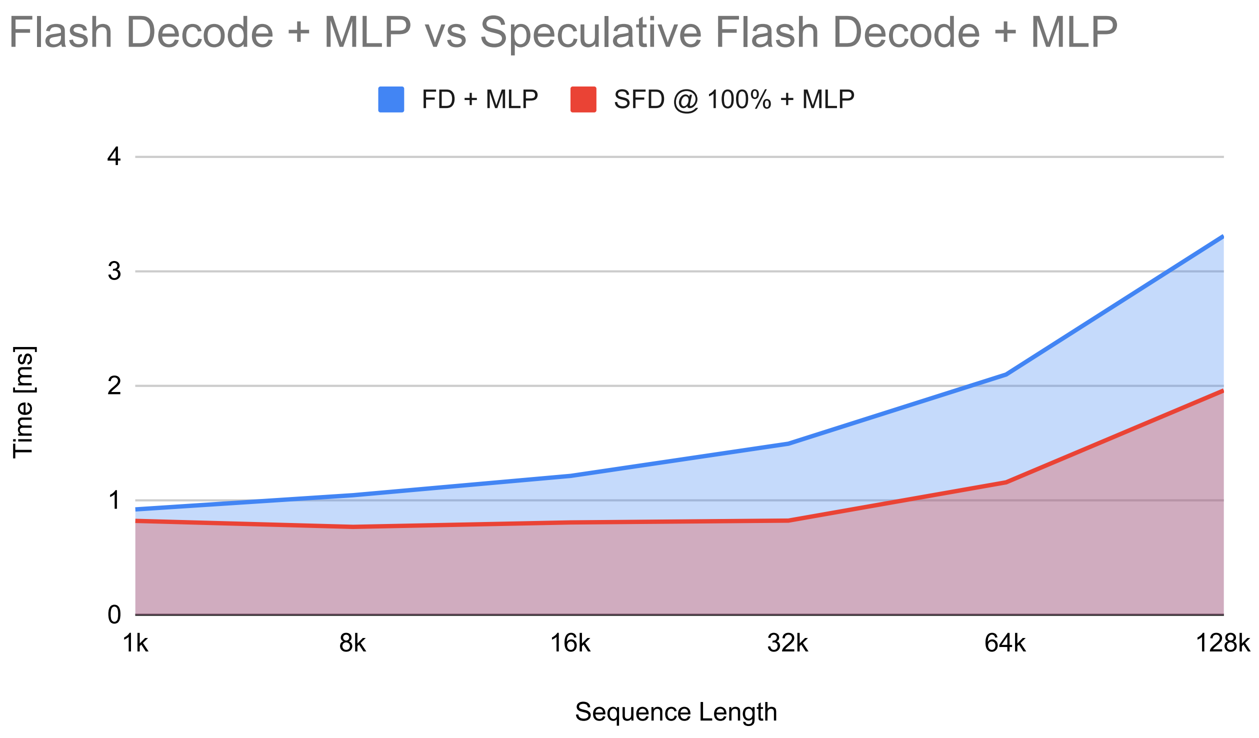


Figure 13. End-to-end performance using Llama3.1 8b shapes for Flash decode followed by MLP, compared to Speculative Flash Decode at 100% specualation followed by MLP. For FD one device is used and for SFD two devices are used.

F. Llama 3.1 8B Implementation on TT-Metalium and Performance Estimation

TT-Metalium supports native mixed precision computation at 4 different data formats on N150 chips: float32, Block Floating Point (bfloat) 16, bfloat8_b, and bfloat4_b. These data formats can be used interchangeably within the model execution to opportunistically speed up computation when full precision is not required. In Llama 3.1 8B's implementation, 4/8-bit feedforward weights and 8-bit precision KV cache has been used to reduce memory bottleneck, while operations sensitive to precision, such as rotary embedding, are performed in full or half precisions. To ensure correctness, op, module, and model level unit tests are performed to ensure output quality. Specifically, Pearson Correlation Coefficient (PCC) and top 1/5 scores has been used as common metrics in unit tests. Generally, quantization is applied to models only if the PCC score is above 0.99 and top 1/5 scores are above 0.9/0.99. The empirically verified strict thresholds ensure that the model benefits as much as it can from the mixed-precision inference without hurting the output quality. Therefore, to ensure our performance estimation of ALSpec matches and benefits the existing Llama model implementations on TT-Metalium, we use the default mixed-precision models brought up on the repository (Tenstorrent, 2024a).

We estimate the pure device performance of the Llama 3.1 8B model during decode phase using the following method:

- We measure the kernel latency (Device FW Duration) of the operations within a single decoder layer, *excluding* the flash decode op. We call this per_layer_latency.
- We also measure the kernel latency of operations outside the decode (embedding and LM-head ops). We call this fixed_latency.
- We sweep the latency of flash decode and speculative flash decode op across context length, from 1K to 128K. We denote the latency $FD(K)$ and $SFD(K)$ for the latency at context length K.
- For speculative flash decode, we record both the sender ($SFD_S(K)$) and receiver ($SFD_R(K)$) kernel latency.
- We repeat the measurement on 1, 2, 4 and 8 devices tensor parallelism.

For decode phase, the input is always the same shape, but only the KV cache gets longer as the context length increase. In the Llama 3 implementation, the only operation that changes latency with respect to context length is the flash decode operation. Therefore, with 32 layers in the Llama 3.1 8B model, we can obtain the token latency and tokens/s/user throughput (1/(token latency)) at context length K using the following formula for full tensor parallelism:

$$\text{token\_latency} = \text{fixed\_latency} + 32 \cdot (\text{per\_layer\_latency} + FD(K))$$

To estimate the token latency at a particular speculation hit rate r [5], we use the following formula:

$$\begin{aligned}\text{token\_latency} = \text{fixed\_latency} + \lfloor 32 \cdot r \rfloor \Big(SFD_R(K) + \max(SFD_S(K) - SFD_R(K), \text{per\_layer\_latency}) \Big) \\ + \lceil 32 \cdot (1 - r) \rceil \cdot \Big(\text{per\_layer\_latency} + SFD_S(K) \Big)\end{aligned}$$

In short, this formula takes the overlapped attention latency when speculation hits, and full layer latency when speculation misses. It serves as an upper bound for latency because we always round down for the number of layers with speculation hits [6].

We present the full latency breakdown for per_layer_latency and fixed_latency across 1, 2, 4 and 8 devices tensor parallel in Table 4, 5, 6, & 7. One can verify the estimation in Figure 7 based on the table and the speculative flash decode latency in Figure 12. The latency breakdown also shows how diminishing returns happen with tensor parallelism, while compute operations such as matmul get lower latency when parallelized to more devices, CCL operation latency such as all-gather increases, making the overall latency reduction minimal.

[5]Note that in our measurement, we take r as a multiple of $1/32$ to ensure that full layers are doing speculation.

[6]This formula doesn't account for the last attention speculation, which should overlap with the fixed latency (LM Head). We are aware of this limitation and we use the formula for its simplicity. In the amortized case of running multiple token generation, this approximation holds as the final attention can overlap with the pre-attention ops of the next token.

Table 4. Llama 3.1 8B latency break down on 1 x N150 chip

Per Layer

OP CODE	DEVICE FW Duration [ns]
LayerNorm	12454
Matmul	109174
ShardedToInterleaved	4299
NLPCreateHeadsDecode	21048
RotaryEmbeddingLlama	4828
RotaryEmbeddingLlama	4827
PagedUpdateCache	13166
PagedUpdateCache	13049
ScaledDotProductAttentionDecode	69853
InterleavedToSharded	1701
NLPConcatHeadsDecode	3055
Matmul	76389
Binary	1823
LayerNorm	12487
Matmul	141261
Matmul	141772
Binary	30204
Matmul	252556
Binary	2654
Total / Layer without flash decode	846747

Embedding & Lm Head

OP CODE	DEVICE FW Duration [ns]
Embeddings	5745
Embeddings	5639
Transpose	13808
Transpose	13512
Slice	2091
Slice	2089
InterleavedToSharded	1694
InterleavedToSharded	1759
Reshard	3386
LayerNorm	13243
Matmul	546986
ShardedToInterleaved	9003
Matmul	545955
ShardedToInterleaved	8833
Matmul	545707
ShardedToInterleaved	8944
Matmul	547068
ShardedToInterleaved	9055
Concat	29186
Total LM Head	2313703

Table 5. Llama 3.1 8B latency break down on Tensor Parallel 2 x N150 chips.

Per Layer Context Length 1K

OP CODE	DEVICE FW Duration [ns]
AllGather	19567
LayerNorm	12471
Matmul	56690
ShardedToInterleaved	2541
NLPCreateHeadsDecode	10972
RotaryEmbeddingLlama	4834
RotaryEmbeddingLlama	4820
PagedUpdateCache	8447
PagedUpdateCache	8383
ScaledDotProductAttentionDecode	64309
InterleavedToSharded	1833
NLPConcatHeadsDecode	2320
Matmul	41730
ReduceScatter	21842
Binary	1621
AllGather	20887
LayerNorm	12486
Matmul	76942
Matmul	77430
Binary	16543
Matmul	128695
ReduceScatter	19108
Reshard	3233
Binary	1620
Total / Layer without flash decode	555015

Embedding & Lm Head

OP CODE	DEVICE FW Duration [ns]
Embeddings	5549
Embeddings	5767
Transpose	13570
Transpose	13590
Slice	2094
Slice	2222
InterleavedToSharded	1748
InterleavedToSharded	1701
AllGather	20607
LayerNorm	12931
Matmul	546309
ShardedToInterleaved	8779
Matmul	546729
ShardedToInterleaved	8992
Concat	17048
Total LM Head	1207636

Table 6. Llama 3.1 8B latency breakdown on Tensor Parallel 4 x N150 chips.

Per Layer

OP CODE	DEVICE FW Duration [ns]
AllGather	22378
LayerNorm	12485
Matmul	39022
ShardedToInterleaved	2057
NLPCreateHeadsDecode	6269
RotaryEmbeddingLlama	4837
RotaryEmbeddingLlama	4834
PagedUpdateCache	6260
PagedUpdateCache	6342
ScaledDotProductAttentionDecode	56024
InterleavedToSharded	1694
NLPConcatHeadsDecode	1955
Matmul	24505
ReduceScatter	24753
Binary	1517
AllGather	22055
LayerNorm	13423
Matmul	39711
Matmul	39791
Binary	15800
Matmul	68424
ReduceScatter	24764
Reshard	3212
Binary	1496
Total / Layer without flash decode	387584

Embedding & Lm Head

OP CODE	DEVICE FW Duration [ns]
Embeddings	5622
Embeddings	5767
Transpose	13328
Transpose	13585
Slice	2088
Slice	2193
InterleavedToSharded	1795
InterleavedToSharded	1703
AllGather	24259
LayerNorm	12906
Matmul	546508
ShardedToInterleaved	8775
Total LM Head	638529

Table 7. Llama 3.1 8B latency breakdown Tensor Parallel on 8 x N150 chip.

Per Layer

OP CODE	DEVICE FW Duration [ns]
AllGather	34069
LayerNorm	12507
Matmul	35745
ShardedToInterleaved	1620
NLPCreateHeadsDecode	3976
RotaryEmbeddingLlama	4839
RotaryEmbeddingLlama	4837
PagedUpdateCache	5007
PagedUpdateCache	5078
ScaledDotProductAttentionDecode	50096
InterleavedToSharded	1708
NLPConcatHeadsDecode	1774
Reshard	2274
AllGatherMatmul	88825
Reshard	2847
Binary	1464
AllGather	54386
LayerNorm	15771
Matmul	22705
Matmul	22899
Binary	15804
Matmul	38486
ReduceScatter	59572
Reshard	2817
Binary	1441
Total / Layer without flash decode	440451

Embedding & Lm Head

OP CODE	DEVICE FW Duration [ns]
Embeddings	5641
Embeddings	5607
Transpose	13717
Transpose	13680
Slice	2095
Slice	2092
InterleavedToSharded	1707
InterleavedToSharded	1726
AllGather	43463
LayerNorm	12909
Matmul	276107
ShardedToInterleaved	5716
Total LM Head	384460

G. Section 2 Needle in a Haystack Experiment Details

The tiny needle in a haystack experiment is served as a motivational experiment rather than a rigorous benchmark. It is conducted on a custom dataset created with the following method:

- First, we create a large context using the text of the paper “Compute Substrate for Software 2.0” (Vasiljevic et al., 2021) and the first 4 chapters of the book “A Tale of the Two Cities” by Charles Dickens. This makes the total context length around 16K.
- Then, we generate 7 random MD5 hashes and insert at random places in the text.
- To ensure the result is not biased towards hash strings, we make up 3 random facts and insert at random places in the text.

We then prompt the model with the following questions after the context:

- Start with text “Answer the following questions if you can. If you don’t know the answer, answer I don’t know.”
- Then, we put questions about fact 1, 2, and 3.
- In the end, we ask the model to repeat hashes 1-7.