[Proposal] Speculative Speculative Decoding: Hiding Draft Latency Through Asynchronous Speculation on Multi-GPU Systems

Marcus Alenius **William Chien**
Carnegie Mellon University
{malenius, wjchien}@andrew.cmu.edu

URL: https://alenius.io/15418-project

1 Summary

We plan to implement speculative speculative decoding (SSD), a technique for parallelizing the sequential token generation of large language models by running the draft and verification phases of speculative decoding concurrently on separate GPUs. While the target model verifies one round of speculation, the draft model predicts the outcomes and precomputes speculations for the next. We will build this in C++/CUDA on multi-GPU V100 nodes using cuBLAS and NCCL, and evaluate where SSD outperforms standard speculative decoding and where asynchronous coordination overhead outweighs the advantage.

2 Background

2.1 Autoregressive Decoding

Large language models generate text autoregressively, meaning that each token is predicted based on all previous tokens. This creates a sequential dependency: the model cannot start generating the next token until the current one is complete. Each inference step (generating one token) involves multiple vector-matrix multiply, barely utilizing the GPU's compute capacity. Yet, the dependence between steps means the GPU cannot move on to the next step until the current one finishes. The result is a powerful GPU that is underutilized during language model inference.

2.2 Speculative Decoding

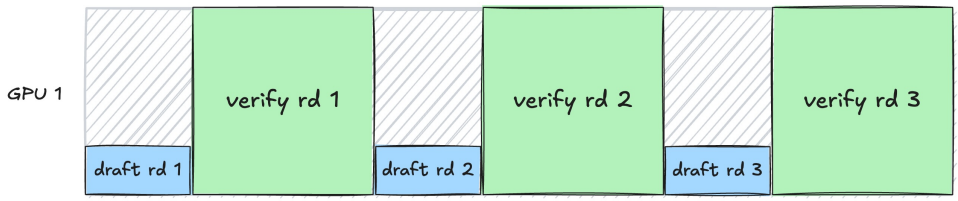
There has been work on parallelizing the generation process (often referred to as the decoding process). One such approach is **speculative decoding (SD)** (Leviathan et al., 2023; Chen et al., 2023). It does so by using a small and fast draft model to generate K candidate tokens autoregressively, then verifying all K tokens in parallel using a single batched forward pass through the large target model. The target model accepts the longest prefix of candidates that matches what it would have generated. When the draft model's acceptance rate is high, multiple tokens are generated per target forward pass, parallelizing the decoding process and improving latency. Crucially, this process is lossless—the output distribution is identical to what the target model would have produced on its own.

However, SD itself has a sequential dependency: the draft model must wait for verification to finish before beginning the next round of speculation, and the target model must wait for the draft model to finish before it can verify. This means that during drafting, the target model's GPUs sit idle, and during verification, the draft GPU sits idle. Draft and verify alternate, but never overlap.

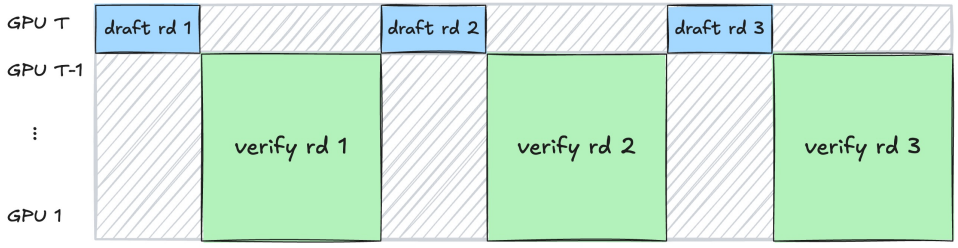
2.3 Speculative Speculative Decoding

Recently, **speculative speculative decoding (SSD)** (Kumar et al., 2026) has been proposed to break this sequential dependency. While the target model is verifying round R's speculation, the draft model (on separate hardware) predicts what the verification outcome will be and pre-computes speculations for round $R + 1$ for multiple possible outcomes in parallel. These pre-computed speculations are stored in a speculation cache. When the actual verification outcome arrives, the draft checks whether it predicted correctly (a cache hit). If so, it returns the pre-computed speculation immediately and the draft overhead is completely hidden. If not (a cache miss), it falls back to generating a speculation just-in-time before the next verification can proceed. Like SD, SSD is lossless: the speculation is always verified by the target model, so the output distribution is unchanged.

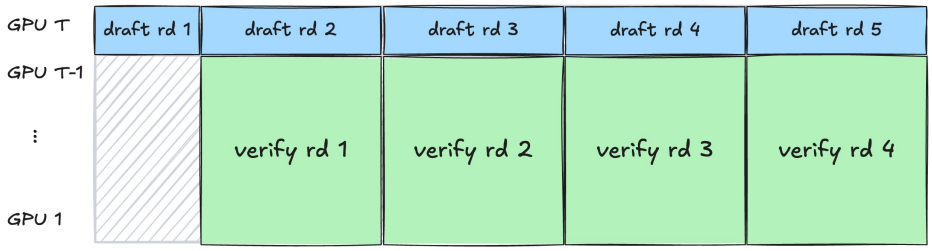
Figure 1 illustrates how SSD overlaps drafting and verification, comparing it to SD.



(a) *SD on one GPU.* The draft model drafts K tokens sequentially, then the target model verifies the K tokens in parallel. Much of the GPU sits idle during the draft phase.



(b) *SD on T GPUs.* The draft model drafts K tokens sequentially on GPU T, then the target model verifies the K tokens in parallel on GPUs 1 through $T-1$. GPUs 1 through $T-1$ sit idle during drafting, and GPU T sits idle during verification.



(c) *SSD on T GPUs.* The draft model drafts K tokens for multiple verification outcomes while the target model verifies the K tokens from the previous round. All GPUs are busy.

Figure 1: Comparison of speculative decoding (SD) and speculative speculative decoding (SSD). SSD overlaps drafting and verification across GPUs, improving utilization.

3 The Challenge

3.1 Parallelizing Autoregressive Decoding

The fundamental challenge is the sequential dependency between tokens: token $N+1$ depends on the output of token N, so we cannot generate multiple tokens in parallel. Within a single token's inference step, there is parallelism in the matrix multiplications, but each step has very low arithmetic intensity. The decode step performs vector-matrix multiplies (one token times the weight matrix), which is memory-bandwidth-bound rather than compute-bound. This means that simply adding more compute (e.g., more GPU cores) does not help—the bottleneck is how fast we can read weights from HBM.

Tensor parallelism (sharding weight matrices across GPUs) can help by distributing the memory bandwidth demand, but it introduces an all-reduce communication after every layer. For the tiny vector-matrix multiplies of single-token decode, the all-reduce latency can easily exceed the per-GPU compute time, limiting how far tensor parallelism can scale. The KV-cache, which stores key/value vectors from all previous tokens, grows linearly with sequence length and must be managed across GPUs, adding memory pressure that interacts with batch size.

Speculative decoding addresses the sequential dependency by converting multiple serial decode steps into one parallel verification step. But as described above, it introduces its own idle time. SSD aims to eliminate this idle time—but doing so introduces a new set of challenges.

3.2 Implementation Challenges

There are several workload and constraint challenges that make this difficult.

Asynchronous coordination under a hard deadline. The draft model must complete all of its pre-speculation work before the target model fin-

ishes its single verification forward pass. Whether this is feasible depends on the draft-to-target model size ratio, the GPU allocation, and the fan-out budget (how many possible verification outcomes it drafts for). If the draft cannot finish in time, the cache is incomplete and hit rates suffer. This creates a scheduling problem with no static solution—the right fan-out budget depends on runtime conditions.

Communication is latency-sensitive. The draft and target must synchronize once per round: the target sends the verification outcome, and the draft returns the pre-computed speculation. This communication competes with the tensor-parallel all-reduces happening within the target's forward pass on the same NVLink interconnect. Any delay directly adds to per-round latency.

Cache hit rate degrades with batch size. At batch size 1, a cache miss only stalls one sequence. At batch size b, the entire batch must wait for the fallback speculator whenever any sequence misses, and the probability of at least one miss is $1-p_{\text{hit}}^b$. At batch size 16 with a per-sequence hit rate of 85%, the probability of a clean round with no misses is only about 7%. This means that the fallback strategy dominates end-to-end performance at larger batch sizes, qualitatively changing the system's behavior and forcing a different set of tradeoffs.

We hope to learn where the boundary lies between configurations where SSD meaningfully outperforms SD and where the overhead of asynchronous coordination and cache misses outweighs the advantage.

4 Resources

Hardware. We will primarily use the GHC cluster machines equipped with an NVIDIA GeForce RTX 2080 GPU for development. For evaluation and multi-GPU development, we will use PSC GPU nodes equipped with 8 NVIDIA Tesla V100-32GB GPUs connected via NVLink. SSD requires the draft and target models to reside on separate GPUs, so at minimum we need $T + 1$ GPUs per run (e.g., 5 GPUs for 4-way tensor parallelism plus one draft GPU). We will verify the NVLink topology early in Week 1, as NCCL send/recv latency between the draft and target GPUs directly affects SSD's per-round overhead.

Software. We will build our transformer-shaped compute graph on cuBLAS GEMM kernels. For tensor-parallel and draft-target communication, we will use NCCL.

Starter code. We are starting from scratch. We will build the entire pipeline (compute graph, tensor parallelism, SD, and SSD) ourselves using the cuBLAS and NCCL APIs.

References. Our primary references are the two papers that introduced speculative decoding: Leviathan et al. (2023) and Chen et al. (2023), as well as Kumar et al. (2026), which recently proposed speculative speculative decoding.

5 Goals and Deliverables

5.1 Plan to Achieve

Transformer-shaped compute graph implementation. As a transformer layer is dominated by a sequence of matrix multiplications, we don't need to implement the full transformer architecture to explore parallelizing the decoding process. Instead, we can create a simplified "transformer-shaped" compute graph that captures the key characteristics of the decoding step: a sequence of matrix multiplies with the correct dimensions and dependencies. For a model with hidden dimension H, a single token, and a current sequence length S, each layer does roughly:

1. QKV projection: $(1 \times H) \times (H \times 3H)$.
2. Attention: QK^T of shape $(1 \times H) \times (H \times S)$, then AV of shape $(1 \times S) \times (S \times H)$.
3. Attention output projection: $(1 \times H) \times (H \times H)$.
4. Feed-forward up projection: $(1 \times H) \times (H \times 4H)$.
5. Feed-forward down projection: $(1 \times 4H) \times (4H \times H)$.

We skip LayerNorm, softmax, activation functions, and residual additions. These are element-wise operations that contribute negligibly to the overall runtime. For the KV cache, we pre-allocate buffers of size $(S_{max} \times H)$ at startup with random values and run the attention GEMMs against them at a configurable sequence length.

We stack L of these layers, call cuBLAS for each GEMM in sequence, and that's one forward pass. We allocate the weight matrices once at startup with random floats.

For tensor parallelism across T GPUs we shard the weight matrices column-wise (or row-wise depending on the GEMM) so each GPU does a $(1 \times H) \times (H/T \times H)$ multiply. We also insert an NCCL all-reduce after each layer to recombine the partial results.

AR baseline. For the autoregressive baseline, we simply run the compute graph as described above N times in a loop.

SD baseline. For the speculative decoding baseline we do the following N times:

1. Draft phase: Run K forward passes through the draft model sequentially (same as AR, but with the small model dimensions). This produces K dummy "tokens."

2. Verify phase: Run one forward pass through the target model, but now the batch dimension is K instead of 1. So the GEMMs become $(K \times H) \times (H \times H)$ instead of $(1 \times H) \times (H \times H)$. This is a bigger matrix multiply that better utilizes the GPU. With tensor parallelism, each GPU does $(K \times H) \times (H/T \times H)$ followed by an all-reduce.

3. Acceptance: Draw from $\text{Bernoulli}(\alpha)$ for each token, find the first rejection at position k. We "generated" $k + 1$ tokens this round.

SSD implementation. Run target model and draft model on separate hardware (the target model stays on GPUs 0 through $T - 1$ with tensor parallelism. The draft model moves to its own dedicated GPU—GPU T). In SD they could share hardware but in SSD they must be separate because they run simultaneously. Figure 2 show the pseudocode.

Throughput and latency measurements. Run the AR, SD, and SSD pipelines for multiple configurations and record timing. At minimum, we should sweep over:

- Fan-out F (say $F = 1, 2, 4, 8, 16$)
- Acceptance rate α (say $\alpha = 0.5, 0.7, 0.9$)
- Speculation length K
- Cache hit rate p_{hit} (for SSD)

```
// On target GPUs:
while not done:
    launch verification forward pass (K x H GEMMs
         + all-reduces)
    wait for verification to complete
    determine outcome (synthetic acceptance with
        alpha)
    NCCL send outcome to draft GPU (acceptance
        length and bonus token ID)
    NCCL recv next speculation from draft GPU

// On draft GPU (running concurrently):
while not done:
    launch pre-speculation (F x (K+1) x K GEMMs
        to fill cache)
    NCCL recv outcome from target
    if cache hit (with probability p_hit):
        NCCL send cached speculation to target
    else:
        run K more GEMMs, then send
```

Figure 2: Concurrent execution of draft and target GPUs in SSD.

Verification window characterization. Report target verification time and draft pre-speculation time ratio: "At draft size X and target size Y with T-way tensor parallelism, the target verification takes 15 ms and each draft forward pass takes 0.3 ms, so we can fit 50 draft forward passes in the window, which allows a maximum fan-out budget of $50/K$."

Roofline-style analysis. Use microbenchmark measurements (NCCL latency/bandwidth, cuBLAS throughput) to explain the bottleneck at each configuration point.

If the work goes slowly. At minimum, we will deliver a working SD implementation with tensor parallelism across multiple GPUs, along with a thorough characterization of the verification window (how many draft forward passes fit within one target verification pass for various model configurations). This analysis would still yield useful insights into the feasibility of SSD on V100 hardware, even if the full asynchronous pipeline is not complete.

Poster session. We plan to present speedup graphs comparing AR, SD, and SSD across configurations (varying α, K, F, and p_{hit}), along with a roofline-style breakdown showing where each configuration is bottlenecked.

5.2 Hope to Achieve

Batch size scaling analysis. Instead of running one sequence through the pipeline, we run b se-

quences. The target's forward pass becomes a larger batched GEMM (batch dimension is $b \times K$ instead of K). The draft builds a cache per sequence, so total pre-speculation work scales by b. On cache lookup, we check each sequence independently, but the system waits for the slowest sequence—if any one has a cache miss, the whole batch stalls for the fallback.

Heterogeneity experiments. Clock-limit the draft GPU to simulate a weaker accelerator, measuring how the draft-to-target speed ratio affects the achievable fan-out and optimal speculation length.

Formal analytical model. Develop a formal analytical model that predicts SSD throughput from hardware parameters and workload parameters, validated against our measurements.

5.3 Performance Targets

We derive quantitative targets from the theoretical speedup formulas in Kumar et al. (2026). Full derivations are in Appendix A.

SD over AR. At acceptance rate $\alpha = 0.9$ and speculation length $K = 5$, the expected tokens per round is $E_{\text{SD}} = (1 - \alpha^{K+1})/(1 - \alpha) \approx 4.7$, and the SD speedup is $E_{\text{SD}}/(1 + T_{\text{SD}})$, where T_{SD} is the draft-to-verification time ratio. We expect T_{SD} to be small (roughly 0.1–0.3) given a draft model $\sim$100$\times$ smaller than the target, yielding a **3–4$\times$ speedup over AR**. At $\alpha = 0.7$ and $K = 3$, we expect roughly **2$\times$ over AR**.

SSD over SD. On a cache hit, draft latency is fully hidden, reducing per-round time from $(1 + T_{\text{SD}})$ to $\max(1, T_p) \approx 1$. The maximum SSD improvement over SD is therefore a factor of $(1 + T_{\text{SD}})$. At $p_{\text{hit}} = 0.85$ and $\alpha = 0.9$, we expect SSD to be **15–25% faster than SD at batch size 1**, consistent with the $\sim$30% improvements reported on H100s with real models. For batch size scaling: the probability of a clean round (all sequences hit) is p_{hit}^b, so at $p_{\text{hit}} = 0.85$ and $b = 8$, only $\sim$27% of rounds avoid a fallback. We expect **SSD's advantage over SD to diminish to near zero by batch size 8–16**.

We will calibrate these targets further once we measure T_{SD} for our model dimensions on the V100 GPUs.

6 Platform Choice

We will build our project on top of NVIDIA GPUs. LLM inference is typically run on GPUs, as transformer language models benefit from their massive parallelism and high memory bandwidth, both of which are critical for inference.

GPUs are especially well-suited to the parallelization strategies we explore. During autoregressive decoding, each step performs a vector-matrix multiply per layer—an operation that is memory-bandwidth-bound and barely utilizes the GPU's compute capacity. This underutilization is exactly what speculative decoding exploits by batching draft tokens into larger, more compute-heavy verification passes, and what SSD exploits further by running the draft model on otherwise-idle GPUs concurrently with verification.

We will use C++, CUDA, cuBLAS, and NCCL to implement our project. This gives us precise control over kernel launches, inter-GPU communication, and critically asynchronous overlap of draft and target execution across GPUs, which is central to SSD. Using C++ over Python and PyTorch is reasonable here because we are not building a full inference engine or implementing a complete model, but instead creating a transformer-shaped compute graph out of cuBLAS GEMM calls and NCCL collectives.

7 Schedule

Week 1 (3/29–4/4)

- Implement single-GPU cuBLAS GEMM sequence for one transformer layer with correct dimensions for both draft and target.
- Set up PSC environment, verify NVLink topology and run NCCL microbenchmarks (all-reduce, send/recv at various message sizes).
- Stack L layers into a full forward pass, measure single-GPU AR decode throughput.
- Measure T_{SD} for target model dimensions on V100 to calibrate performance targets.
- Compare microbenchmark results to V100 roofline and document baseline numbers.

Week 2 (4/5–4/11)

- Implement tensor-parallel target model (column/row-wise weight sharding, NCCL all-reduce after each layer).
- Implement single-GPU SD pipeline (draft loop, batched verification, synthetic acceptance).
- Integrate tensor-parallel target with SD pipeline for multi-GPU SD. Validate SD speedups against performance targets from

Appendix A.

Week 3 (4/12–4/18)—Milestone Report Due 4/14

- Implement target GPU loop for SSD (launch verification, determine outcome, NCCL send/recv with draft GPU).
- Implement draft GPU loop (pre-speculation with fan-out, cache storage and lookup).
- Write milestone report.
- Implement communication protocol between draft and target (outcome transmission, speculation return, cache hit/miss branching).
- Integration testing: verify that SSD matches SD output under $p_{\text{hit}} = 0$ (pure fallback) as a correctness check.

Week 4 (4/19–4/25)

- Conduct throughput and latency sweeps over F, α, K, and p_{hit}.
- Perform verification window characterization (draft-to-target time ratio, maximum fan-out budget).
- Conduct roofline-style analysis using microbenchmark data.
- Perform batch size scaling experiments (hope to achieve).
- Debug and re-run any configurations with anomalous results.

Week 5 (4/26–5/2)—Final Report Due 4/30, Poster Session 5/1

- Generate final figures.
- Write final report.
- Prepare poster presentation.

References

Charlie Chen, Sebastian Borgeaud, Geoffrey Irving, Jean-Baptiste Lespiau, Laurent Sifre, and John Jumper. 2023. Accelerating large language model decoding with speculative sampling. *Preprint*, arXiv:2302.01318.

Tanishq Kumar, Tri Dao, and Avner May. 2026. Speculative speculative decoding. *Preprint*, arXiv:2603.03251.

Yaniv Leviathan, Matan Kalman, and Yossi Matias. 2023. Fast inference from transformers via speculative decoding. *Preprint*, arXiv:2211.17192.

A Speedup Derivations

A.1 Expected Tokens per SD Round

In speculative decoding, the draft proposes K tokens and the target verifies them in order. Each draft token is accepted independently with probability α. The target accepts the longest prefix of consecutive accepted tokens, then always produces one bonus token. The number of generated tokens is at least i if and only if the first $i-1$ draft tokens were all accepted. So,

$$E_{\text{SD}} = E[\text{tokens}] = \sum_{i=1}^{K+1} P(\text{generate} \geq i \text{ tokens})$$
$$= \sum_{i=1}^{K+1} \alpha^{i-1} = \sum_{j=0}^{K} \alpha^{j} = \frac{1-\alpha^{K+1}}{1-\alpha}.$$

A.2 SD Speedup over AR

Autoregressive decoding produces 1 token per target forward pass of duration T_{target}—so AR generates tokens at a rate of $1/T_{\text{target}}$ tokens per unit time. Each SD round runs K sequential draft passes (total time T_{draft}) followed by one target verification (T_{target}), producing E_{SD} tokens in expectation—so SD generates tokens at a rate of $E_{\text{SD}}/(T_{\text{draft}} + T_{\text{target}})$ tokens per unit time. The speedup is the ratio of tokens-per-unit-time:

$$\text{speedup}_{\text{AR}\to\text{SD}} = \frac{E_{\text{SD}}/(T_{\text{draft}} + T_{\text{target}})}{1/T_{\text{target}}}$$
$$= \frac{E_{\text{SD}}}{T_{\text{draft}}/T_{\text{target}} + 1} = \frac{E_{\text{SD}}}{1 + T_{\text{SD}}},$$

where $T_{\text{SD}} = T_{\text{draft}}/T_{\text{target}}$.

A.3 SSD Improvement over SD

In SSD, the draft runs concurrently with verification on separate hardware. On a cache hit (probability p_{hit}), the draft finishes within the verification window ($T_p < 1$), so the round takes $\max(1, T_p) = 1$ unit of time, where T_p is the time taken by the primary speculator. On a cache miss, the draft must speculate just-in-time after verification completes, costing $1 + T_b$ in total, where T_b is the time taken by the backup speculator. The expected per-round time is:

$$p_{\text{hit}} \cdot 1 + (1 - p_{\text{hit}}) \cdot (1 + T_b).$$

The speedup of SSD over SD, assuming similar expected tokens per round, is:

$$\text{speedup}_{\text{SD}\to\text{SSD}} \approx \frac{1 + T_{\text{SD}}}{p_{\text{hit}} + (1 - p_{\text{hit}})(1 + T_b)}.$$

The upper bound of $(1 + T_{\text{SD}})$ is achieved when $p_{\text{hit}} = 1$.