Optimized CUDA Kernel for Rejection Sampling in Speculative Decoding

Parallel Programming Final Project Report

Hong-Yan Huang (R14922156)
Bo-Ru Chen (R14922158)
Guan-Cheng Chang (R14922172)

National Taiwan University

December 2025

Abstract

Speculative decoding is a promising technique for accelerating large language model (LLM) inference by using a smaller draft model to propose tokens that are then verified by a larger target model. A critical component of this approach is rejection sampling, which determines whether to accept or reject each proposed token. In this work, we present an optimized CUDA kernel for rejection sampling that achieves up to **9.39× speedup** over the baseline Python implementation and **112.98× speedup** over the naive CUDA implementation for the sampling operation itself. Our key optimization leverages warp-level parallel reduction using CUDA shuffle instructions to accelerate the argmax operation during token recovery. We validate our implementation on two model families (Llama-3.2 and Qwen2.5) across various batch sizes, demonstrating consistent performance improvements while maintaining algorithmic correctness with identical acceptance rates across all implementations.
GitHub Repository: https://github.com/Teacup1117/pp25_final_project

Contents

1 Introduction

1.1 Background and Motivation

Large Language Models (LLMs) have achieved remarkable success across various natural language processing tasks. However, their autoregressive generation process is inherently slow due to the sequential nature of token generation—each new token requires a full forward pass through the model. This limitation becomes particularly problematic for large models with billions of parameters, where inference latency can significantly impact user experience and system throughput.

Speculative decoding [1] addresses this challenge by employing a smaller, faster "draft" model to propose multiple candidate tokens, which are then verified in parallel by the larger "target" model. This approach can achieve 2–3× speedup in LLM inference while maintaining the exact output distribution of the target model.

A critical component of speculative decoding is the **rejection sampling** algorithm, which decides whether to accept or reject each proposed token based on the probability ratio between the target and draft models. When a token is rejected, a recovery sample must be drawn from a modified distribution, requiring an efficient argmax operation over the entire vocabulary.

1.2 Problem Statement

The rejection sampling operation in speculative decoding involves:

1. For each draft token, computing the acceptance probability $\min(1, p_{\text{target}}/p_{\text{draft}})$
2. If rejected, sampling from the recovery distribution $\max(0, p_{\text{target}} - p_{\text{draft}})$
3. If all tokens are accepted, appending a bonus token

The primary performance bottleneck lies in the **argmax operation during recovery**, which requires scanning the entire vocabulary (often 32K–152K tokens) to find the maximum value. A naive implementation results in $O(\text{vocab\_size})$ sequential operations per batch element, which becomes prohibitively expensive for modern LLMs with large vocabularies.

1.3 Contributions

In this work, we make the following contributions:

1. **Optimized CUDA Kernel (V2)**: We develop a warp-level parallel reduction kernel that reduces the argmax complexity from $O(\text{vocab\_size})$ to $O(\text{vocab\_size}/256)$ through efficient use of CUDA shuffle instructions.
2. **Comprehensive Benchmarking**: We evaluate our implementation across two model families (Llama-3.2 and Qwen2.5), multiple batch sizes (1–8), and different vocabulary sizes (32K–152K).
3. **End-to-End Integration**: We integrate our optimized kernel with the HuggingFace Transformers library and demonstrate performance in realistic inference scenarios.
4. **Open-Source Release**: We release our complete implementation, including benchmarking scripts and documentation, at https://github.com/Teacup1117/pp25_final_project.

2 Related Work

2.1 Speculative Decoding

Speculative decoding was introduced by Leviathan et al. [1] as a method to accelerate autoregressive generation without modifying the output distribution. The key insight is that a smaller draft model can often predict the same tokens as the target model, and when it fails, the target model's computation can be reused to recover.

Several extensions have been proposed to improve speculative decoding:

- **Medusa** [2]: Uses multiple prediction heads attached to the target model instead of a separate draft model.
- **EAGLE** [3]: Employs an autoregression head that predicts future features rather than tokens directly.
- **Lookahead Decoding** [4]: Uses n-grams from the prompt to guide speculation.

2.2 CUDA Kernel Optimization

Efficient CUDA kernel design is crucial for GPU-accelerated applications. Key optimization techniques include:

- **Parallel Reduction**: Converting sequential operations into parallel tree-based reductions to maximize GPU utilization [5].
- **Warp-Level Primitives**: Using shuffle instructions (`__shfl_down_sync`) for efficient intra-warp communication without shared memory [6].
- **Memory Coalescing**: Organizing memory accesses to maximize bandwidth utilization.

2.3 LLM Inference Systems

Modern LLM inference systems employ various optimizations:

- **vLLM** [7]: Introduces PagedAttention for efficient KV-cache management.
- **TensorRT-LLM**: NVIDIA's optimized inference engine with kernel fusion and quantization support.
- **Flash Attention** [8]: Memory-efficient attention computation through tiling and recomputation.

3 Method

3.1 Rejection Sampling Algorithm

The rejection sampling algorithm for speculative decoding follows Leviathan et al. [1]. Given draft probabilities p_d, target probabilities p_t, and draft token x, the algorithm proceeds as:

Algorithm 1 Rejection Sampling for Speculative Decoding

Require: Draft probs p_d, Target probs p_t, Draft tokens x, Uniform samples u
Ensure: Output tokens, acceptance count

for each token position $i = 1, \ldots, K$ **do**
  $r \leftarrow p_t[x_i]/p_d[x_i]$ ▷ Acceptance ratio
  if $u_i < \min(1, r)$ **then**
    Accept token x_i
  else
    Compute recovery distribution: $q \leftarrow \max(0, p_t - p_d)$
    Normalize: $q \leftarrow q/\sum q$
    Sample recovery token from q (using argmax for determinism)
    break ▷ Early exit
  end if
end for
if all K tokens accepted **then**
  Append bonus token sampled from p_t
end if

3.2 Baseline Implementation

Our baseline implementation uses pure Python with PyTorch operations:

Listing 1: Baseline rejection sampling (simplified)

```
def baseline_rejection_sample(draft_probs, target_probs,
                              draft_tokens, uniform_samples):
    for batch_idx in range(batch_size):
        for token_idx in range(num_spec_tokens):
            ratio = target_probs[token_idx, draft_tokens[token_idx]] /
                \
                    draft_probs[token_idx, draft_tokens[token_idx]]
            if uniform_samples[token_idx] < min(1.0, ratio):
                # Accept
                output[batch_idx, token_idx] = draft_tokens[token_idx]
            else:
                # Reject and recover
                recovery = torch.clamp(target_probs - draft_probs, min
                    =0)
                output[batch_idx, token_idx] = torch.argmax(recovery)
                break
    return output
```

This implementation has $O(B \times K)$ kernel launches and $O(\text{vocab\_size})$ complexity per recovery operation, where B is batch size and K is the number of speculative tokens.

3.3 V1 CUDA Kernel: Fused Implementation

The V1 kernel fuses all operations into a single CUDA kernel launch, eliminating Python overhead:

- Each CUDA thread processes one batch element
- Sequential processing of speculative tokens within each thread
- **Bottleneck**: Sequential argmax over vocabulary for recovery ($O(\text{vocab\_size})$)

3.4 V2 CUDA Kernel: Warp-Level Parallel Reduction

Our optimized V2 kernel addresses the argmax bottleneck using warp-level parallel reduction:

3.4.1 Architecture Overview

- Each CUDA block processes one batch element
- 256 threads per block (8 warps)
- **Phase 1**: Sequential accept/reject decisions
- **Phase 2**: Parallel argmax using warp reduction

3.4.2 Warp-Level Reduction

The key optimization is the `warpReduceArgMax` function that performs parallel reduction within a warp using shuffle instructions:

Listing 2: Warp-level argmax reduction

```
__device__ __forceinline__ void warpReduceArgMax(
    float& max_val, int& max_idx) {

    for (int offset = 16; offset > 0; offset /= 2) {
        float other_val = __shfl_down_sync(0xFFFFFFFF, max_val, offset)
            ;
        int other_idx = __shfl_down_sync(0xFFFFFFFF, max_idx, offset);

        if (other_val > max_val) {
            max_val = other_val;
            max_idx = other_idx;
        }
    }
}
```

3.4.3 Block-Level Reduction

After warp-level reduction, results are combined across warps using shared memory:

Listing 3: Block-level reduction using shared memory

```
__shared__ float warp_max_vals[8];  // 256 threads / 32 = 8 warps
__shared__ int warp_max_idxs[8];

// Each warp's lane 0 writes to shared memory
if (lane_id == 0) {
    warp_max_vals[warp_id] = local_max_val;
    warp_max_idxs[warp_id] = local_max_idx;
}
__syncthreads();

// First warp performs final reduction
if (warp_id == 0 && lane_id < 8) {
    float val = warp_max_vals[lane_id];
    int idx = warp_max_idxs[lane_id];
    warpReduceArgMax(val, idx);
    if (lane_id == 0) {
        global_max_idx = idx;
```

```
    }
}
```

3.4.4 Complexity Analysis

Table 1: Complexity comparison of different implementations

Operation	**V1**	**V2**
Accept/Reject	$O(1)$	$O(1)$
Argmax (Recovery)	$O(\text{vocab\_size})$	$O(\text{vocab\_size}/256)$
Total per batch	$O(K \times \text{vocab\_size})$	$O(K \times \text{vocab\_size}/256)$

4 Experiment

4.1 Experimental Setup

4.1.1 Hardware Configuration

- **GPU**: NVIDIA RTX 4090 (82 SMs, Compute Capability 8.9)
- **CUDA**: Version 12.4
- **Memory**: 24 GB GDDR6X

4.1.2 Software Configuration

- **PyTorch**: Version 2.x
- **Python**: Version 3.10
- **Precision**: FP16 (float16)

4.1.3 Model Configurations

We evaluate on two model families with different vocabulary sizes:

Table 2: Model configurations for evaluation

Configuration	**Draft Model**	**Target Model**	**Vocab Size**
Llama-3.2	1B	3B	128,256
Qwen2.5	0.5B-Instruct	1.5B-Instruct	151,936

4.1.4 Benchmark Protocol

- Tokens generated per run: 50
- Speculative tokens per step: 8
- Warmup iterations: 10
- Measurement iterations: 100
- Random seed: 42 (for reproducibility)

4.2 Experiment 1: Llama-3.2 Model Family

4.2.1 Pure Rejection Sampling Performance

Table 3 shows the pure rejection sampling latency across different batch sizes.

Table 3: Rejection sampling latency (ms) - Llama-3.2 (1B → 3B)

Batch Size	Baseline (ms)	Fused (ms)	CUDA V2 (ms)	Fused Speedup	CUDA V2 Speedup
1	11.64	22.04	5.76	0.53×	**2.02**×
2	16.02	9.40	5.59	1.70×	**2.86**×
4	32.88	9.87	5.62	3.33×	**5.85**×
8	59.08	13.76	7.06	4.29×	**8.37**×

Key Observations:

- CUDA V2 maintains stable low latency (5–7ms) across all batch sizes
- At batch size 8, CUDA V2 achieves **8.37**× speedup over baseline
- Fused PyTorch is slower than baseline at batch=1 due to kernel launch overhead
- CUDA V2 demonstrates excellent scalability (1.23× increase from batch 1 to 8)

4.2.2 Overall Throughput

Table 4 shows the end-to-end throughput including model inference.

Table 4: Overall throughput (tokens/s) - Llama-3.2

Batch Size	Baseline (tok/s)	Fused (tok/s)	CUDA V2 (tok/s)	CUDA V2 Speedup
1	27.53	28.13	28.95	1.05×
2	39.91	40.08	40.39	1.01×
4	67.76	66.30	68.54	1.01×
8	81.78	81.06	80.34	0.98×

4.2.3 Acceptance Rate Validation

Table 5: Acceptance rate consistency - Llama-3.2

Batch Size	Baseline	Fused	CUDA V2
1	71.43%	71.43%	71.43%
2	62.50%	62.50%	62.50%
4	60.27%	60.27%	60.27%
8	57.59%	57.59%	57.59%

All three implementations produce **identical acceptance rates**, validating the correctness of our CUDA kernel.

4.3 Experiment 2: Qwen2.5 Model Family

4.3.1 Pure Rejection Sampling Performance

Table 6: Rejection sampling latency (ms) - Qwen2.5 (0.5B → 1.5B)

Batch Size	Baseline (ms)	Fused (ms)	CUDA V2 (ms)	Fused Speedup	CUDA V2 Speedup
1	8.75	23.36	4.77	0.37×	**1.83**×
2	16.67	10.21	4.73	1.63×	**3.53**×
4	33.76	11.20	4.87	3.02×	**6.94**×
8	59.67	15.83	6.35	3.77×	**9.39**×

Key Observations:

- CUDA V2 achieves **9.39**× speedup at batch size 8
- Consistent low latency (4.7–6.4ms) across batch sizes
- Superior scalability: 8× batch increase results in only 1.33× latency increase

4.3.2 Overall Throughput

Table 7: Overall throughput (tokens/s) - Qwen2.5

Batch Size	Baseline (tok/s)	Fused (tok/s)	CUDA V2 (tok/s)	CUDA V2 Speedup
1	13.79	13.94	14.23	1.03×
2	20.94	21.08	20.59	0.98×
4	41.29	42.24	42.56	1.03×
8	77.31	79.95	79.45	1.03×

4.3.3 Acceptance Rate Analysis

Table 8: Acceptance rate - Qwen2.5

Batch Size	Baseline	Fused	CUDA V2
1	46.43%	46.43%	46.43%
2	50.89%	50.89%	50.89%
4	60.71%	60.71%	60.71%
8	65.40%	65.40%	65.40%

Interestingly, Qwen2.5 shows **increasing acceptance rates** with larger batch sizes (46% → 65%), opposite to the Llama-3.2 trend (71% → 58%).

4.4 Scalability Analysis

Table 9: Scalability comparison (batch 1 → batch 8)

Method	Batch=1	Batch=8	Growth Factor
Baseline	8.75 ms	59.67 ms	6.8× (Linear)
Fused PyTorch	23.36 ms	15.83 ms	0.68× (Improved!)
CUDA V2	4.77 ms	6.35 ms	**1.33× (Near-constant)**

4.5 Bottleneck Analysis

To understand why the overall throughput improvement is modest despite significant sampling speedup, we analyze the time breakdown:

Table 10: Time breakdown analysis (Qwen2.5, batch=8)

Method	Sampling Time	Total Time	Sampling %
Baseline	59.67 ms	2587 ms	2.31%
CUDA V2	6.35 ms	2517 ms	0.25%

Key Finding: Rejection sampling accounts for only 0.25%–2.31% of total inference time. The **LLM forward pass dominates** (>97% of time), explaining why even a 9.39× sampling speedup yields only 3% overall improvement.

4.6 Hardware Profiling with Nsight Compute

We profiled the CUDA V2 kernel using NVIDIA Nsight Compute:

Table 11: Nsight Compute profiling results

Metric	Value
SM Busy	3.93%
Memory Throughput	0.16%
L1 Cache Hit Rate	87.49%
Theoretical Occupancy	100% (48 warps/SM)
Achieved Occupancy	2.08% (limited by grid size)
Registers per Thread	40

The low SM utilization indicates that the kernel is not compute-bound. The high L1 cache hit rate (87.49%) shows effective memory access patterns.

5 Discussion

5.1 Summary of Results

Our optimized CUDA V2 kernel demonstrates:

1. **Significant Sampling Speedup**: Up to 9.39× over baseline for the rejection sampling operation
2. **Excellent Scalability**: Near-constant time complexity with increasing batch size

3. **Correctness Verified**: Identical acceptance rates across all implementations
4. **Limited Overall Impact**: Only 3% end-to-end improvement due to LLM forward pass dominance

5.2 Limitations

1. **LLM Forward Pass Bottleneck**: Rejection sampling is not the primary bottleneck in speculative decoding. To achieve significant end-to-end speedup, optimizations must target the model inference itself (e.g., Flash Attention, quantization).
2. **Limited GPU Utilization**: The kernel achieves only 2–4% SM utilization due to the inherently sequential nature of the accept/reject phase.
3. **Fused PyTorch Overhead**: The fused PyTorch implementation shows worse performance than baseline at batch=1 due to kernel launch overhead.

5.3 Future Work

1. **Flash Attention Integration**: Combining our optimized sampling with Flash Attention for the forward pass could yield multiplicative speedups.
2. **Model Quantization**: INT8/FP8 quantization could reduce memory bandwidth requirements and accelerate the forward pass.
3. **Increased Speculation Length**: Testing with more speculative tokens (16–32) may improve efficiency for high-acceptance-rate scenarios.
4. **Multi-GPU Scaling**: Extending the implementation to support distributed inference across multiple GPUs.

6 Conclusion

We presented an optimized CUDA kernel for rejection sampling in speculative decoding that achieves up to 9.39× speedup over the baseline implementation. Our key contribution is the use of warp-level parallel reduction with CUDA shuffle instructions to accelerate the argmax operation during token recovery.

While the sampling optimization itself is significant, our analysis reveals that the LLM forward pass remains the dominant bottleneck in speculative decoding, accounting for over 97% of total inference time. This suggests that future optimization efforts should focus on the model inference component through techniques such as Flash Attention, quantization, and KV-cache optimization.

Our implementation is fully open-source and available at https://github.com/Teacup1117/pp25_final_project, enabling reproducibility and further research in this area.

7 Work Distribution

Table 12: Work distribution among team members

Team Member	Contributions
Hong-Yan Huang (R14922156)	CUDA kernel design and implementation (V2), warp-level parallel reduction optimization, performance profiling with Nsight Compute, benchmark script development, documentation and report writing, throughput analysis
Bo-Ru Chen (R14922158)	Final project topic research, midterm proposal writing, Baseline Python implementation, PyTorch fused implementation, correctness verification tests
Guan-Cheng Chang (R14922172)	End-to-end experiment design, Llama and Qwen model experiments, throughput analysis, integration with HuggingFace Transformers, CUDA kernel design and implementation (V1)

References

[1] Y. Leviathan, M. Kalman, and Y. Matias, "Fast inference from transformers via speculative decoding," in *Proceedings of the 40th International Conference on Machine Learning (ICML)*, 2023.

[2] T. Cai et al., "Medusa: Simple LLM inference acceleration framework with multiple decoding heads," in *arXiv preprint arXiv:2401.10774*, 2024.

[3] Y. Li et al., "EAGLE: Speculative sampling requires rethinking feature uncertainty," in *arXiv preprint arXiv:2401.15077*, 2024.

[4] Y. Fu et al., "Break the sequential dependency of LLM inference using lookahead decoding," in *arXiv preprint arXiv:2402.02057*, 2024.

[5] M. Harris, "Optimizing parallel reduction in CUDA," *NVIDIA Developer Technology*, 2007.

[6] NVIDIA Corporation, "CUDA C++ Programming Guide," 2023.

[7] W. Kwon et al., "Efficient memory management for large language model serving with PagedAttention," in *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP)*, 2023.

[8] T. Dao et al., "FlashAttention: Fast and memory-efficient exact attention with IO-awareness," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

A Appendix: Reproducibility

A.1 Environment Setup

Listing 4: Environment setup commands

```
# Clone repository
git clone https://github.com/Teacup1117/pp25_final_project.git
```

```
cd pp25_final_project

# Install dependencies
pip install torch numpy matplotlib transformers

# Set CUDA path
export PATH="/usr/local/cuda-12.4/bin:$PATH"

# Build CUDA extension
cd src/cuda/csrc
python setup.py build_ext --inplace
cd ../../..
```

A.2 Running Experiments

Listing 5: Commands to reproduce experiments

```
# Verify correctness
python verify_kernel_correctness.py
python deep_verify_v2.py

# Run performance benchmark
python benchmark_kernel_versions.py --gpu 0 --iterations 100

# Generate performance report
python generate_performance_report.py
```