CTC Beam Search with CUDA

15-418: Parallel Computer Architecture and Programming, Fall 2025

Benjamin Kleyner
Computer Science Department
Carnegie Mellon University
benkleyner@cmu.edu

Julius Arolovitch
Dept. of Electrical & Computer Engineering
Carnegie Mellon University
jarolovi@andrew.cmu.edu

Maxim Yagnyatinskiy
Computer Science Department
Carnegie Mellon University
myagnyat@andrew.cmu.edu

Abstract

We present a high-performance batched Connectionist Temporal Classification (CTC) Beam Search decoder for Recurrent Neural Networks (RNNs) implemented in CUDA to accelerate sequence decoding tasks on NVIDIA GPUs. The core challenge addressed is mapping the dynamic, irregular, and memory-bound nature of the beam search algorithm onto the data parallel SIMT architecture. In CTC Beam search a subsequently generated token does not depend on a previously generated one, offering more potential axes of parallelization in contrast with autoregressive Beam search which does have this dependency. Our approach focuses on optimizing the iterative decoding loop by developing custom device-side kernels for parallel hypothesis expansion and efficient parallel top-k selection for beam pruning. To further address the load imbalance caused by varying quality of active hypotheses at each time step, we implemented dynamic beam width optimizations, exponentially decreasing the search space based on a predicted decay schedule. By maximizing on-device computation and improving load imbalances, our optimized CUDA implementation achieves a speedup of over 2X compared to established CPU-based solutions, decoding in excess of 1.44 hours of speech per second.

1 Background

1.1 Connectionist Temporal Classification (CTC)

Connectionist Temporal Classification (CTC) is a loss function and decoding framework (Graves et al. [2006]) designed for sequence-to-sequence tasks where the alignment between input and output sequences is unknown, notably Automatic Speech Recognition (ASR). In these domains, the input sequence length, T, (number of acoustic frames) is typically much larger than the output sequence length, L, (number of tokens). Furthermore, there is no ground truth for the start and end timestamps for a particular output token. CTC circumvents the need for pre-segmented or pre-aligned data by inferring the alignment implicitly. It introduces a "blank" token, ϵ, to represent silence or transitions between distinct phonemes, and trains the model by marginalizing over all valid input-output mappings.

Given an input sequence $\mathbf{X} = (X_1, \ldots, X_T)$, a neural network model (e.g., Wav2Vec2, Conformer) outputs a tensor of log-probabilities over the vocabulary $\mathcal{V} = \mathcal{L} \cup \{\epsilon\}$ at each time step. In practice, to maintain numerical stability and avoid underflow when processing long sequences, CTC systems work with log-probabilities. The input to our decoder is a tensor of log-probabilities $\mathbf{P} \in \mathbb{R}^{B \times T \times V}$, where B is the batch size, T is the number of time steps, and V is the size of the vocabulary ($|\mathcal{V}|$). Thus, $\mathbf{P}_{t,c} = \log P(c \mid X_t)$.

CTC decoding assumes conditional independence between outputs at different time steps given the input. A path $\pi = (\pi_1, \ldots, \pi_T)$ is a sequence of tokens from $\mathcal{V}$ representing a specific *time-synchronous* alignment. In probability space, the probability of a path is the product of its token probabilities; however, working with log-probabilities, this becomes a summation, which is computationally efficient for SIMT architectures:

$$\log P(\pi|\mathbf{X}) = \sum_{t=1}^{T} \mathbf{P}_{t,\pi_t} \tag{1}$$

A mapping function $\mathcal{B} : \mathcal{V}^T \to \mathcal{L}^{\leq T}$ maps these paths to *order-synchronous* output sequences by first removing repeated adjacent tokens and subsequently removing blank tokens. For instance, as illustrated in recent course materials (Raj [2025]), the path $\pi^{(1)} =$ `RRR--EE--DDD` maps to the output "RED", while $\pi^{(2)} =$ `RR-E-EED` maps to "REED". This highlights the need for the blank token to distinguish repeated symbols in the underlying token sequence.

The fundamental objective of CTC decoding is to find the most likely output sequence $\mathbf{l}^*$. Notably, this is *not* equivalent to finding the single most likely path (which is what a greedy decoder does). Instead, the decoder must sum the probabilities of *all* paths that map to $\mathbf{l}$:

$$\mathbf{l}^* = \underset{\mathbf{l} \in \mathcal{L}^{\leq T}}{\operatorname{argmax}} \sum_{\pi \in \mathcal{B}^{-1}(\mathbf{l})} P(\pi|\mathbf{X}) \tag{2}$$

Greedy decoding approximates this by selecting $\operatorname{argmax} \pi_t$ at each step, but this is often suboptimal because it doesn't aggregate log-probabilities from multiple valid alignments (e.g., distinguishing "RED" vs "REED" based on the blanks). Note that while the objective above is defined as a sum of probabilities, our implementation operates on the log-probability tensor $\mathbf{P}$. Since the exact summation over all V^T paths is intractable, beam search is required to approximate the top candidates efficiently.

1.2 Beam Search Decoding

Beam search is a heuristic graph search algorithm widely used in deep learning for sequence generation. Unlike exhaustive breadth-first search, which considers all possible expansions, or greedy decoding, which only considers the single best option at each step, beam search maintains a fixed number, W ("beam width"), of most promising partial hypotheses (or "beams") at each step.

At each time step t, the algorithm performs three high-level operations:

1. **Expansion:** Each of the W active hypotheses is extended with all possible next tokens from the vocabulary
2. **Scoring:** The likelihood (or log-probability) of these new, extended hypotheses is computed
3. **Pruning:** From all the generated candidates, only the top-W hypotheses with the highest scores are selected to become the active beams for the next time step

Maintaining W active beams leads to significantly better computational efficiency than an exhaustive search while avoiding local errors that a greedy approach might make.

1.3 CTC Beam Search vs. Auto-Regressive Beam Search

While the high-level concept of maintaining W "beams" is shared, the algorithmic implementation differs significantly between the CTC beam search and AR beam search paradigms. The two strategies are tailored to different model architectures and their sequential conditional dependencies. CTC beam search is predominantly used with non-auto-regressive models (like purely acoustic models in ASR) that output frame-level probabilities independent of previous decoding decisions. On the other hand, AR beam search is integral to auto-regressive models, such as decoders in large language models (LLMs) or sequence-to-sequence machine translation models, where each output token's generation explicitly depends on the full sequence of previously generated tokens. In this project we focus on CTC Beam Search decoding, consequently we will derive optimizations specific to its implementation which may not generalize to auto-regressive beam search decoding.

2 Opportunities and Challenges of GPU Parallelism

The inherent structure of CTC beam search presents both significant opportunities for massive parallelism on GPUs and unique challenges due to its dynamic and data-dependent nature.

2.1 Opportunities for Parallelism

The primary enabler of GPU acceleration in CTC beam search is the pre-computation of the acoustic model's log-probability tensor $\mathbf{P}$. Unlike auto-regressive models that require sequential model inference, CTC's independence across time steps for the acoustic probabilities means the forward pass of a neural network happens before the decoding logic allowing for greater flexibility in optimization without sacrificing correctness.

Specifically, at each time step t, the most parallelizable operations are:

- **Hypothesis Expansion:** For each of the W active beams, every token $c \in \mathcal{V}$ (including ϵ) is considered as a potential extension. This generates a total of $W \times |\mathcal{V}|$ new candidate prefixes. Each of these $W \times |\mathcal{V}|$ expansions can proceed in parallel, computing new scores based on the current beam's state and the $\mathbf{P}_{t,c}$ value. This regular, element-wise computation is ideal for the SIMT (Single Instruction, Multiple Threads) architecture of GPUs since thousands of threads can simultaneously compute scores and generate new candidate information.
- **Parallel Top-W Pruning:** The need to select the top-W candidates after expansion and merging also presents a significant parallelism opportunity. While naive sorting of $W \times |\mathcal{V}|$ elements is expensive, highly optimized parallel sorting algorithms (e.g., those in Thrust) are well-suited for GPUs.

These algorithmic steps offer clear opportunities to saturate GPU cores with arithmetic operations and high-throughput data processing, making GPUs attractive for accelerating CTC decoding.

2.1.1 Batched Processing

Beyond inherent algorithmic parallelism opportunities, GPU accelerators are well suited to batched processing which is common in DL applications. If multiple input sequences (e.g., audio utterances) are decoded simultaneously, the entire beam search process can be parallelized across the batch dimension. This means that B independent beam search instances can execute concurrently, allowing for higher GPU occupancy and more efficient utilization of compute resources, especially for common operations like shared memory reductions and global memory reads.

2.2 Challenges in Mapping to SIMT Architectures

Despite the clear opportunities, mapping the entire CTC beam search loop onto SIMT introduces several critical challenges, primarily arising from the dynamic and irregular nature of the algorithm:

- **Irregular Memory Access for State Tracking:** Beam search is fundamentally a traversal of a dynamic prefix tree (or trie). As hypotheses evolve, fork, and merge, the underlying data structures supporting these prefixes can become highly fragmented. Storing beam states (e.g., prefix tokens, scores, parent pointers) in traditional tree-like or linked-list structures leads to sparse, indirect memory access patterns. This results in poor cache locality and non-coalesced global memory transactions, which severely degrades performance on GPU.
- **Prefix Merging and Deduplication:** This is perhaps the most challenging step. Multiple paths can collapse to the same output prefix (e.g., 'ABBB-' and 'AB–' both yield "AB"). Identifying and grouping identical prefixes across $W \times |\mathcal{V}|$ candidates, and then efficiently reducing their scores, is difficult to parallelize without introducing high synchronization overhead (e.g., atomics) or significant data movement. We chose to use a hashing scheme to address, introduced later.
- **Load Imbalance and Thread Divergence:**
  - **Variability in Merges:** The number of candidates that merge into a unique prefix can vary significantly between different prefixes and time steps. This leads to imbalanced work distribution during the merge/reduce phase.
  - **Dynamic Beam Pruning:** While the expansion might be fixed at $W \times |\mathcal{V}|$ candidates, the actual number of unique, surviving candidates after merging can fluctuate. As discussed later, we applied additional pruning criteria leading to decreases in effective beam width leading to warp divergence.
- **Efficient Parallel Top-W Selection:** After expansion and merging, the system must select the top-W beams globally or per-batch from the candidate set $\mathcal{C}$, which can contain up to $W \times |\mathcal{V}|$ elements. For large vocabularies sorts can become prohibitely expensive on GPU.

These challenges manifested themselves during development requiring carefully designed, custom CUDA kernels that exploit GPU architectural specifics (e.g., shared memory, warps, global memory coalescing, `__syncthreads` barriers) to achieve high performance.

2.3 CTC Beam Search Algorithm

The discussion above highlights the iterative nature of CTC beam search and the specific points amenable to, or challenging for, GPU parallelism.

Algorithm 1 formalizes this process, explicitly marking the opportunities for parallel execution within the decoding loop. The "Parallel Hypothesis Expansion" and "Parallel Merge and Prune" steps represent the critical kernels where GPU acceleration can yield significant speedups over CPU-bound implementations.

Algorithm 1: CTC Beam Search Decoding

Input: Log-probabilities $\mathbf{P} \in \mathbb{R}^{T \times |\mathcal{V}|}$, Beam width W

Initialize $\mathcal{H} \leftarrow \{(\text{prefix} = \emptyset, P_b = 0, P_{nb} = -\infty)\}$
for $t \leftarrow 1$ ***to*** T **do**
  $\mathcal{C} \leftarrow \emptyset$ `// Parallel Hypothesis Expansion`
  for ***each*** *beam* $\ell \in \mathcal{H}$ ***in parallel*** **do**
    for ***each*** *token* $c \in \mathcal{V}$ ***in parallel*** **do**
      if $c = \epsilon$ **then**
        $P'_b \leftarrow \log(e^{P_b(\ell)} + e^{P_{nb}(\ell)}) + \mathbf{P}_{t,\epsilon}$
        $P'_{nb} \leftarrow P_{nb}(\ell) + \mathbf{P}_{t,\text{last}(\ell)}$
        Add (ℓ, P'_b, P'_{nb}) to $\mathcal{C}$
      end
      else
        $\ell_{new} \leftarrow \ell + c$
        if $c = \textit{last}(\ell)$ **then**
          $P'_{nb} \leftarrow P_b(\ell) + \mathbf{P}_{t,c}$
        end
        else
          $P'_{nb} \leftarrow \log(e^{P_b(\ell)} + e^{P_{nb}(\ell)}) + \mathbf{P}_{t,c}$
        end
        $P'_b \leftarrow -\infty$
        Add $(\ell_{new}, P'_b, P'_{nb})$ to $\mathcal{C}$
      end
    end
  end
  `// Parallel Merge and Prune`
  Merge candidates in $\mathcal{C}$ with identical prefixes and sum log-probs
  $\mathcal{H} \leftarrow$ Top-$W(\mathcal{C})$ based on $P_{total} = \log(e^{P_b} + e^{P_{nb}})$
end
return Best hypothesis in $\mathcal{H}$

2.4 Adaptive Beam Width Scheduling

Standard beam search employs a fixed width W for the entire sequence. However, this is not always the most efficient approach.

In the early stages of decoding, the distribution of prefixes is highly uncertain (high entropy), requiring a wide beam to capture potential optimal paths. As the sequence progresses, the distribution often sharpens, and the probability mass concentrates on fewer hypotheses making a large beam width redundant. Motivated by this observation, we implement adaptive beam width scheduling which attempts to exploit this property by dynamically adjusting W_t.

2.5 Baseline Implementations and Benchmarks

To evaluate the performance and accuracy of our custom CUDA decoder, we identified several widely used CTC decoding libraries to serve as benchmarks.

CPU Benchmarks:

- **Flashlight:** The primary reference for our comparisons is Flashlight Kahn et al. [2016], a C++ beam search decoder backend used by TorchAudio. However, it is a CPU-bound library, meaning it has limited opportunity for parallelism and can incur significant overhead from device-to-host memory transfers when processing the output of acoustic models running on GPU.
- **pyctcdecode:** We also benchmarked against pyctcdecode Technologies [2022], a popular library in the Hugging Face ecosystem. The library creates a Python binding for C++ logic and is also CPU-based. We evaluated both its standard sequential mode and its batched mode to provide a fair comparison against our batched GPU implementation.

GPU Benchmarks and Limitations:

- **TorchAudio CUCTCDecoder (Deprecated):** TorchAudio previously included an experimental GPU decoder under the *CUCTCDecoder* module. However, this module has been formally deprecated and removed from recent library versions due to maintenance challenges. We have used this library recently, but since embarking we were unable to get a full run with this implementation.
- **NVIDIA FlexCTC (NeMo):** We attempted to integrate NVIDIA's FlexCTC, provided within the NeMo framework Kuchaiev et al. [2019]. From our reading this appears as the closest competitor platform-wise to what we implemented, however, we found the library to have prohibitive integration overhead, particularly when trying to run the standalone decoder. Consequently, we were unable to establish a stable, isolated benchmark for FlexCTC within our testing environment.

Therefore, our primary performance analysis focuses on the speedup relative to the robust CPU baselines (Flashlight, pyctcdecode, and pyctcdecode-batched), as we were unable to find and/or integrate a readily available GPU alternative currently existing in the PyTorch ecosystem.

3 Approach

3.1 Data Structures

Our implementation parallelizes CTC beam search decoding by decomposing the expansion, merging, and pruning stages into distinct CUDA kernels that operate concurrently across the batch and beam dimensions. The decoder maintains the state of all active hypotheses in flat, contiguous arrays on the GPU, following a structure-of-arrays layout so that consecutive threads access consecutive memory locations. This layout minimizes memory divergence and allows for coalesced global memory accesses.

To make merging of duplicate prefixes amenable to highly parallel sorting and reduction, each candidate hypothesis is assigned a compact integer key encoding both its batch index and a rolling hash of its output-sequence prefix. Let $b \in \{0, \dots, B-1\}$ denote the batch index and $\{c_1, \dots, c_L\}$ the token sequence, with $c_t \in \{0, \dots, V-1\}$. A 32-bit rolling hash is maintained according to

$$h_0 = 0, \qquad h_t = (\alpha h_{t-1} + (c_t + 1)) \bmod 2^{32}, \quad t = 1, \dots, L,$$

for a fixed multiplier α. To implement the DJB2 string hash, we use $\alpha = 33$; empirically, this value yields favorable hash value distributions while meaningfully mixing new token additions. At runtime, the 32-bit key space is partitioned into k_{batch} bits for the batch index and k_{hash} bits for the hash, with

$$k_{\text{batch}} = \lceil \log_2 B \rceil, \qquad k_{\text{hash}} = 32 - k_{\text{batch}}.$$

The packed key for a candidate at time t in batch b is then

$$\text{key} = (b \ll k_{\text{hash}}) \;\big|\; \big(h_t \;\&\; (2^{k_{\text{hash}}} - 1)\big),$$

so the high bits uniquely identify the batch and the low bits encode a truncated rolling hash of the prefix. Within a fixed batch, two different prefixes collide only if their hashes agree modulo $2^{k_{\text{hash}}}$.

For fixed N, increasing k_{hash} strictly decreases the collision probability. The dynamic bit allocation used here chooses k_{batch} just large enough to represent all batch indices, dedicating all remaining bits to the hash. While the overall hashing optimization may well lead to reduced decoding performance due to hash collisions, it enables the massively parallel sort and reduce operations which form the backbone of this implementation's speedup over CPU.

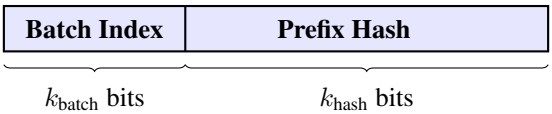


Figure 1: Bit-wise layout of the packed key used for parallel sorting.

3.2 Parallel Hypothesis Expansion

At each time step, an expansion kernel extends every active hypothesis with all possible tokens in parallel. However, extending with the entire vocabulary is computationally wasteful, as most tokens have negligible probability. To mitigate this, we implemented a pre-expansion token pruning step. Prior to expansion, a dedicated kernel selects only the top-K most likely tokens from the acoustic model's output distribution for each batch. This kernel is utilizes shared memory to perform reductions on the probability distribution in an effort to reduce pressure on global memory.

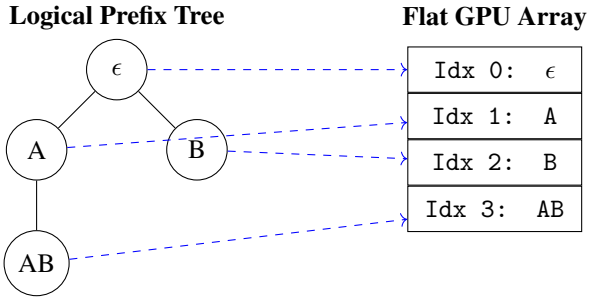


Figure 2: Mapping of the logical dynamic prefix tree to a static flat array structure in GPU global memory to enable coalesced access.

We launch the expansion grid where each thread is responsible for a single candidate extension from this pruned set. The kernel reads the current beam state and the log-probabilities for the selected tokens. It then applies CTC-specific transition rules: extending with a blank token updates the blank score, while extending with a non-blank token updates the non-blank score. The kernel handles the "repeated token" case locally: if the new token matches the previous token of the beam, it can only transition from the previous blank state to avoid collapsing the repeat. The result is a set of expanded candidates with updated scores and new rolling hashes. Threads that generate invalid extensions mark their candidates with invalid keys to be filtered out during the sort.

3.3 Parallel Reduction and Merging

After keys and candidate probabilities are computed, we employ the Thrust library to merge colliding prefixes. First, `thrust::sort_by_key` groups all candidates sharing the same key. Then, `thrust::reduce_by_key` aggregates these groups, summing their log-probabilities to compute the total probability of each unique prefix. This step merges all beams that yield the same prefix within a batch, barring collisions resulting from the hashing routine. Following reduction, we use `thrust::gather` to retrieve auxiliary data for the unique candidates in order to prepare them for the selection phase.

3.4 Top-K Selection

Once unique candidates are merged and their total scores are computed, a pruning kernel selects the top-W beams for the next time step. First, we sort the unique candidates within each batch by their total score. Then, the `top_k` kernel - parallelized over batches - uses binary search to locate the start of each batch's segment in the sorted array. It copies the top W candidates to the beam state buffers for the next iteration. Simultaneously, it records the parent indices and tokens into a history buffer. This history is later used by a kernel responsible for reconstructing hypotheses to backtrack and recover the final output sequences after the last time step.

3.5 Dynamic Beam Width Scheduling

A key optimization in our CTC beam search decoder is dynamic beam width adjustment driven by the entropy of the active beam distribution. Standard CTC beam search maintains a fixed beam width k across all time steps, even though the effective diversity of hypotheses can decrease as decoding progresses. In our experiments, we observed that the top hypotheses tend to converge toward a smaller set of highly similar prefixes. This empirical trend motivated our hypothesis that beam

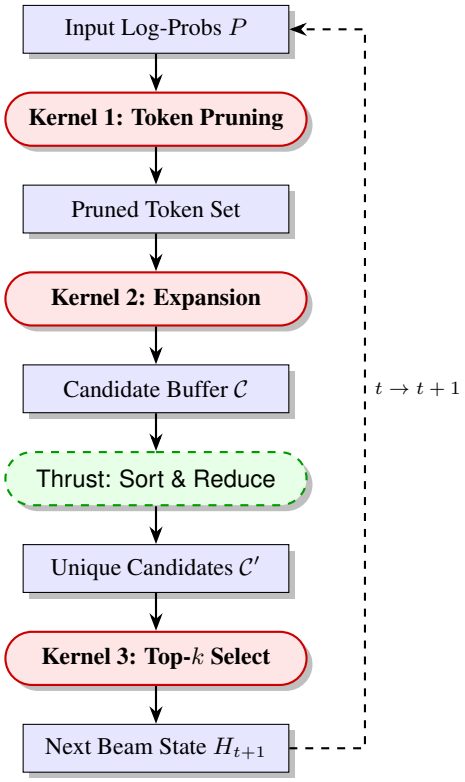


Figure 3: Overview of the GPU kernel pipeline. The process alternates between custom CUDA kernels (red) and Thrust library primitives (green) to transform data residing in global memory buffers (purple).

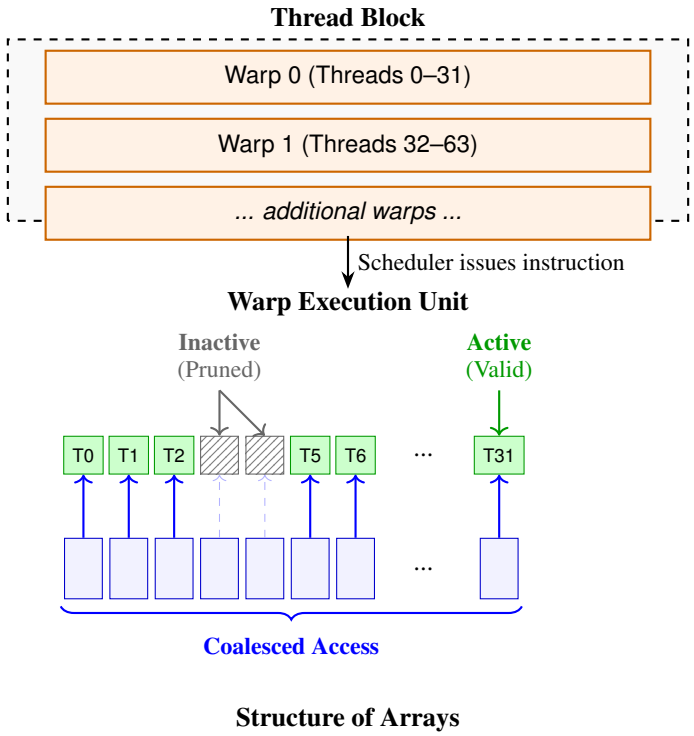


Figure 4: Mapping of CUDA kernels to hardware Warps

diversity generally diminishes over time. So, maintaining a large k in later steps yields diminishing accuracy returns while incurring compute and memory overhead.

Based on this hypothesis, we parametrize beam width using an exponential decay schedule $w(t) = Ae^{Bt} + C$ with a minimum width floor. The parameters for this decay are provided by one of three schedulers: a naive static schedule, a lookup-table (LUT) policy, and a learned multilayer perceptron (MLP) policy.

3.5.1 Naive Static Schedule

The naive schedule provides a simple, easily interpretable baseline using a fixed exponential decay. The scheduling interface supports an initial beam width and a configurable number of time steps during which the beam should remain at this initial value before decay begins. It also supports a minimum beam width that acts as a strict lower bound.

This is a valuable baseline because it isolates the benefit of the time-dependent contraction of the problem independent of any other factors. However, because the parameters are static this schedule cannot dynamically preserve larger beam widths during decoding of more ambiguous segments or accelerate contraction in confident segments. This may lead to over-pruning in challenging segments or under-pruning in simpler segments.

3.5.2 LUT Scheduler

The LUT scheduler extends the naive baseline by conditioning decay behavior on an entropy-derived uncertainty metric. It maintains a two-dimensional table of precomputed decay parameters across discretized time and entropy bins. Entropy history is smoothed via fixed look back window to reduce noise. Then, it is mapped into time and entropy indices that select the corresponding schedule entry. The LUT can be quickly generated and does not require a training pipeline like the MLP scheduler.

The primary advantage of this approach is that it provides entropy sensitivity with minimal runtime overhead and strong interpretability. The main limitation of this approach is expressiveness. Discretization constrains the scheduler's ability to capture more uncertain trajectories. Moreover, since the LUT is ultimately a fixed policy, adapting it to a different scheduling hypothesis will require regeneration and retuning.

3.5.3 MLP Scheduler

The MLP scheduler replaces the discretized LUT with a compact learned model. The model maps a history vector of size 3 to three normalized outputs (α, β, γ) that are transformed into the decay parameters (A, B, C). The model has a single hidden layer reflecting the relatively simplistic nature of the learned exponential decay schedules. The size of this hidden layer was selected to be 2 neurons through a benchmark against the LUT scheduler (Table 1). This design enables a continuous policy that can encode more complex dependencies than the LUT while still respecting the interface specified beam ranges.

Table 1: Architecture search results for the entropy MLP scheduler.

Hidden	MSE	MAE	MLP (ns)	LUT (ns)	Ratio	Memory
16	0.000105	0.006853	77.2	5.0	15.44×	460 B
12	0.000108	0.006923	83.1	5.0	16.62×	348 B
8	0.000064	0.004631	68.6	4.8	14.29×	236 B
6	0.000109	0.007175	65.8	5.0	13.16×	180 B
4	0.000108	0.006934	56.1	5.0	11.22×	124 B
3	0.000148	0.007807	55.4	5.3	10.45×	96 B
2	0.000104	0.006970	50.0	5.3	9.43×	68 B
1	0.016631	0.096473	48.5	4.8	10.10×	40 B

The primary benefit of the MLP approach is adaptivity. With appropriate synthetically generated training data, it can learn when to slow contraction to protect accuracy when there is consistent uncertainty and when to aggressively prune to recover performance. There is a significant query performance dropoff compared to the LUT scheduler due to the computational complexity of inference.

Additionally, the training pipeline must be run prior to any use of the MLP scheduler making it less amenable to rapid experimentation and prototyping as compared to the other presented alternatives.

3.5.4 Accuracy Comparison

Across all benchmarks and diagnostics the MLP scheduler is considerably more accurate than the LUT at predicting the entropy-conditioned decay parameters (5). The parameter error distributions are tightly grouped and centered near zero and the output curves generated the MLP consistently match the ground truth across a variety of entropy trajectories (5). The LUT has much more variance in its error distributions and produces more aggressively contracted schedules in several instances (5). This could lead to over-pruning at time steps where there is more uncertainty (i.e. higher entropy).

However, the accuracy advantage of the MLP comes with the direct cost of substantially more expensive queries. The MLP inference time is an order of magnitude greater than a table lookup. The LUT has essentially constant query time and very minimal latency. In practice, these results suggest a tradeoff. The LUT scheduler is preferable when scheduling overhead must be negligible while the MLP scheduler is preferable when its higher accuracy is expected to recover enough computation time to offset the higher per-query cost.

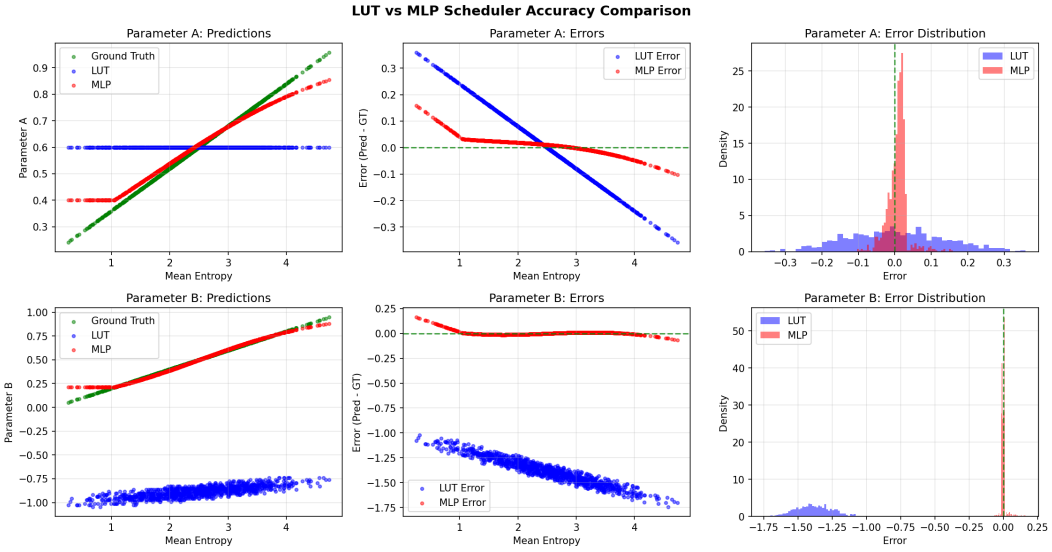


Figure 5: LUT vs. MLP accuracy comparison for decay parameters A, B. Evaluated on 1000 synthetic entropy histories at time step $t = 50$.

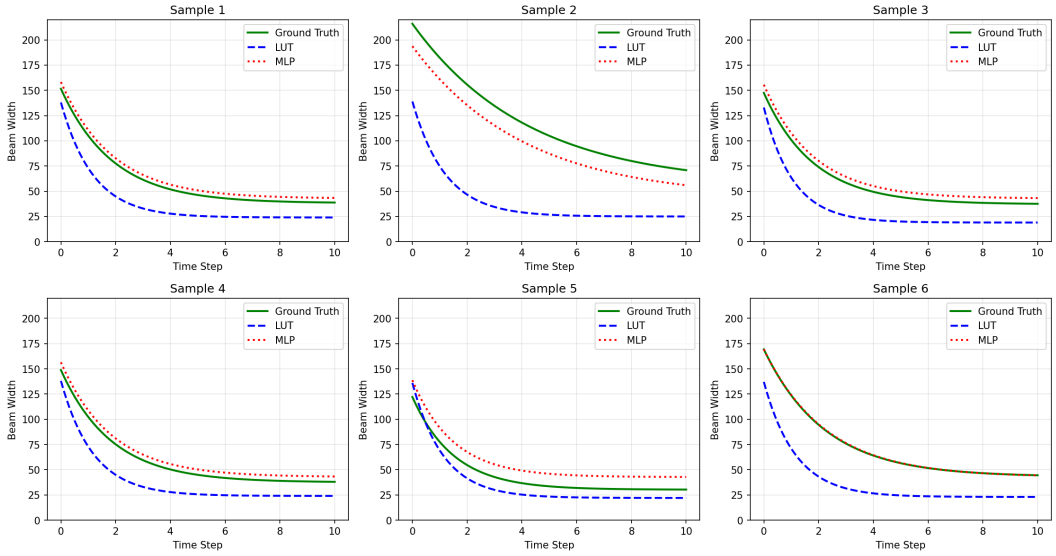


Figure 6: Representative exponential decay beam-width curves $f(x) = Ae^{Bx} + C$ samples, comparing ground truth to LUT and MLP predicted schedules. Generated with $W_{min} = 10, W_{max} = 200$.

4 Evaluation Framework

Figure 7 illustrates the data flow through our evaluation framework.

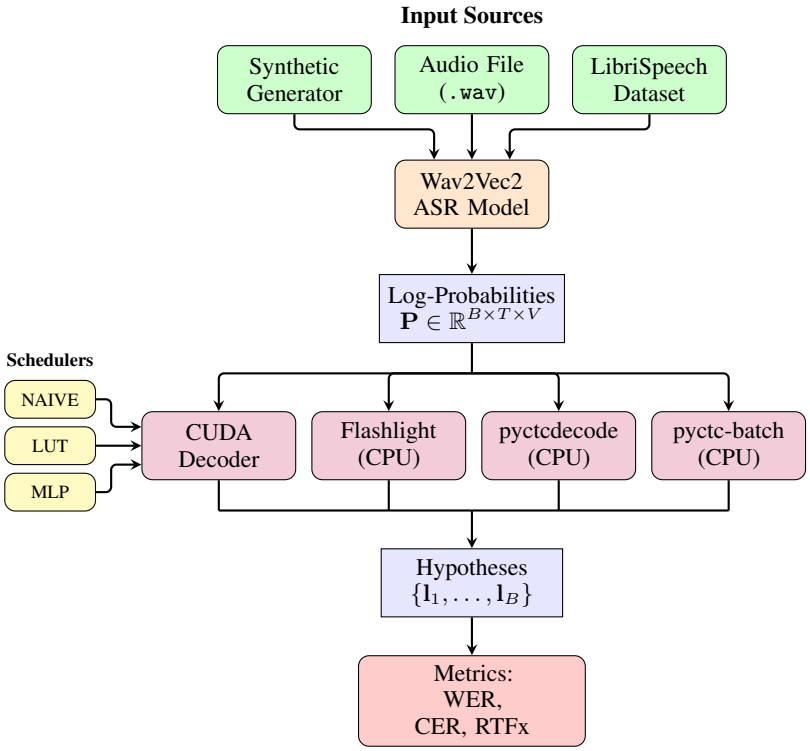


Figure 7: Testing pipeline data flow. Input sources (green) feed into the Wav2Vec2 acoustic model (orange), producing log-probability tensors. The CUDA decoder (pink, left) accepts adaptive beam width schedulers (yellow), while CPU baselines include Flashlight, pyctcdecode, and pyctcdecode-batched. All outputs are evaluated using standard ASR metrics (red).

4.1 Input Modes

Our framework supports three input modes:

- **Synthetic mode** generates random log-probability tensors $\mathbf{P}_{b,t,v} = \log \text{softmax}(\mathbf{Z})$ where $\mathbf{Z} \sim \mathcal{N}(0, 1)$; for rapid correctness verification without acoustic model overhead.
- **Real audio mode** processes `.wav` files through the acoustic model for end-to-end testing.
- **Benchmark mode** systematically evaluates on the LibriSpeech `test-other` dataset Panayotov et al. [2015], which contains acoustically challenging recordings to stress-test the decoders.

4.2 Acoustic Model

We use Wav2Vec2 Baevski et al. [2020] (`WAV2VEC2_ASR_BASE_960H`) as our acoustic model, which outputs frame-level log-probabilities over a 29-token vocabulary. The acoustic model forward pass is excluded from decoder timing measurements to ensure fair comparison.

4.3 Metrics

Real-Time Factor (RTFx) measures throughput as the ratio of audio duration to decoding time; an RTFx of 100 means the decoder processes audio 100× faster than real-time. We report **Word Error Rate (WER)** and **Character Error Rate (CER)** to measure transcription accuracy via Levenshtein distance.

5 Results

5.1 Development Setup

For development, we ran a Google Cloud Platform `n1-highmem-2` instance equipped with a dual-core Intel Haswell CPU (13 GB RAM) and a single NVIDIA Tesla T4 GPU (16 GB VRAM). We utilized the Google Deep Learning VM image running Debian 11 (Bullseye) with CUDA 12.4.

5.2 Benchmarking Setup

For experiments, we benchmarked on a Lenovo Legion Pro 7i Gen 10 laptop equipped with an Intel Core Ultra 9 275HX CPU (24 cores, up to 5.4 GHz) and an NVIDIA GeForce RTX 5070 Ti Laptop GPU, running Ubuntu 24.04. On this system, we use PyTorchs Linux nightly wheels built against CUDA 12.8, together with the corresponding NVIDIA driver.

5.3 Beam Width Sensitivity

The results in Figure 8 demonstrate the scaling characteristics of our GPU-accelerated approach compared to CPU-based benchmarks. The `torchaudio-flashlight` benchmark shows a strong linear dependence on beam width, with decoding time increasing from about 0.6s at $W = 10$ to over 10.0s at $W = 100$. This seems to indicate that some less-optimized CPU implementations struggle to handle the increased computation load of larger beam width since they process hypothesis expansions mostly sequentially. In contrast, our `cuda-beamseach` implementation exhibits much shallower scaling, peaking around 1s of total decoding time. The vanilla `cuda-beamsearch` performs the worst of the CUDA flavors. LUT produces the fastest CUDA decoding. This validates the effectiveness of our parallel expansion and reduction kernels, which can amortize the cost of processing additional candidates at higher beam widths. Interestingly, `pyctcdecode` remains competitive even at higher beam widths, likely due to the inherent sparsity of CTC outputs leveraged by the CPU's superior branch prediction. Furthermore, a beam width of 100 is likely insufficient to saturate the GPU threads giving the CPU implementation a chance due to its higher clock speed. Additionally, we note that the results demonstrate a markable improvement in the adaptive schedule schemes to decoding time, demonstrating improvement in runtime without loss of accuracy, seen in Figure 10.

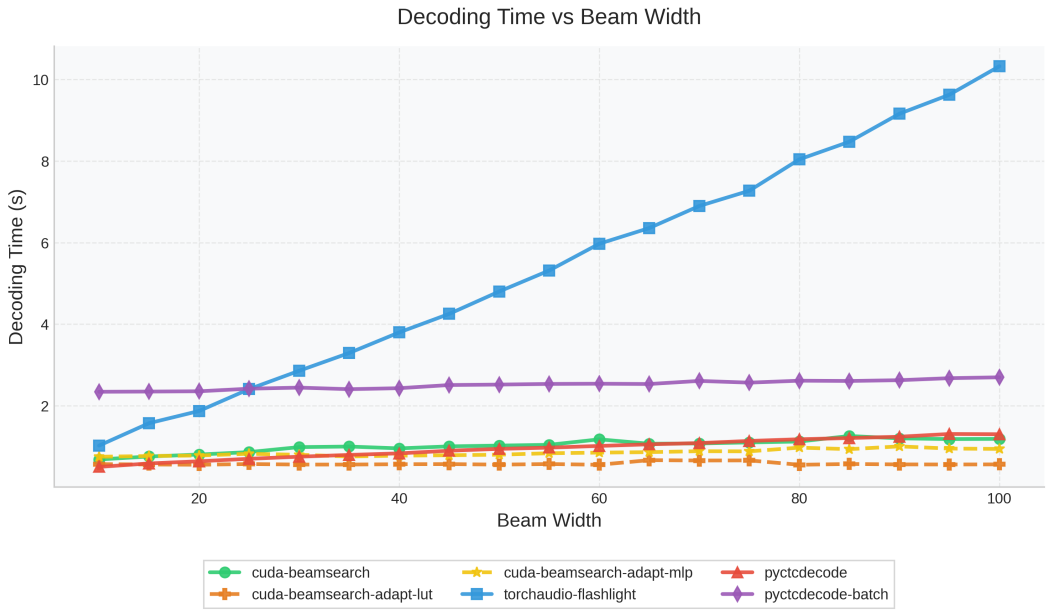


Figure 8: Total Decoding Time(s) as a function of Beam Width (W), comparing the performance of our CUDA implementation against CPU baselines. Batch size $B = 32$, evaluated on 200 samples.

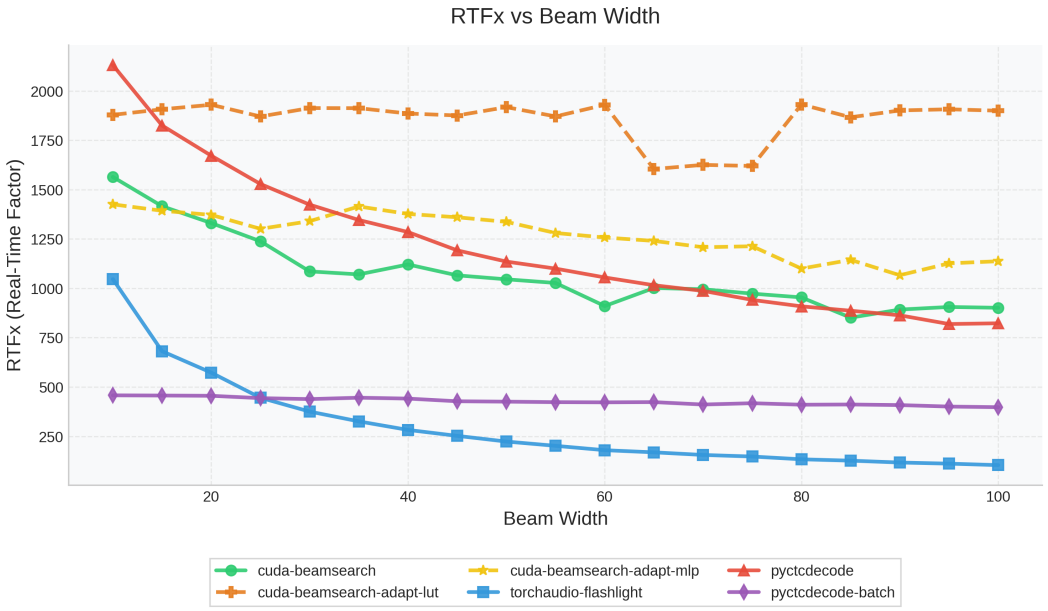


Figure 9: Real-Time Factor (RTFx) as a function of Beam Width (W), comparing the throughput efficiency of our CUDA implementation against CPU baselines. Batch size $B = 32$, evaluated on 200 samples.

Looking at figure 9 we get a clearer picture of decoding efficiency over looking at raw latency. For narrow beam widths, ($W = 10$), `pyctcdecode` achieves the highest peak throughput of ≈ 2100 RTFx. This is somewhat expected as for small workloads the GPU launch overhead dominates, whereas the CPU more efficiently leverages its higher clockspeed. However, the scaling becomes clearer when looking at larger beam widths. `pyctcdecode` exhibits an approximately linear decay, losing around 50% of its throughput as beam width increases to 100. This suggests the CPU decoding task is compute-bound by the sequential hypothesis expansion, a step that we perform in parallel in CUDA decoders. In contrast, our `cuda-beamsearch-adapt-lut` implementation demonstrates remarkable stability, maintaining a nearly flat RTFx profile (≈ 1750–$1900\times$) across the entire range of beam widths. The consistent performance suggests the GPU may be under-saturated, meaning the cost of additional candidates is effectively amortized. Thus, while the CPU is highly efficient for trivial search spaces, our CUDA implementation offers superior scalability, sustaining high-throughput performance where CPU baselines begin to bottleneck.

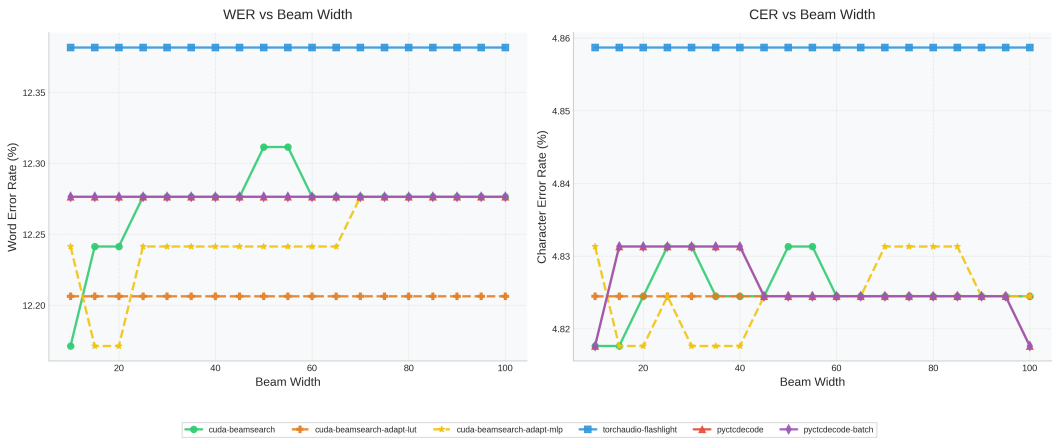


Figure 10: WER and CER as a function of Beam Width (W). Batch size $B = 32$, evaluated on 200 samples.

We observe consistently low WER and CER from our CUDA decoders, matching or going below the CPU-based decoders. The LUT version performs most consistently overall, while the MLP flavor dipped to the lowest overall WER. Among the CPU decoders we observe WERs consistent with our CUDA decoders from the `pyctcdecode` and `pyctcdecode-batch` implementations, with the Flashlight implementation consistently producing slightly higher WER and CER than all other decoders tested. The results show our CUDA decoders do not significantly sacrifice accuracy for better decoding throughput.

5.4 Batch Size Sensitivity

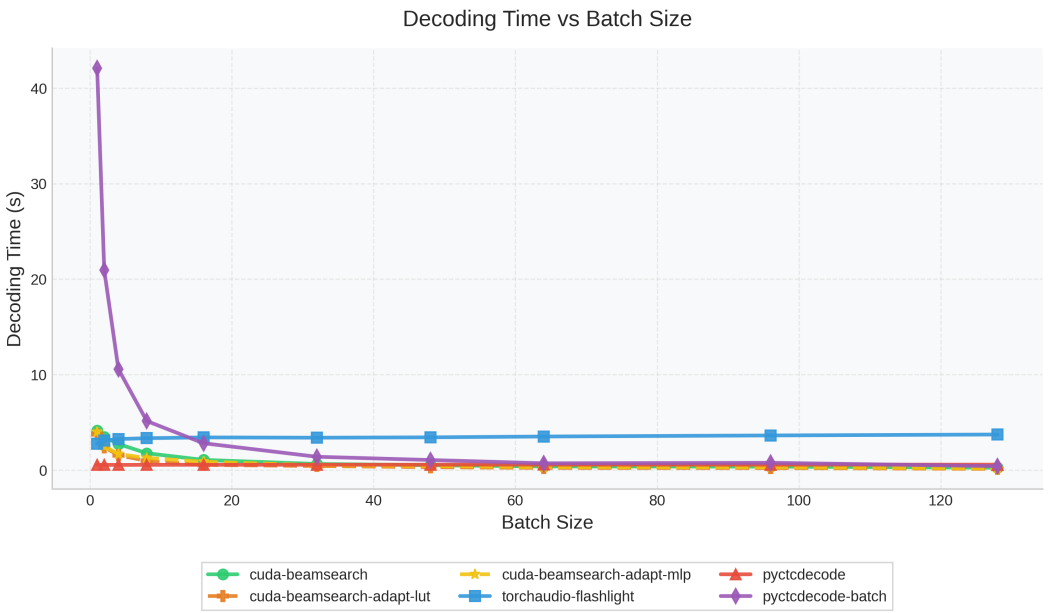


Figure 11: Decoding latency as a function of batch size comparing our implementation to standard CPU benchmarks. Beam width $W = 50$, evaluated on 128 samples.

Figure 11 shows the decoding time versus batch size, and demonstrates competitive performance of our implementation against CPU-based baselines. Notably, the high latency for `pyctcdecode-batch` at small batch sizes is due to the process initialization overhead. This decoder instantiates a `multiprocessing.Pool` for every inference pass. Consequently, at low batch size the fixed cost of forking Python processes dominates. In contrast, our CUDA implementation avoids the per-batch instantiation penalty thanks to a more persistent GPU context. Per-batch decoding times do decrease for the naive, LUT, and MLP-scheduled flavors of our implementation as scheduler initialization overheads become amortized across batches, but remain competitive against all CPU benchmarks for all tested batch sizes.

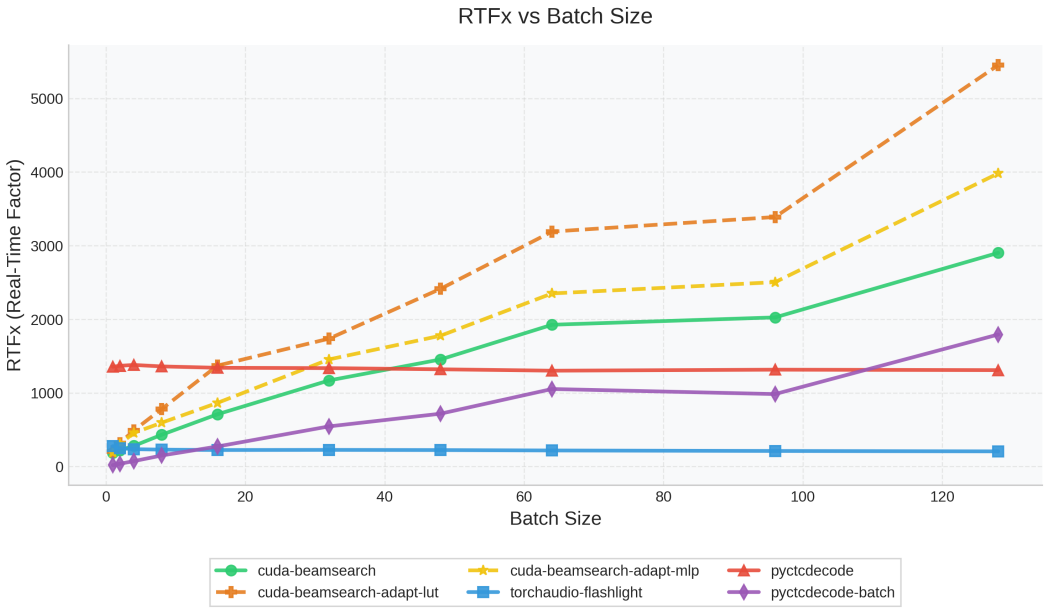


Figure 12: Real-Time Factor (RTFx) as a function of Batch Size (B), comparing the throughput efficiency of our CUDA implementation against CPU baselines. Beam Width $W = 50$, evaluated on 128 samples.

Figure 12 demonstrates the large throughput advantage of our CUDA implementation over baselines. The CPU-based `pyctcdecode-batch` scales initially but shows more modest gains beyond $B = 64$, likely due to saturation of work across processor cores. Our implementation, however, exhibits approximately linear scaling up to the maximum tested batch size at which it exceeds 5000 RTFx. At smaller batch sizes, one of our tested CPU benchmarks outperforms the GPU implementation, presumably due to the overhead of CUDA kernel launches and memory transfers. With larger batch sizes these overheads are amortized. The trend shown by our implementation indicates that we have not yet hit the hardware throughput limit, so higher throughput could be achieved with larger batch sizes. Our testing was limited by available GPU memory on our machine.

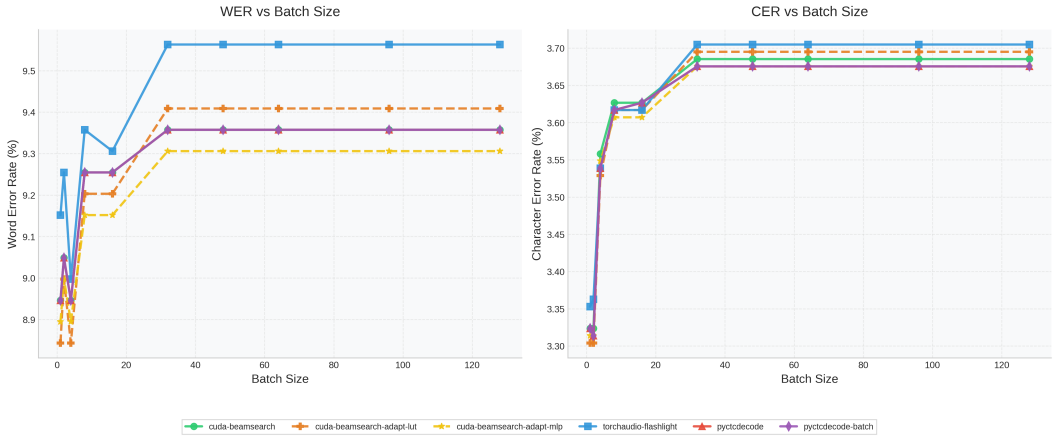


Figure 13: WER and CER as a function of Batch Size (B). Beam width $W = 50$, evaluated on 128 samples.

Interestingly, the accuracy of our method decreases with increased batch size. This was an intriguing result, as it directly visualizes the impact of hash collisions on accuracy. As more bits of the beam hashes are dedicated to representing the batch to which they corresponds, the string hash space

decreases and collision probability increases. To the best of our knowledge, this is the only source of error in our implementation that would increase at larger batch sizes. However, it remains at least as competitive as CPU-based baselines across all batch sizes.

5.5 Overall Speedup

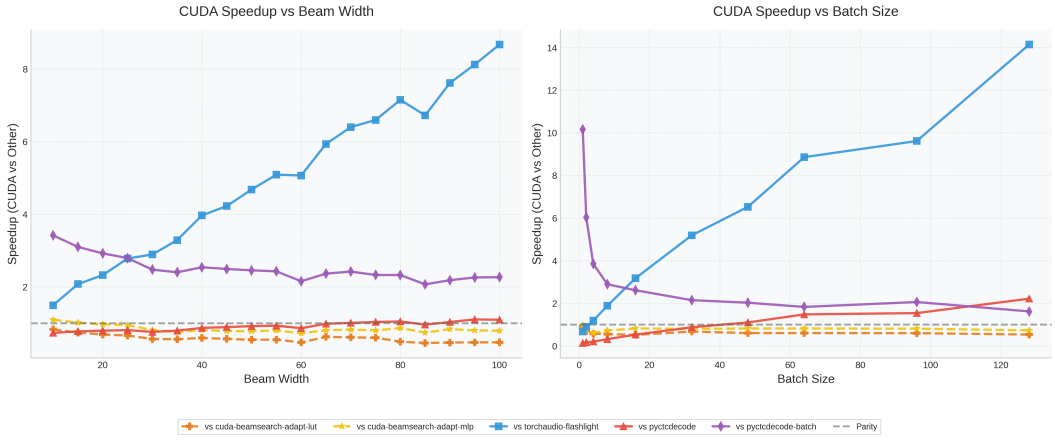


Figure 14: Relative Speedup of CUDA implementation vs CPU baselines across Beam Width (left) and Batch Size (right).

Against the `torchaudio-flashlight` baseline, our CUDA implementation achieves a speedup that scales linearly with beam width. This confirms that while CPU implementations are very sensitive to the size of the search space, our CUDA kernels successfully amortize the cost of wider beams. Against the optimized `pyctcdecode-batch` the parallelism of the SIMT architecture of the GPU still provides a more than $2\times$ speedup. So, our implementation outperforms even highly-optimized CPU implementations.

The adaptive beam width variants of our pipeline (LUT and MLP) are both faster than the static implementation. The adaptive LUT scheduler is roughly $2\times$ faster than the static baseline. This trend validates the hypothesis that dynamically narrowing the beam width based on entropy alllows for computational savings without sacrificing parallelism.

We see the same trend of throughput advantage for the CUDA implementations in the Speedup vs. Batch Size experiment. The speedup against `torchaudio-flashlight` grows linearly as the CPU implementation processes sequences serially. Against `pyctcdecode-batch` the speedup converges to appoximately $2\times$ for $B > 60$. So, while batched CPU decoding is effective, there is a hardware ceiling that it is surpassed by GPU implementations.

6 Discussion

6.1 Iterative Optimization

In our first implementation, we implemented the trie structure by keeping each candidate beam as a standalone integer array. On each expansion, the parent beam would be deep-copied and appended with the new token to produce a new candidate beam. This was not only slow, but also saw memory usage explode exponentially. Next, we attempted to optimize the memory usage by using a pointer-based scheme. This did, as expected, considerably reduce memory utilization since we now forewent the storing each beam as an independent integer array. However, performance was still poor in the sorting and pruning portions of the implementation - we hypothesize that this was due to thread divergence seen in traversing the tree to access node values in kernels. Finally, we arrived at the conclusion that performance would be most favorable if we could compute and access

necessary beam information in constant time, for which we designed a flat table data structure indexed by a rolling string hash of the beam tokens. This design enabled us to compute offsets for the desired table entries and access data in constant time. This implementation also is more amenable to memory coalescing in the deduplication and selection stages.

The string hash optimization also serves as a core optimization in our approach. Despite sacrificing correctness for runtime, the optimization avoids chasing pointers back to root to collect tokens in order to pairwise-compare beams for deduplication. Instead, we store beam representations as a single integer at each node of the abstract trie. We quickly ran into an implementation error when beam hashes did not include information on their batch index, which was then added to make up half of the hash. However, we saw an accuracy decrease when switching to the hashing method. We traced this down to hashing collisions that were occuring. Motivated by hash collision probability decreasing with hash bit length, we changed the hashing approach to adaptively allocate only enough bits to represent the batch index, and to dynamically allocate the remaining bits towards the rolling string hash. This improved accuracy considerably, though we note that this remains a limitation of the implementation.

Finally, we implemented dynamic beam width schedules to further optimize runtime. Motivated by the observation that model uncertainty typically decreases as decoding progresses, we sought to adaptively prune the beam width over time. We initially experimented with fixed, user-defined decay schedules, but found this impractical as it required manual hyperparameter tuning for every decoding task. Instead, we observed that the entropy of the token distribution tends to decay exponentially during decoding. Leveraging this property, we developed an adaptive scheduler that estimates the entropy decay rate from the initial time steps. This estimate is then used to parameterize an exponential decay curve for the beam width, allowing the decoder to automatically tighten its search space as confidence increases. This approach yields significant runtime benefits, particularly for long sequences, without requiring manual intervention.

6.2 Limitations and Continued Work

Our implementation contains several limitations that present opportunities for future optimization. First, our decoder cannot scale to arbitrarily sized inputs due to the capacity bounds of GPU memory. While CPU baselines are also memory-limited, system RAM capacity is typically larger than device memory. Furthermore, our current approach pre-allocates state buffers based on the maximum sequence length to ensure contiguous memory access. A future improvement would be to implement a dynamic system where inputs are batched through the decoder dynamically based on total available GPU memory, rather than allocating for the worst-case scenario before commencing.

Another limitation arises from our packed key representation, which compresses both the batch index and the prefix hash into a single 32-bit integer. This creates a fundamental trade-off: increasing the batch size requires more bits for the batch index, leaving fewer bits for the hash. This elevates the probability of hash collisions, where distinct hypotheses are erroneously merged, degrading accuracy. While our dynamic bit allocation mitigates this, scaling to massive batch sizes remains risky. Moving to larger key representation would eliminate this bottleneck but would incur a cost in memory bandwidth, which are already the primary bottlenecks of the algorithm. Thus, perhaps it would be most promising to implement a dynamically-sized hash scheme.

Additionally, our pre-expansion pruning kernel relies on loading the probability distribution of the entire vocabulary into shared memory to perform fast reductions. Given the shared memory limit on CUDA architectures, our decoder is effectively limited to vocabularies smaller than several thousand tokens. This prevents the use of modern Large Language Model (LLM) tokenizers which often exceed tens of thousands tokens, for example.

Finally, our adaptive beam width scheduler calculates a global beam width based on the average entropy of the entire batch. This assumes a homogeneous difficulty distribution across input samples. In real-world scenarios where a batch contains a mix of clear and noisy signals, the averaged schedule may be too simple of an assumption. A per-sample adaptive scheduler could resolve this but may introduce thread divergence.

7 Student Work Distribution

Group Member	Specific Contributions	% Contribution
Julius Arolovitch	• Developed hashing scheme and structure of arrays memory layout. • Developed kernels for expansion, reduction, reconstruction, and parent beam search kernel. • Developed pruning of tokens in expansion stage, inspired by `pyctcdecode`. • Developed first iteration of fixed schedule with user-defined schedule parameters. • Refactored initial implementations to improve code hygiene. • Assisted in benchmarking, plot generation, writing non-scheduler portions of the Approach, results discussion, Iterative Optimization, and Limitations & Continued Work sections.	33.3%
Benjamin Kleyner	• Initial naive kernel implementation with expensive deep copies • Dynamic beam width schedulers implementation • Dynamic beam width schedulers benchmarking • MLP synthetic data generation and training pipeline • Sensitivity study generation script • Initial results collection and analysis • Data structure and warp mapping diagrams • Wrote abstract • Wrote sections detailing dynamic schedulers • Created data structure, kernel pipeline, and mapping to SIMT diagrams	33.3%
Maxim Yagnyatinskiy	• Implemented evaluation framework (synthetic, `.wav`, LibriSpeech) • Integrated kernels with dynamic beam width schedulers (entropy history, beam width adjustments) • Configured PyTorch/Python bindings and integrated initial kernels • Developed setup guide, configured build toolchains and dependencies • Wrote background, evaluation framework sections • Assisted in writing results analysis • Setup, configured GCP VM instance • Maintained project website (https://418-beamers.github.io/)	33.3%

References

Alexei Baevski, Yuhao Zhou, Abdelrahman Mohamed, and Michael Auli. wav2vec 2.0: A framework for self-supervised learning of speech representations. In *Advances in Neural Information Processing Systems*, volume 33, pages 12449–12460. Curran Associates, Inc., 2020.

Alex Graves, Santiago Fernández, Faustino Gomez, and Jürgen Schmidhuber. Connectionist temporal classification: labelling unsegmented sequence data with recurrent neural networks. In *Proceedings of the 23rd international conference on Machine learning*, pages 369–376, 2006.

Jacob Kahn, Vineel Vineet, Vineel Pratap, Awni Hannun, Vitaliy Liptchinsky, Gabriel Synnaeve, and Ronan Collobert. Flashlight: Enabling innovation in speech recognition. In *Proceedings of the 33rd International Conference on Machine Learning*, 2016.

Oleksii Kuchaiev, Jason Li, Huyen Nguyen, Oleksii Hrinchuk, Ryan Leary, Boris Ginsburg, Samuel Kriman, Stanislav Beliaev, Vitaly Lavrukhin, Jack Cook, et al. Nemo: a toolkit for building ai applications using neural modules. *arXiv preprint arXiv:1909.09577*, 2019.

Vassil Panayotov, Guoguo Chen, Daniel Povey, and Sanjeev Khudanpur. Librispeech: an ASR corpus based on public domain audio books. In *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5206–5210. IEEE, 2015. doi: 10.1109/ICASSP.2015.7178964.

Bhiksha Raj. Deep learning: Sequence to sequence models - connectionist temporal classification. Lecture Slides, Fall 2025, 2025. Slides 97–100.

Kensho Technologies. pyctcdecode: A fast and feature-rich ctc beam search decoder for speech recognition. `https://github.com/kensho-technologies/pyctcdecode`, 2022.