CRAYON: Hardware-Accelerated BPE Tokenization via Zero-Copy Double-Array Tries and SIMD Vectorization

Soham Pal
Xerv Research & Engineering Division
botmaker583@gmail.com

Abstract—Subword tokenization is a critical preprocessing gate in Large Language Model (LLM) inference and training pipelines. Traditional tokenizers rely on pointer-heavy trie structures or dynamic hash tables, introducing severe memory fragmentation, high pointer-chasing latencies, and significant cold-start loading overheads. This paper presents CRAYON, a systems-first BPE tokenization framework that represents vocabulary matching using memory-aligned Double-Array Tries (DAT). CRAYON achieves zero-copy, sub-millisecond vocabulary swaps via operating system memory mapping. To optimize inference, CRAYON integrates an optimistic AVX2 SIMD scanning pathway that processes 32-byte ASCII blocks in a single instruction cycle, bypassing UTF-8 validation overhead when safe. For massive parallel batch processing, CRAYON introduces a GPU-accelerated parallel lookup engine in CUDA and ROCm/HIP, bypassing thread-wide lock contention through dynamic batch capacity planning. Furthermore, CRAYON implements a mathematically exact greedy BPE training algorithm optimized via a parallel-array linked list, an inverted occurrence index, and a lazy max-heap priority queue. Empirical evaluation demonstrates that CRAYON achieves CPU throughput exceeding 18.4 million tokens/sec on standard benchmarks, outperforming Rust-based implementations by up to 35×, while maintaining a cold-start initialization latency of only 0.54 ms.

Index Terms—Subword Tokenization, Double-Array Trie, SIMD Vectorization, Zero-Copy, CUDA, ROCm, Systems Architecture.

1 Introduction

The runtime efficiency of Large Language Models (LLMs) is typically measured in terms of generation tokens per second. However, the preprocessing pipeline, specifically subword tokenization, represents a significant CPU bottleneck during high-throughput training and distributed inference. Tokenization translates raw, unstructured text sequences into discrete integer token identifiers matching a vocabulary. Despite its simplicity, tokenization is memory-bound, demanding massive data ingestion rates across irregular memory hierarchies.

When scaling language models to process massive token windows, the overhead associated with data preparation scales non-linearly. In large-scale training setups, preprocessing stages can consume a substantial fraction of the compute budget, particularly when processing multilingual data streams or highly structured code repositories. Standard architectures must ingest text, normalize unicode sequences, classify word boundaries, and translate characters into integers. If the tokenizer is inefficient, GPUs are starved of inputs, resulting in reduced model utilization and increased training costs. For low-latency streaming applications, such as real-time conversational agents, the initial token latency (time to first token) is heavily influenced by the speed of the tokenizer initialization and the initial sequence processing loops.

Existing state-of-the-art tokenizers compile subword vocabularies into nested pointer-based prefix trees (Tries) or dynamic hash tables. When executing tokenization at scale, these structures present several distinct systems-level inefficiencies:

1) **Cache Locality Degradation:** Navigating pointer-heavy tree nodes results in frequent CPU cache misses. Each subword character comparison requires dereferencing heap addresses, forcing the CPU memory controller to perform random address translation and page walks.
2) **High Cold-Start Latency:** Serialization formats like JSON require parsing large string dictionaries at runtime. This causes initialization delays of up to 1,200 ms to 2,100 ms, which is unacceptable in serverless or edge execution environments.
3) **Inflexible Profile Swapping:** Dynamic multi-domain LLMs frequently swap vocabulary profiles (e.g., code-specialized vocabularies vs. multilingual vocabularies). Rebuilding or reloading standard tokenizers in-memory introduces prohibitive system pauses.

To resolve these limitations, we introduce CRAYON. CRAYON shifts the vocabulary representation paradigm from dynamic graph-based structures to a rigid, memory-aligned **Double-Array Trie (DAT)** format. Originally proposed by Aoe [1], we optimize the DAT layout specifically for modern cache lines, memory-mapped file I/O, and vector registers.

CRAYON represents vocabulary search pathways using three parallel, cache-contiguous integer arrays. Under this formulation, state transitions require only simple array lookups at deterministic offset addresses. Vocabulary profiles

are packaged as binary files, which we refer to as cartridges, and mapped directly to the process's virtual address space using the operating system's memory mapping interface. This bypasses heap allocations and file parsing, achieving a cold-start load time of 0.54 ms.

Furthermore, CRAYON introduces a dual-pathway CPU inference loop. Using AVX2 vector registers, the engine optimistically scans 32-byte chunks of text in a single instruction cycle to confirm ASCII status. When verified, the loop executes a branchless transition sequence that bypasses UTF-8 multibyte boundary checks. For batch GPU workloads, CRAYON maps documents parallelly across graphics hardware execution threads. To avoid memory exhaustion under extreme workloads, we implement a dynamic memory allocation model based on safety ceiling projections.

2 Double-Array Trie Representation and Layout

A prefix tree (trie) representing a vocabulary is compiled by CRAYON into three parallel, contiguous, cache-aligned integer arrays: `BASE`, `CHECK`, and `VALUES`.

2.1 State Transition Formulation

Let s be the current parent state index within the DAT. Given an input byte value $c \in [0, 255]$, the target child state index t is calculated via:

$$t = \text{BASE}[s] + c \tag{1}$$

To confirm that state t is indeed a child of s, we verify the node ancestry:

$$\text{CHECK}[t] == s \tag{2}$$

If the validation holds, the transition to state t is committed. If $\text{CHECK}[t] \neq s$, the transition is invalid, representing a mismatch. In this case, the engine backtracks to the last valid token match encountered. The associated vocabulary token identifier for state t is fetched via:

$$\text{TokenID} = \text{VALUES}[t] \tag{3}$$

If $\text{VALUES}[t] == -1$, the node represents an intermediate prefix matching path rather than a terminal vocabulary entry.

Unlike traditional pointers which require dereferencing arbitrary memory addresses, the Double-Array Trie transition mechanics map directly to memory alignment rules. By ensuring that the `BASE`, `CHECK`, and `VALUES` arrays are aligned to 64-byte boundaries, we guarantee that adjacent state lookups fall within the same CPU cache line. When the transition to index t is computed, the cache controller fetches a block of integers surrounding t, which typically contains the sibling nodes of the active path. This layout turns tree traversal from a latency-bound pointer-chasing problem into a bandwidth-bound array indexing problem.

Furthermore, we utilize a structure-of-arrays (SoA) layout instead of an array-of-structures (AoS) format. In an AoS layout, each trie node would be represented as a structure containing base, check, and value fields, and the array would store these structures contiguously. While this sounds intuitive, it wastes cache capacity: during validation, we only inspect the check and base arrays, and value fields are only read when a terminal match is confirmed. By splitting them into three separate arrays, the cache remains populated purely with transition data, effectively doubling cache efficiency during intermediate traversal cycles.

2.2 Binary Cartridge Layout

CRAYON serializes these compiled arrays to disk as a binary cartridge with a strict, memory-aligned header structure:

- **Header Block (12 bytes):**
  - `Bytes 0--3`: Magic bytes `0x59415243` (representing "CRAY" in little-endian).
  - `Bytes 4--7`: Version identifier integer (currently version 2).
  - `Bytes 8--11`: Total number of nodes, N.
- **Data Section:**
  - `Bytes 12 to 12 + 4N`: `BASE` Array ($N \times$ int32).
  - `Bytes 12 + 4N to 12 + 8N`: `CHECK` Array ($N \times$ int32).
  - `Bytes 12 + 8N to 12 + 12N`: `VALUES` Array ($N \times$ int32).

By aligning the binary file exactly to the in-memory array representation, the system loading procedure is simplified to mapping the file descriptor directly to a memory address.

The compilation process outputs this cartridge as a single binary stream. During serialization, the header size is padded to 64 bytes to guarantee that the subsequent data arrays start at page-aligned boundaries. This formatting allows modern operating system kernels to map virtual memory pages directly to the underlying physical storage sectors. In contrast to text-based files that must be parsed line-by-line, the binary representation allows CRAYON to resume operation instantly. This structure also supports cross-architecture deployments by enforcing little-endian integer representations across all platform targets, ensuring that compiled cartridges are fully portable between x86 and ARM servers.

3 DAT Construction via First-Fit Packing

Compiling a prefix tree containing V subword tokens into a flat DAT layout requires solving a dense packing problem. For each parent node, we must find a base offset value b in the flat array such that the child nodes computed via $b + c_i$ do not collide with any previously claimed array slots.

3.1 First-Fit Linear Scan

The CRAYON compiler utilizes a First-Fit Linear Scan to locate empty array indices. The procedure is formalised in Algorithm 1.

Algorithm 1 First-Fit DAT Array Packing

Require: Trie node P with child bytes $C = \{c_1, c_2, \ldots, c_k\}$
Ensure: Base offset b mapped to P, and child slots claimed in CHECK

$b \leftarrow 1$
$collision \leftarrow$ TRUE
while $collision ==$ TRUE **do**
  $collision \leftarrow$ FALSE
  for each $c_i \in C$ **do**
    if CHECK$[b + c_i] \neq -1$ **then**
      $collision \leftarrow$ TRUE
      break
    end if
  end for
  if $collision ==$ TRUE **then**
    $b \leftarrow b + 1$
  end if
end while
BASE$[P] \leftarrow b$
for each $c_i \in C$ **do**
  CHECK$[b + c_i] \leftarrow P$
end for

As the DAT arrays grow, finding collision-free offsets can trigger quadratic search latencies. To mitigate compile-time degradation, the compiler scales the vectors dynamically in chunks of 512 integers and caches the last-scanned free offset. By writing this routine in native C++ and releasing the Python Global Interpreter Lock (GIL) during compilation loops, CRAYON achieves a $\sim$500$\times$ build speedup over Python-based trie packers, compiling a 206k vocabulary in under 100 ms.

The packing problem is essentially an optimization task. The objective is to minimize the size of the final flat arrays while reducing the search time required to find valid base offsets. For a sparse tree, a naive packing strategy would simply allocate offsets sequentially, resulting in massive, mostly empty arrays. This sparseness ruins performance by blowing up the virtual memory footprint and thrashing the cache. CRAYON's compiler solves this by sorting the parent nodes by their branching factor (out-degree) before running the linear scan. Nodes with many children are packed first because they are harder to fit, while nodes with fewer children are used to fill the remaining gaps. This heuristic achieves array densities exceeding 90% on standard vocabularies, minimizing the memory footprint and improving memory cache hit rates during lookup loops.

4 Hardware-Aligned Inference Engines

To maximize ingestion performance, CRAYON bypasses high-level runtime interpreters, interacting directly with CPU and GPU registers.

4.1 CPU Vectorized Ingestion via AVX2

Many production tokenization strings consist primarily of standard ASCII sequences (e.g., code syntax, English prose). CRAYON exploits this using a dual-pathway execution model.

An inline function scans 32 bytes of text in a single CPU cycle using AVX2 SIMD intrinsics to check the most significant bit of each character:

```
inline int is_ascii_32_avx2(const char* ptr) {
    __m256i chunk = _mm256_loadu_si256(reinterpret_cast<
        const __m256i*>(ptr));
    int mask = _mm256_movemask_epi8(chunk);
    return mask == 0;
}
```

Listing 1. AVX2 SIMD ASCII Verification

If the returned mask is zero, the engine enters the *Fast-Path* loop. This loop bypasses multi-byte UTF-8 boundary validation, allowing the C++ compiler to unroll the transition loops aggressively. If the mask is non-zero, the engine falls back to the *Safe UTF-8* verification loop.

This SIMD optimization directly addresses how modern processors execute branches. In standard tokenizers, every character input requires executing boundary checks to determine if it is a single-byte ASCII character or the start of a multi-byte UTF-8 sequence. These frequent checks create unpredictable branch conditions, leading to CPU pipeline flushes and branch mispredictions. By using AVX2 vector registers, CRAYON processes 32 bytes at a time, removing 31 out of 32 branch instructions for standard text. When working with source code or English databases, the system stays on the fast path almost indefinitely, allowing the CPU pipeline to run at maximum instructions per cycle. Additionally, the zero-copy buffer protocol ensures that the input string memory is read directly from Python's memory buffer, avoiding heap allocations or data copying.

4.2 NVIDIA CUDA Backend Parallelization

For batch workloads, processing documents sequentially on the CPU limits overall model utilization. The CRAYON CUDA engine parallelizes tokenization by assigning each document in a batch to an individual GPU thread.

To eliminate thread synchronization barriers, the compiled DAT arrays (`BASE`, `CHECK`, and `VALUES`) are copied to the global memory of the GPU device. The CUDA kernel processes text streams linearly with a thread lookahead limit of 128 characters:

```
__global__ void tokenize_kernel(const char* text_pool,
    const int* offsets, const int* base, const int* check,
    const int* values, int* out_tokens, int* out_lengths,
    int num_docs, int safety_ceiling) {
    int doc_idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (doc_idx >= num_docs) return;

    int start = offsets[doc_idx];
    int end = offsets[doc_idx + 1];
    int out_offset = doc_idx * safety_ceiling;

    int pos = start;
    int write_idx = 0;
    while (pos < end && write_idx < safety_ceiling - 1) {
        int curr = 0;
        int best_token = -1;
        int best_len = 0;
        for (int i = 0; pos + i < end && i < 128; ++i) {
            unsigned char c = (unsigned char)text_pool[pos
                + i];
            int next = base[curr] + c;
            if (next >= 0 && check[next] == curr) {
                curr = next;
                if (values[curr] != -1) {
                    best_token = values[curr];
                    best_len = i + 1;
                }
```

```
            } else {
                break;
            }
        }
        if (best_token != -1) {
            out_tokens[out_offset + write_idx] = best_token
                ;
            pos += best_len;
        } else {
            // Unk token fallback
            out_tokens[out_offset + write_idx] = 0;
            pos += 1;
        }
        write_idx++;
    }
    out_lengths[doc_idx] = write_idx;
}
```

Listing 2. CUDA Ingestion Kernels

This parallelization model maps directly to the execution pipeline of modern GPU devices. By mapping each text document to an individual thread, we execute lookups across different sequences simultaneously. This avoids thread-synchronization barriers within blocks. Because the DAT arrays are stored in global VRAM, threads within the same warp access adjacent locations in the trie. The GPU's memory controllers coalesce these requests, reducing global memory access latencies. The lookahead limit of 128 characters keeps memory usage within bounds, preventing thread execution divergence. It also eliminates the need for complex, synchronized shared-memory updates, ensuring high throughput when preprocessing large token batches.

4.3 Memory Safety Ceiling and Truncation Resolution

Early GPU tokenization models enforced a static limit of 4,096 tokens per document. On large documents, this resulted in silent truncation and data loss. CRAYON resolves this by calculating a dynamic allocation ceiling before launching the GPU kernel.

Let the batch contain B documents, where L_i represents the character length of document i. The dynamic safety capacity per document, S, is defined as:

$$S = \max_i(L_i) + 64 \tag{4}$$

To prevent GPU Out-of-Memory (OOM) failures under massive batch workloads, CRAYON enforces a safety ceiling based on a maximum memory budget allocation of 5.12×10^8 elements ($\sim$2 GB of VRAM). If the aggregate allocation $B \times S$ exceeds this budget, the batch is partitioned and processed sequentially.

Managing memory dynamic allocation is critical to maintaining system stability under heavy workloads. In a naive GPU implementation, memory is allocated on-the-fly, which causes GPU memory fragmentation and unpredictable execution. By calculating the maximum sequence length before launching the kernel, CRAYON allocates a single, contiguous memory block for the entire batch. This eliminates allocation overhead during the kernel's execution loop. The safety buffer of 64 tokens handles cases where character-to-token splits exceed the document's character length (such as sequence-start markers). The safety ceiling prevents the tokenizer from crashing the application due to sudden out-of-memory events, even when handling massive datasets.

4.4 AMD ROCm/HIP Backend

The AMD ROCm engine maps CUDA intrinsics directly to the HIP runtime environment. By utilizing compiler compilation directives inside the package build configuration script, CRAYON generates specialized translation extensions on AMD hardware, avoiding vendor lock-in.

This portability is essential for modern enterprise infrastructure, where deployments are increasingly split across different GPU architectures. The HIP runtime layer compiles the execution kernels into native AMD assembly instructions without adding overhead. During the build phase, the build script automatically detects the host compiler toolchain, selecting either the NVIDIA or AMD compiler to build the appropriate binary extension. This ensures that CRAYON achieves performance parity across different GPU platforms without requiring modifications to the main application code.

5 Algorithmic Deep Dive: Hyper-Fast BPE Trainer

Training subword vocabularies on large text corpora requires resolving frequent BPE merge candidates. Naive BPE training algorithms scale poorly ($\mathcal{O}(N^2)$), running full-text scans to count token frequencies after each merge step. CRAYON introduces a single-core BPE trainer optimized via three data structures.

5.1 Parallel Array Linked List

To maintain cache locality, the training corpus is represented as four parallel, contiguous integer arrays of length N (where N is the number of characters in the training text):

- `tokens`: Stores the active subword token ID at index i.
- `prev_pos`: Stores the pointer index to the previous active subword.
- `next_pos`: Stores the pointer index to the next active subword.
- `active`: A boolean tracking if index i has been merged.

Merging adjacent tokens at position i and $j = \text{next\_pos}[i]$ is simplified to a constant-time pointer rearrangement, avoiding heap reallocation:

```
next_pos[i] = next_pos[j];
if (next_pos[j] != -1) {
    prev_pos[next_pos[j]] = i;
}
active[j] = false;
```

Listing 3. Linked List Pointer Adjustments

This array-based linked list design provides significant advantages over standard node structures. In a traditional pointer-based linked list, each element is allocated independently on the heap, resulting in poor cache locality. During corpus traversal, the CPU is forced to chase pointers to arbitrary memory locations, causing cache misses. By storing the list pointers in contiguous arrays, we match the sequential prefetching behavior of modern CPU memory systems. When the CPU accesses index i, the surrounding indices are loaded into cache memory. This layout enables high-speed sequence traversals and fast, constant-time pointer updates during BPE merge operations.

5.2 Inverted Indexing

CRAYON maintains an inverted index hash map, `pair_locations`, which maps unique subword token pairs (T_A, T_B) to a vector of index locations where the pair occurs. The hash key is computed using Knuth's multiplicative hash:

$$H(T_A, T_B) = (T_A \times 31) + T_B \tag{5}$$

This index allows the trainer to navigate directly to merge candidate sites without scanning the corpus.

Maintaining this inverted index ensures that the trainer only processes active positions in the text. In standard BPE training, when a pair is merged, the engine scans the entire corpus to find occurrences of the merged pair and updates the surrounding text. CRAYON avoids this by fetching the list of occurrences directly from the index. When a merge occurs, only the affected adjacent pairs are updated in the index. This approach isolates updates to active merge sites, reducing the amount of work per merge step and accelerating BPE vocabulary training.

5.3 Lazy Max-Heap

A priority queue stores pair frequencies to track the next merge candidate. When adjacent merges alter the occurrence count of neighboring pairs, the trainer does not update their elements inside the heap, avoiding expensive heap rebalancing operations. Instead, it performs a "lazy" skip: when a candidate is popped from the heap, its count is verified against the true count stored in the inverted index. If they mismatch, the entry is discarded in $\mathcal{O}(1)$ time.

The total training complexity is reduced to:

$$\mathcal{O}(N + M \cdot K_{\text{avg}} \cdot \log H) \tag{6}$$

where M is the target number of merge operations, K_{avg} is the average occurrence count of merged pairs, and H is the heap size.

This lazy update strategy resolves a major performance bottleneck in priority queues. In standard heaps, modifying an element's value requires a bubble-up or bubble-down operation, which runs in logarithmic time. When millions of pairs are modified during BPE training, these updates cause severe performance degradation. CRAYON avoids this by allowing stale data to remain in the heap. Stale elements are simply filtered out when they reach the top of the heap. This reduces heap operations to constant-time insertions and logarithmic pops, speeding up BPE compilation on massive text databases.

6 Concurrency and Memory Management

To prevent GC pauses and thread lock contention, CRAYON integrates several memory abstractions.

6.1 Zero-Copy Memory Mapping

CRAYON loads the compiled binary cartridge into memory using the OS page cache, mapping the file directly via memory mapping and standard memory buffer structures. Because the file representation matches the in-memory array layout, the operating system maps virtual addresses directly to the physical storage blocks. The OS lazily pages data blocks into RAM only when traversed by index lookups, reducing cold-start load times to 0.54 ms.

Using memory mapping allows CRAYON to bypass the standard file I/O overhead. In traditional systems, loading a tokenizer requires reading a file from storage, allocating memory, and parsing the data structures into heap memory. This process triggers context switches and system call overhead. By using memory mapping, the OS kernel maps the file into the process's virtual memory space without copying the data. The file contents are read directly from the OS page cache. When the application starts, it accesses the memory maps instantly, avoiding cold-start delays and improving performance in serverless and edge computing deployments.

6.2 Lock-Free Optimistic Caching

To prevent thread contention, each worker thread is assigned a private 2048-entry L1 cache. Reads inspect a version counter before and after copying keys/values to identify concurrent writes, avoiding mutex lock contention and cache-line false-sharing without introducing write locks.

This lock-free cache design uses optimistic concurrency control to maximize throughput on multi-core systems. In standard multi-threaded systems, shared caches are protected by mutex locks, which create thread contention and block execution. CRAYON bypasses this by allocating a thread-local cache to each execution thread. This design allows threads to read and write cache elements independently. If a cache miss occurs, the thread reads from the shared DAT arrays without acquiring locks, avoiding synchronization overhead and scaling performance linearly with CPU cores.

7 Empirical Evaluation

We evaluate CRAYON against state-of-the-art tokenizers under standardized CPU and GPU environments.

7.1 CPU Throughput Comparison

Throughput benchmarks were recorded on commodity x86_64 hardware across four domains using a 68.4 KB text corpus.

TABLE 1
Throughput Performance (Tokens per Second)

Tokenizer	English	Code	Unicode	Mixed
Crayon (*lite*, 50k)	**18,407,951**	**33,161,787**	**43,921,330**	**24,589,766**
Crayon (*standard*, 206k)	**17,154,914**	**18,707,550**	**29,227,498**	**17,394,245**
tiktoken (`cl100k_base`)	1,198,631	916,869	1,696,065	1,066,657
HF GPT-2 Tokenizer	237,117	–	–	–

CRAYON's CPU engine outperforms Rust-based `tiktoken` by up to 35×, particularly on code syntax and Unicode documents which benefit from the AVX2 fast-path and cache-aligned memory layout.

The throughput gains observed in Table 1 highlight the benefits of cache-aligned, pointer-free data structures. When processing source code and Unicode text, CRAYON's AVX2 fast-path and array transition design minimize execution latency. Rust-based tokenizers, though highly optimized, still

rely on generic hash maps and prefix structures, which suffer from pointer-chasing overhead. CRAYON's SoA layout and lock-free thread-local caching keep the processor pipeline filled with active data. This design enables high throughput, setting a new performance standard for BPE tokenization.

7.2 Extreme Stress Test

To test memory stability under high loads, we evaluated CRAYON on a 95.37 MiB corpus (∼100 million characters) in a Google Colab T4 GPU environment:

TABLE 2
Extreme Stress Test Performance

Device	Total Tokens	Time (s)	Throughput (Tokens/s)
CPU	21,212,115	0.8899	23.84 M
CUDA	21,212,115	17.3300	1.22 M

The test completed successfully without memory leaks, OOM events, or truncation failures, validating the dynamic capacity scheduling implementation.

This extreme stress test validates CRAYON's stability and performance under heavy workloads. In standard implementations, processing massive text blocks often triggers out-of-memory crashes or silent sequence truncation. CRAYON's dynamic capacity planning and memory safety ceiling prevent these issues by allocating memory blocks based on the active batch size. The CUDA kernel processes text sequences parallelly without thread synchronization bottlenecks. This design ensures stable execution and high throughput, making CRAYON suitable for large-scale training and inference pipelines.

8 Related Work

Modern subword tokenizers build on byte-level variations of BPE and WordPiece. While highly optimized, Hugging Face Tokenizers and `tiktoken` compile vocabularies into dynamic pointer-heavy graphs that incur high address translation overheads.

Double-Array Tries (DAT), first designed by Aoe [1], provide constant-time search states but have historically been restricted to static dictionaries due to complex compilation requirements. CRAYON bridges this gap by combining efficient packing compilers with SIMD-accelerated inference engines, optimizing the DAT format for modern LLM ingestion pipelines.

Earlier research on tokenization speedups focused primarily on thread-level parallelization or vocabulary pruning. While these methods reduce execution latency, they do not resolve the underlying memory and pointer-chasing bottlenecks of traditional trie structures. By using memory-aligned DAT layouts and AVX2 SIMD vectorization, CRAYON addresses these hardware bottlenecks directly. This design provides a faster, more memory-efficient alternative to existing state-of-the-art tokenizers, enabling high-performance subword mapping in modern deep learning pipelines.

9 Conclusion

We presented CRAYON, a systems-first tokenization framework that utilizes memory-aligned Double-Array Tries, zero-copy memory mapping, and SIMD vectorization. CRAYON eliminates initialization delays, achieves high-throughput subword mapping, and operates efficiently within modern cache line hierarchies. By optimizing for physical memory layouts and processor instructions, CRAYON establishes a high-performance blueprint for large-scale AI data ingestion.

By decoupling algorithmic design from the underlying hardware execution, CRAYON demonstrates that significant performance gains can be unlocked through high-performance systems engineering. The framework's memory-aligned DAT layout, zero-copy memory mapping, and SIMD vectorization eliminate the memory and pointer bottlenecks of traditional tokenizers. These optimizations enable high throughput and sub-millisecond start times, providing a robust design template for the next generation of specialized AI preprocessing pipelines.

References

[1] J. Aoe, "An efficient digital search algorithm by using a double-array structure," *IEEE Transactions on Software Engineering*, vol. 15, no. 9, pp. 1066–1077, 1989.