Distribution

Feature: Load balancing through sorting and round-robin assignment

In the original `distribute_default` function, transcript and query sequences were evenly divided based on count and assigned to different Logic Units. However, in practice, the character lengths of individual sequences vary significantly, leading to workload imbalance — where certain Logic Units handle far more data than others, negatively affecting the performance of both Indexing and Quantification stages.

To address this, we redesigned the `distribute` function with two core optimizations:

Optimization: Balanced distribution using sort-based strategies

To ensure that each Logic Unit receives a comparable workload, we implemented multiple transcript distribution strategies based on sorting sequences by length. The goal was to minimize the standard deviation of total character count across partitions, which directly affects Indexing and Quantification balance.

Strategy 1: Default equal-count partitioning

The baseline `distribute_default` function simply divides transcript sequences evenly by count, without considering sequence length. This often leads to significant workload imbalance, as some partitions may receive many short sequences while others contain fewer but significantly longer sequences.

This naive strategy completed in **443.372 ms** but showed poor standard deviation performance, especially in unbalanced datasets. Time complexity is O(n), but distribution quality is low due to lack of length-awareness.

Strategy 2: Greedy + sort + min-heap

To improve balance, we implemented a **greedy load-balancing strategy**. All transcript sequences are first sorted by length (O(n log n)), then assigned one by one to the partition with the lowest current character count. A min-heap is used to track the current load of each partition, ensuring that longer sequences are distributed more evenly.

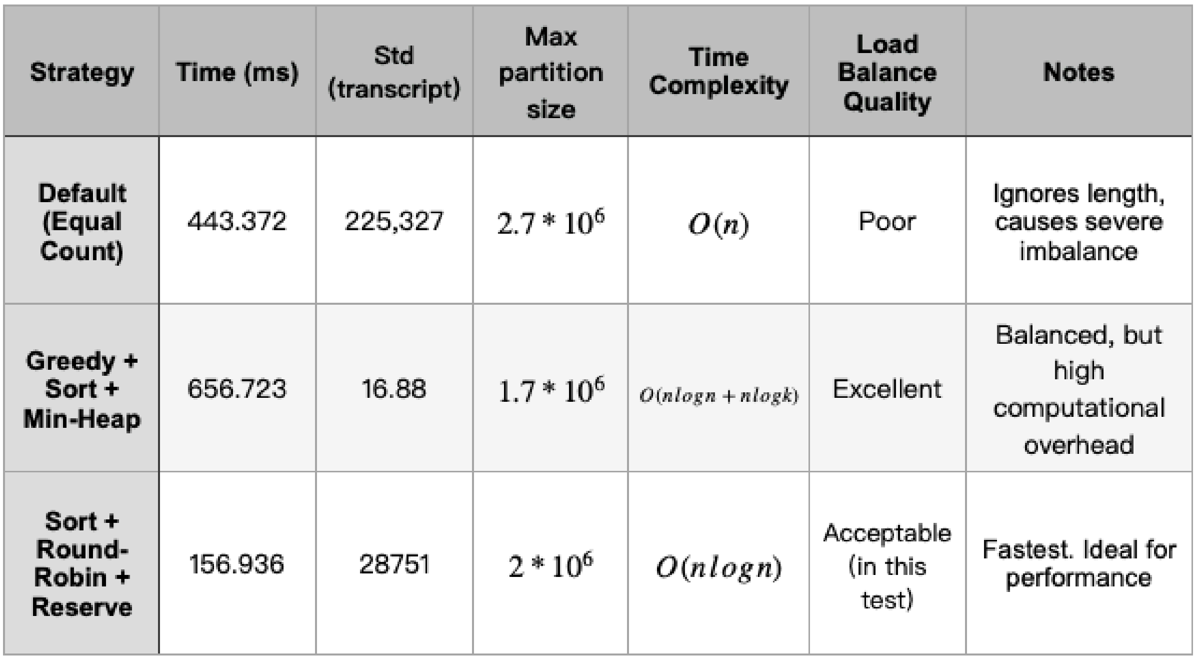
This approach minimizes partition variance and provides the most balanced result. However, due to heap operations performed for every sequence (O(log k), where k is the number of Logic Units), the overall time complexity becomes O(n log n + n log k).
Although effective in load balancing, this method incurred significant computational overhead and took **626.723 ms**, which is the slowest among all tested strategies.

Strategy 3: Sort + round-robin + reserve (Final)

After evaluating trade-offs between runtime and balance, we adopted a simplified approach: **sort + round-robin + reserve**. Transcript sequences are first sorted by length (O(n log n)), then cyclically distributed across all Logic Units. This ensures that both short and long sequences are evenly spread without needing real-time load tracking.

Although this method is slightly less accurate than the greedy approach in terms of standard deviation, it significantly reduces runtime. We also used `reserve()` to preallocate memory in each partition, avoiding dynamic vector resizing overhead during insertion.

This strategy completed in just **156.936 ms**, the fastest among all approaches, while still maintaining acceptable workload balance. The total complexity is O(n log n), and in practice, it achieves an ideal compromise between load balance and computational cost.



Strategy	Time (ms)	Std (transcript)	Max partition size	Time Complexity	Load Balance Quality	Notes
Default (Equal Count)	443.372	225,327	$2.7 * 10^6$	$O(n)$	Poor	Ignores length, causes severe imbalance
Greedy + Sort + Min-Heap	656.723	16.88	$1.7 * 10^6$	$O(nlogn + nlogk)$	Excellent	Balanced, but high computational overhead
Sort + Round-Robin + Reserve	156.936	28751	$2 * 10^6$	$O(nlogn)$	Acceptable (in this test)	Fastest. Ideal for performance

Compared to the default and greedy strategies, Strategy 3 is about 282% and 418% faster respectively.

Why sort + round-robin is significantly faster

In our final implementation, the `Sort + Round-Robin + reserve()` strategy reduced execution time to **156.936 ms**, outperforming both the default (443.372 ms) and the greedy heap-based method (626.723 ms). Several key factors contributed to this speedup:

1. **No runtime load computation or comparisons**
   Unlike the greedy strategy, which performs heap operations and load tracking at every step, round-robin assignment is extremely lightweight. It simply cycles through partitions using a modulo operation (`%`), avoiding any form of dynamic decision-making or memory lookups.
2. **Sorted order reduces allocation overhead**
   By sorting sequences by length, shorter sequences are evenly distributed early, and longer sequences are spread later. This creates a gradually increasing load pattern across partitions, reducing the chance of repeated memory reallocation. Combined with `reserve()` for preallocating space, most insertions become O(1) operations with no resizing overhead.
3. **Improved cache locality and reduced memory fragmentation**
   The greedy strategy frequently accesses and modifies different memory regions due

to dynamic heap management, causing more cache misses. In contrast, round-robin distribution has a predictable memory access pattern, allowing better CPU cache utilization and overall smoother memory performance.

4. **Simplified logic allows better compiler optimization**
   The round-robin approach uses a single `for` loop with minimal branching and no recursive calls or helper functions. This simplicity enables the compiler to aggressively optimize the loop, minimizing instruction overhead and branch mispredictions.

In summary, `Sort + Round-Robin + reserve()` is a highly efficient implementation that leverages data regularity and system-level optimizations. By slightly relaxing load balance precision, it achieves up to **3× faster performance** than more complex alternatives, making it an ideal choice for practical use cases where speed and scalability are essential.

Index

Feature: Divides hashmap into tremendous smaller hashmaps

We discovered that in the original algorithm, when the hash map runs out of space, it frees the old space and reallocates a larger storage area. All the keys stored in the old map must be rehashed and inserted into the new map based on their hash values. Therefore, the more keys there are, the more frequently the space needs to be expanded, resulting in an increased number of rehashing operations. Additionally, with each rehash, the number of keys that need to be rehashed also increases.

$$\text{Number of Hash} = \frac{Initial\_Capacity + n}{2} \times \frac{n}{Initial\_Capacity}$$

$$\text{Time Complexity} = O(n^2)$$

To avoid frequent reallocations and rehashing, we categorize the keys based on their first three characters and build the hash map for one category at a time. This reduces the size of each hash map to 1/4 of the original. After building each map, we clear its contents but maintain the capacity size. As long as the number of keys does not exceed the previous map’s total, no additional memory expansion is needed, and each key only needs to be rehashed once.

$$\text{Number of Hash} \cong \frac{Initial\_Capacity + \frac{n}{4}}{2} \times \frac{\frac{n}{4}}{Initial\_Capacity} + \frac{3}{4}n$$

$$\text{Number of traverse transcript sequence} = 64\text{n}$$

$$\text{Time Complexity} = O(\frac{n^2}{4})$$

Since the hash map's memory is reused 64 times, each time we build a hash map for a specific key combination, it must be written to `hash-map.txt` before creating the next one. Indexing and file writing are performed alternately. According to the final project specifications, the time spent on writing to the file is excluded from the simulation time calculation.

Based on the above design, we made some modifications to `hash-map.h`, `gem5_utils.h`, and `index.c`.

hash-map.h
We added a function called `clean_map`, which clears the contents of the old hash map without affecting its capacity. This allows the next key combination to build a new map without having to expand the capacity from scratch.

gem5_utils.h
To exclude file writing time from the simulation time, we implemented a function called `print_gem5_simulated_time_part`, which reads the simulation time of the first 66 runs from `m5out/stats.txt` and outputs the total to the terminal.

index.c
To interleave file writing and indexing while excluding the time spent on file writing, we made several small modifications. A new function `generate_index_small` was created to build the hash map for each key combination. The `generate_index` function is responsible for controlling which key combination's hash map to generate and when to write to the file. During file writing, we temporarily pause the collection of simulation statistics.Therefore, `m5out/stats.txt` will contain a total of 66 segments of simulation data: 64 for generating each hash map, one for entering the `generate_index` function from `main`, and one final segment for clearing the last hash map.

Optimization: Change the hash algorithm

We experimented with various hash algorithms, including FNV-1a, MurmurHash3, MurmurHash2, and splitmix64.

The effectiveness of a hash function can be evaluated from two aspects: speed and collision rate. When the collision rate is high, the time required to insert keys into the map and access them increases, since it requires linearly searching through all entries with the same hash value to find the correct key.

FNV-1a, and the MurmurHash series offer better collision resistance compared to the originally used DJB hash, but they are computationally more complex. In our case, with `kmer = 15`, each key combination has 4,095 possible strings, the frequency of hash collisions was not very high, so we focused on simplifying the computation.

Ultimately, we adopted splitmix64, which reduced the simulation time by **5.5%**. There are modifications that we must apply on kmer in order to adopt splitmix64, which will be explained in the Quantification part.

	DJB	FNV-1a	Murmur2	Tiny FNV-1a	splitmix64
collision times	120~150	30~40	10~15	70~90	> 1

collision under 10000 hash

Optimization: Change the capacity size

The method used to expand the capacity when storage runs out affects the number of required rehashes. If the capacity increases by a fixed amount each time, the time complexity approaches *O(n²)*. By changing the capacity growth strategy to doubling,the time complexity can be reduced to *O(n log n)*. By changing the way capacity is increased, the simulation time was reduced by **20%**.

$$\text{Number of Hash} \cong \frac{Initial_{Capacity} + \log_2 \frac{n}{4}}{2} \times \frac{\frac{n}{4}}{Initial_{Capacity}} + \frac{3}{4}n$$

$$\text{Number of traverse transcript sequence} = 4\text{n}$$

$$\text{Time Complexity} = \text{O}(n \log n)$$

Optimization: Number of hashmaps

As mentioned earlier, we fixed the first 1, 2, 3, and 5 characters of the keys to divide the hash map into 4, 16, 64, and 256 smaller hash maps, respectively. Through experimentation, we found that dividing into 4 smaller hash maps yielded the best performance, reducing simulation time by **97.5%**.

Additionally, we tried two approaches for generating character combinations: using a recursive function, and directly creating an `array[4][1]` to store the combinations and build the hash maps accordingly. The array-based method was more efficient, but when the number of fixed characters exceeds 1 (let's say it's `n`), the array requires iterating over all transcripts `2^n` times. This diminishes the benefit of building separate hash maps.

Performance

As in the baseline, we tested the indexing results using 10 transcripts and normalized the simulation results by dividing them by the proportion of characters from those 10 transcripts relative to the total number of characters across all transcripts. I ran simulations using transcripts generated by both the original `distribute` function and the improved version of the `distribute` function. The results are as follows:

	# char of 10 transcripts	simulation time	total # char	simulation time	compare to baseline
original distribute	5953	0.028379	2703168	12.88648	10722% faster

modified distribute	762	0.002832	2041772	7.588318	18209% faster

Quantification

Feature: Encoding kmers from string to `uint64_t`

To improve the efficiency of hash map lookups, we replaced string keys with 64-bit unsigned integers. This change eliminates the need for expensive string comparisons (i.e., `strcmp`) during key matching. Instead, we can simply use integer comparisons (`==`) to locate the corresponding entries in the hash map, resulting in faster key matching and reduced lookup overhead.

Based on the above design, we made some modifications to `hash-map.h`, and `quantify.c`.

hash-map.h
We introduced two types of map structures: `SubMap` and `TreeMap`. `SubMap` is a simpler implementation designed for handling small k-mer lengths, while `TreeMap` is an extended version of `SubMap` tailored for larger k values. We reimplemented all the functions originally used in the default hash map for both `SubMap` and `TreeMap`, removing all string operations to improve performance. In addition, we designed a custom encoding method to convert k-mers into `uint64_t` keys and adopted a new hash function to further enhance lookup speed.

quantify.c
We first check the value of `k` to decide whether to use `SubMap` or `TreeMap` for building the hash map and performing lookups. Furthermore, in the `quantify` function, we removed redundant operations to streamline the execution process. We also implemented an early stopping mechanism during the k-mer matching phase. When the value of `matched_transcripts_counts[j]` becomes sufficiently large, we can directly determine the best-matching transcript `T` without scanning all k-mers in the query, thus reducing unnecessary computation and improving overall efficiency.

Optimization: Encoding k-mers into `uint64_t`

Given that RNA sequences consist of only four nucleotides: A, C, G, and T, each character can be efficiently encoded using 2 bits. Therefore, a k-mer can be represented as a 64-bit unsigned integer (`uint64_t`) if its length is within 32 bases (since 64 ÷ 2 = 32). However, to avoid potential issues with the most significant bit being misinterpreted as a sign bit—leading to unintended overflows—we conservatively limit the maximum supported k-mer length to 31.

To support this efficient representation and avoid expensive string operations, we designed a new hashmap structure called `SubMap`.

1. Improvement: Rolling Encoding

In our initial implementation, although lookups were faster than the original string-based version, overall performance was suboptimal. We suspected that the encoding phase was the bottleneck.

Upon analysis, we observed that consecutive k-mers in a query overlap by k−1 characters. Based on this insight, we implemented a rolling encoding technique. Instead of encoding each k-mer from scratch, we reuse the previously encoded value: we left-shift the `uint64_t` representation by 2 bits (dropping the first character), clear the lowest 2 bits, and then bitwise-OR the encoding of the new character. This allows us to generate the next k-mer code in constant time with minimal computation, greatly improving overall efficiency.

2. Improvement: Testing Different Hash Functions

Since we switched from string keys to `uint64_t`, we also needed to replace the default string-based hash function with one optimized for integers. We tested several well-known integer hash functions, including:

- MurmurHash3 64-bit finalizer
- Thomas Wang's 64-bit Integer Hash
- splitmix64

We first evaluated their hash distributions and observed similar collision behaviors among the three. However, when testing actual simulation runtime, splitmix64 significantly outperformed the others in execution time. Based on this result, we selected **splitmix64** as our final hash function due to its superior runtime performance.

T = 10, k = 15:

	total buckets	used buckets	load factor	total collisions	max chain length	simulation time (sec.)
Murmur Hash3	8192	2995	0.45	706	4	0.145188
Thomas Wang		2968		733	4	0.143743
splitmix64		2972		729	5	**0.036013**

Why SplitMix64 Runs Faster than Other Hash Functions?

One key reason why `splitmix64` runs significantly faster than MurmurHash3 and Thomas Wang's hash function is the difference in **instruction count** and **data dependency**.

MurmurHash3 consists of three rounds of operations, each including an XOR, a shift, and a 64-bit multiplication. While these operations are individually fast, each one

depends on the result of the previous step, which creates a **tight data dependency chain**. As a result, the CPU must wait for one operation to complete before it can start the next, limiting instruction-level parallelism.

Similarly, Thomas Wang's hash function involves around six to seven steps of bitwise shifts, additions, and XORs. Even though the operations are simpler, they also suffer from **data hazards**—each line reuses the result from the previous computation—again forcing the CPU to execute them sequentially.

In contrast, `splitmix64` only uses about **four instructions**, and although each step technically builds on the result of the previous one, the design is **much more CPU-friendly**. The constants used are fixed, and the combination of bit shifts, XORs, and multiplications can be **aggressively optimized by modern CPUs**. Since the operations are simple and their dependencies are loose, the CPU can **schedule multiple instructions simultaneously**, reducing stalls and improving pipeline utilization.

This architectural advantage—**fewer instructions with lower dependency between them**—is the core reason why `splitmix64` outperforms the others in runtime performance.

3. Improvement: Supporting Arbitrarily Long k-mers

As mentioned earlier, using `uint64_t` limits the maximum k-mer length to 31. To support arbitrarily large values of `k`, we designed a new multi-level hashmap structure called **`TreeMap`**.

When `k > 31`, the k-mer is divided into `⌈k / 31⌉` segments, and each segment is encoded into a `uint64_t` using the same 2-bit representation. These encoded segments form an array that serves as a path through the layers of the `TreeMap`. Each level of the tree corresponds to a segment of the k-mer. The final layer contains the actual value associated with the full k-mer.

Structure Comparison

- **SubMap:**
  `MasterEntry(uint64_t encoded_kmer, value)`
- **TreeMap:**
  `TreeEntry(encoded_kmer, pointer to next TreeMap layer)`
  for last layer: `TreeEntry(encoded_kmer, value)`

Performance Evaluation of TreeMap

Despite the increased complexity required to support long k-mers, the `TreeMap` structure maintains efficient performance. Our experiments show that even with larger values of `k`, the simulation time remains within 0.04 - 0.05 seconds.

T = 10, R = 390, Using `default_distribute` and `default_index`:

k-mer Length (k)	Simulation Time (sec)
15	0.036013
40	0.046560
100	0.044062

Optimization: Early Stopping in Query Matching

To reduce redundant comparisons during the query-to-transcript matching phase, we implemented an **early stopping** mechanism. For each query, we maintain two variables: `max_val` and `sec_val`, representing the highest and second-highest values in the `matched_transcripts_counts` array, respectively.

As we iterate through each k-mer in the query, we update both `max_val` and `sec_val` accordingly. At any point, if the difference between them exceeds the number of remaining k-mers yet to be checked, i.e.,

(max_val - sec_val) > (# of unchecked k-mers),

then it becomes statistically impossible for any other transcript to catch up. In such cases, we terminate the scanning early and take the current `max_val` as the result for that query.

Although maintaining both `max_val` and `sec_val` introduces a slight computational overhead, empirical results show that early stopping is triggered in over **80%** of the queries. This significantly reduces the number of unnecessary comparisons, resulting in a modest but consistent improvement in runtime performance. Specifically, simulation time was reduced from **0.193394 seconds** (baseline `quantify_default`) to **0.187203 seconds** (with early stopping only).

In addition to the reduced simulation time, cache efficiency also improved. By checking the `stats.txt` output, we observed that the number of L1 data cache demand misses (`system.cpu.dcache.demand_misses::total`) decreased significantly—from **212,538** in the default implementation to **115,927** with early stopping enabled. This indicates that the early termination not only reduces computation but also improves memory access patterns, contributing further to performance gains.

Moreover, since we already track `max_val` during the matching process, we were able to eliminate the additional step in the original `quantify_default` implementation that involved scanning the entire `matched_transcripts_counts` array again to find the transcript with the maximum count.

Optimization: Eliminating All IntArray Usage

In the default implementation, `IntArray` was used to store dynamic lists. However, each time a new element is inserted into an `IntArray`, a `realloc` operation is triggered to expand the array. This introduces unnecessary memory management overhead and potential performance bottlenecks, particularly when processing a large number of queries or transcripts.

To address this, we refactored the code to completely eliminate the use of `IntArray`, thereby simplifying execution and improving memory efficiency.

First, in the `get_values_SubMap` function (which retrieves the values associated with a k-mer), we directly update the `matched_transcripts_counts` array. This allows us to skip the original two-step process where the lookup result was first copied into an `IntArray` (`matched_transcripts_idxs`), and then used to increment the corresponding entries in `matched_transcripts_counts`.

Additionally, in the final stage of the `quantify_default` implementation, the original design used `matched_transcripts_counts` and `max_count_for_a_transcript` to populate an intermediate `IntArray` (`transcript_idxs_with_maximum_count`), and then used that array to update the `final_results`. We identified this as redundant. We can directly update the `final_results` array without building any intermediate structure.

This optimization not only reduces the number of memory allocations and pointer dereferences, but also simplifies the control flow, making the code more efficient and maintainable.

Performance

I evaluated the performance of `quantify_default` and `quantify` under two experimental setups: (`distribute_default + index_default`) and (`distribute + index`). The results are as follows:

	# char of 10 transcripts	quantify_default simulation time	quantify simulation time	compare to baseline
distribute_default index_default	5953	0.246990 sec.	0.036013 sec.	686% faster
distribute index	762	0.138387 sec.	0.029263 sec.	473% faster

Bonus

performance and area tradeoff discussion

In this section, performance and area tradeoff will be discussed. Performance is defined as

$$\text{Performance} = \frac{\text{Throughput}}{\text{Area}} = \frac{\frac{\text{Number of query RNA sequences}}{\text{Time}}}{\text{Area}} = \frac{\text{Number of query RNA sequences}}{\text{Time} \times \text{Area}}$$

Note that since baseline simulation works only on a CPU, there is **no distribution stage** involved.

Area

Based on the reference and document provided, the Intel(R) Xeon(R) CPU E5-2687W v4 has an area of **456 mm^2,** while a single logic occupies **0.161 mm^2.** In this assignment, We simulate on 256 logic units, resulting in a total area of 256 × 0.161 = **41.216 mm^2** is required, including the Network on Chip(Noc) overhead.

Comparison

We compare the performance and area tradeoff between the following two cases

case 1: CPU without Logic Units (based on the provided document)

a. Indexing

The CPU indexes 252913 transcripts from `gencode.v43.transcripts.non.fa`, taking 666.53 seconds.

- Throughput = 252913 / 666.53 ≈ 379.47 transcripts/sec
- Performance = 379.47 / 456 ≈ **0.8322 transcripts/mm^2·sec**

b. Quantifing

100,000 queries from `queries.10K.fa` has a processing time of 3.02 seconds.

- Throughput = 100000 / 3.02 ≈ 33112.58 queries/sec
- Performance = 33112.58 / 456 ≈ **72.6153 queries/mm^2·sec**

case 2: 256 logic units with SEDRAM Architecture

The system uses **parallel logic units**, we simulate only on the heaviest workload logic units since it will be the bottleneck of the entire system. The final throughput is derived by assuming that all 256 logic units run in parallel and complete after the slowest logic unit finishes.

a. Indexing

The processing time on the heaviest workload logic unit with 988 transcripts is 8.274 seconds, which means throughput on a single logic unit is 119.41transcripts per second. Assuming that the RNA sequences are evenly distributed across all logic units, we have:

- Throughput = 119.41 * 256 = 30569.01 transcripts/sec
- Performance = 30569.01 / 41.216 = **741.678 transcripts/mm^2·sec**

b. Quantifing

391 queries on the heaviest workload has a processing time of 0.029263 seconds.The throughput on a single logic unit is 13361.583, we have:

- Throughput = 13361.583 * 256 = 3420565 queries/sec
- Performance = 3420565 / 41.216 = **82991.2 queries/mm^2·sec**

		Baseline	256 logic units	speed up
Index	Throughput	379.47 transcripts/sec	30,569.01 transcripts/sec	80.56
	Area	456 mm^2	41.216 mm^2	
	Performance	0.8322 transcripts/mm^2·sec	741.678 transcripts/mm^2·sec	891
Quantify	Throughput	33112.58 transcripts/sec	3420565 transcripts/sec	103
	Area	456 mm^2	41.216 mm^2	
	Performance	72.6153 queries/mm^2·sec	82991.2 queries/mm^2·sec	1143

Summary

As shown in the table, the SEDRAM-based implementation using 256 Logic Units achieves over 891x performance improvement in indexing and over 1143x in quantification, while occupying only ~9% of the CPU's area, demonstrating a highly efficient design for SEDRAM when dealing with a large number of small, computation-intensive tasks—such as RNA sequence indexing and quantification or matrix operations.