# GPUMusket: A Parallelized GPU-Accelerated Pre-Processing Algorithm on Substitution-Based Errors for Short-Read Sequences in De Novo Genome Assembly

Gio Gonzales
Department of Computer Science, College of Computing and Information Sciences, Caraga State University

Mary Abigail Soliva
Department of Computer Science, College of Computing and Information Sciences, Caraga State University

Rudolph Joshua Candare
Department of Computer Science, College of Computing and Information Sciences, Caraga State University

https://doi.org/10.5109/7395506

出版情報：Proceedings of International Exchange and Innovation Conference on Engineering & Sciences (IEICES). 11, pp.114-122, 2025-10-30. International Exchange and Innovation Conference on Engineering & Sciences
バージョン：

# GPUMusket: A Parallelized GPU-Accelerated Pre-Processing Algorithm on Substitution-Based Errors for Short-Read Sequences in De Novo Genome Assembly

Gio Gonzales[1], Mary Abigail Soliva[1] , Rudolph Joshua Candare[1]
[1] Department of Computer Science, College of Computing and Information Sciences,
Caraga State University – Main Campus, Butuan, Philippines
Corresponding author email: gio610gonzales@gmail.com

**Abstract:** *GPU parallelization offers a promising solution to the scalability and speed limitations of CPU-based genomic error correction—but can it maintain accuracy while overcoming memory bottlenecks? This research presents a high-performance GPU-accelerated tool (via Python Ray and Rust) built on Musket's multi-stage framework. Benchmarks reveal that GPUMusket significantly improves processing speed and scalability compared to CPU-based Musket, albeit with higher memory usage (though still lower than SMusket). Error correction results show increased true positives but also higher false positives, while small genomes exhibit more false negatives versus Musket's precision-focused approach. However, GPUMusket achieves >98.7% sensitivity on simulated data and excels at low coverages. For de novo assembly, both tools perform comparably on simple genomes, though Musket retains a slight edge in complex cases. By balancing GPU-driven efficiency with reliable accuracy, GPUMusket emerges as a practical alternative for bioinformatics pipelines prioritizing rapid preprocessing without sacrificing robustness—a critical advance for large-scale genomic analyses.*

**Keywords:** GPU-processing; parallelism; error-correction; genome; bioinformatics.

## 1. INTRODUCTION

The prerequisite for studying the evolutionary development of any organism requires an accurate and complete genome [1]. Genome assembly is performed to create a representation of the original chromosomes merged from a considerable number of short DNA sequences [2]. One of the most popular yet challenging techniques of genome assembly, called de novo assembly, can assemble short reads from organisms without requiring any reference genome [1]. However, as more organisms' genomes are sequenced and assembled, this assembly method has grown increasingly challenging and computationally demanding, mainly when dealing with short, erroneous sequence reads and repetitive repeats [3].

Recently, short-read de novo genome assemblers have been demonstrated to provide higher throughput, accuracy, and cost-effectiveness [4][22] compared to long-read sequencing, which has higher error rates and incurred costs. Additionally, assembly times of Illumina-type short reads (averaging between 150bp to 600bp) may vary depending on several factors, such as genome size, read length, assembly algorithm, and the processing equipment utilized [5]. Hence, the assembly of small genomes like bacteria can range from minutes to hours, while large and complex genomes like plants and homo sapiens can span between days or 2 weeks due to their high repeat content [6]. Despite its advantages, it can be tedious to generate reliable contiguous sequences due to errors in sequencing, which occur more in low-complexity regions or DNA stretches containing biased patterns and repetitive GC or AT base pair (bp) contents [7]. Moreover, sequencing errors typically lead to complex graphs necessary for the contiguous sequence assembly [6]. Consequently, pre-assembly data cleaning (or preprocessing) has emerged as a critical step in de novo assembly from NGS reads to mitigate assembly quality issues. Addressing the problem in short read sequences, algorithms for removing errors have been proposed for genomic sequencing data, which attempts to rectify them by changing individual bases in reads while keeping the original quality scores [8]. A commonly used method referenced from the short-read assembly pipeline is known as the "correction then assembly" (CTA) strategy, which can produce more accurate and continuous assemblies because of its ability to distinguish segmental duplications, especially for large plant genome assemblies [1]. Therefore, error correction in sequenced reads is crucial due to its positive effects on ensuring applications' quality and run-time/memory efficiency.

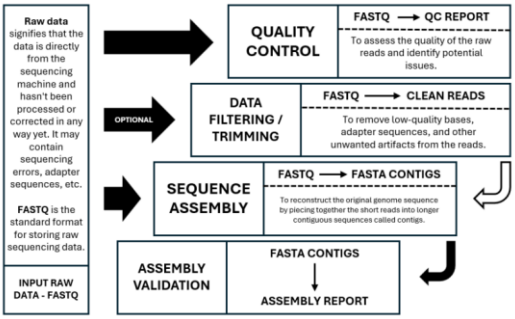


Fig. 1. General Genome Assembly Workflow [16]

According to performance evaluation results by [9], Musket, a CPU-based tool, ranks among the top correctors regarding de novo genome assembly criterion and correction quality due to its multistage workflow and k-mer spectrum approach using a master-slave model. However, the computational cost of the error correction phase is often neglected in the de novo assembly, and high memory requirements and excessive CPU hours can significantly increase the overall cost [10]. Additionally, its distributed alternative improves throughput in multi-

node clusters but remains inefficient in single-node environments [11].

Much of the field of de novo assembly still relies on CPU-based solutions [12]. Although graphics processing units (GPUs) have become a growing interest in bioinformatics, state-of-the-art algorithms have not exploited the parallel processing power of GPUs [13] not because they lack potential, but due to algorithmic mismatches, memory limits, and entrenched CPU toolchains [14]. Therefore, much of its advancements with de novo assembly have only been applied to overlap detection and layout simplification [15], as well as read mapping and alignment [13]. With leveraging the collective power of multiple GPUs in error correction, it poses a significant advantage in genome recovery and assembly by balancing the workload to increase efficiency in processing times for varying genome sizes and quality of outcomes. Here, we present a parallelized GPU-accelerated algorithm designed to address the limitations of CPU-bound correctors by restructuring Musket's multistage k-mer workflow for GPU execution. This study seeks to establish a parallelized and efficient substitution-based, pre-processing enhancement for the Musket multi-stage framework through GPU acceleration. This research aims to contribute to advancements in error correction tools for de novo genome assembly in bioinformatics. The specific objectives of this study are as follows:

- Develop a parallelized error correction algorithm for short read de novo genome assembly, leveraging the Musket error correction algorithm's multi-stage workflow and the power of GPUs.
- Evaluate the performance improvement of the GPU-powered error correction method compared to the existing, open source, and multithreading-based parallel implementations on error correction like Musket and SMusket, prioritizing metrics like processing speed and memory usage and its true positive and false positive rates.
- Assess the impact of the developed parallel error correction algorithm on the overall quality of the assembled genomes (of assorted sizes) to a specific de novo assembler using evaluation metrics such as assembly accuracy metrics like N50 value, contig number, and overall error rate.

## 2. METHODS

This study introduces a parallel error correction framework for short-read sequencing data, employing a master-worker architecture accelerated by GPU processing. The methodology leverages Python Ray to manage GPU resources, with each GPU assigned to a separate Ray actor for distributed task execution. To optimize read extraction from FASTQ files—a performance bottleneck in Python—the system integrates a Rust-based module for efficient parsing. The input dataset is partitioned into equal chunks, distributed across actors, and processed independently. Each actor extracts k-mers, computes offsets, and converts reads into a 2D numerical array using CUDA-optimized kernels.

The k-mers from all actors are aggregated into a global spectrum, where canonical k-mers are identified, and low-frequency k-mers below a dynamically calculated threshold are filtered out. The refined global spectrum is then redistributed to the actors for the correction phase, where CUDA kernels perform multistage error correction on local reads. Finally, corrected reads are merged in the driver program, output to a FASTQ file, and assessed via de novo assembly, with optional read trimming for comparative evaluation. This approach ensures high-throughput processing while maintaining accuracy for downstream genomic analysis.

### 2.1 Implementation of Parallel Technique

This approach utilizes a master-slave architecture like its original implementation [13] to handle large datasets efficiently by distributing the workload across multiple computing processing elements equipped with GPU resources. Each slave processing element processes a subset of the data independently, while the master processing element consolidates the results to form a globally distributed memory structure. This structure is the backbone for constructing the k-mer spectrum, identifying erroneous sequences, and performing the multi-stage error correction process.

#### 2.1.1 Construction of Valid K-Mers

The generation of valid k-mers is a foundational step in sequence analysis, where sequencing reads are split into overlapping subsequences of fixed length *k*. This process is accelerated using cuDF, a GPU-optimized DataFrame library from the RAPIDS ecosystem, which exploits massive parallelism to handle large genomic datasets efficiently [23]. Each read is processed independently across GPU threads, with k-mers extracted via cuDF's *character_ngrams* method, systematically generating all possible k-length subsequences. To minimize memory usage, the k-mers—initially stored as strings—are converted into a compact *uint64* numerical format using *astype*. After extraction, identical k-mers are aggregated, and their frequencies are computed using functions like *count* or *sum*, distinguishing high-frequency (biologically relevant) k-mers from low-frequency (potentially erroneous) ones. The *value_counts* method tallies occurrences, filtering out unique k-mers (likely sequencing artifacts) and applying a dynamic cutoff threshold to retain only statistically significant k-mers. The resulting k-mer spectrum is then transferred to GPU memory for downstream analysis, ensuring rapid and scalable processing. By leveraging GPU parallelism and optimized data structures, this method dramatically accelerates k-mer construction while preserving accuracy in large-scale genomic workflows.

#### 2.1.2 GPU Data Preparation

Using the library of Numba CUDA, it is essential to recognize that directly manipulating individual bases or string data types presents significant challenges when working with genomic data. This is primarily due to the limitations of GPU architecture, which is optimized for parallel processing and numerical computing rather than complex string operations. To overcome these limitations, each nucleotide base is transformed into numerical representations, allowing for more efficient computations as it streamlines data manipulation and effectively handles potential uncertainties in sequencing data. Once the bases are encoded, we can structure the dataset shaping it into a two-dimensional dataset where each row represents an individual read. This reduces the overhead and leverages contiguous memory access patterns favored by GPU architecture.

#### 2.1.3 Cutoff Threshold Calculation

A cutoff threshold in k-mer extraction is a specific multiplicity value that acts as a dividing line between trusted k-mers and erroneous ones. Its use ensures that only high-quality, biologically meaningful k-mers contribute to the analyses and that the computational complexity of subsequent steps is reduced as the data size is effectively filtered. This calculation is usually represented by a multiplicity histogram, which usually has a valley (a dip in the curve before the region of higher multiplicity begins), used as a boundary marker separating two distinct regions: left of the valley (represents untrusted k-mers with low occurrence count) and right of the valley (k-mers with high occurrence reflecting actual biological sequences). To determine the cutoff value, the point of lowest density in the valley must be identified as the optimal threshold for cutoff because it minimizes the overlap between genuine and spurious k-mers.

#### 2.1.4 Calculation of Canonical K-Mers

K-mer or substrings of fixed length k is a standard method in handling varying genome lengths to efficiently index or compare sequences [18]. Storing kmers directly as strings can be inefficient; therefore, a translation method converts these string sequences into integer values to facilitate faster data handling and reduced storage requirements [19]. Once k-mers are converted into integers, they can be stored in data structures known as arrays. In parallel processing, this translation method can allow the easier distribution of the workload across multiple threads and computing nodes, enabling simultaneous processing to speed up the analysis.

However, in most cases, genomes are not given as complete sequences but rather in reads or contigs, making it difficult to determine their reading direction because this might represent either the original sequence or its reverse complement. To address this ambiguity, the method assumes that both the k-mer and its reverse complement are equivalent, and a canonical representative is chosen for each pair [20]. This unique representative is the lexicographically smaller of the two k-mers, which means it is the one that would appear first in the dictionary or alphabetical order. This standardization ensures consistency in data processing and redundancy.

For odd-length k-mers (<= 3), each k-mer can be paired with its reverse complement such that these opposites will pair to form one canonical k-mer [18]. As a result, the number of canonical k-mers is half the total number of k-mers. On the other hand, even length k-mers (<= 4) can include k-mers that are palindromes which are considered canonical due to the non-existence of a unique reverse complement; thus, they stand alone in the count of canonical k-mers [21]. They increase the total of canonical k-mers, making it more than just half of the total k-mers.

### 2.2 Error Correction Multi-Stage

The process integrates multiple complementary techniques, each targeting specific challenges associated with sequencing data. Utilizing Python Ray, an error correction task is first defined, and one GPU resource is allocated for its execution. To enable efficient GPU-based processing, we use Numba to define a GPU kernel as the entry point for executing one-sided correction, along with the GPU device functions that serve as utility functions for this process.

Consequently, several GPU threads are allocated, ensuring that the number of allocated threads is greater than or equal to the number of reads in the dataset. This allocation is determined by computing the threads per block and blocks per grid. The selected number of threads per block must be a multiple of 32, as GPU kernels issue instructions in warps consisting of 32 threads. This ensures optimal parallel execution by fully utilizing the warp-based architecture of modern GPUs. Each read in the dataset is assigned to a single GPU thread. To optimize memory access, global memory-stored reads are transferred to each thread's local memory since querying global memory frequently is computationally expensive. This memory transfer significantly enhances performance by reducing memory latency. In addition, the same multi-stage error correction step will be performed iteratively on each individual dataset by a minimum of (>=5) to determine if there are inconsistencies in accuracy metrics after benchmarking.

About the multi-stage error correction workflow, GPUMusket follows the same algorithm introduced by Musket [17], but is reimplemented using the Python programming language, taking into consideration the constraints in a GPU context.

The workflow consists of three key steps for error correction. First, a two-sided conservative correction evaluates both leftmost and rightmost k-mers covering each position *i*, iteratively identifying a unique alternative base that ensures all overlapping k-mers are trusted (default: 2 iterations). Second, a one-sided aggressive correction applies stricter changes while limiting corrections per k-mer (default: 4) to prevent false positives. However, one-sided correction carries the risk of propagating errors if a mistakenly "trusted" k-mer leads to incorrect assumptions. To address this, look-ahead validation is implemented as an additional safeguard. This step evaluates a predefined number of neighboring k-mers (default: 4) to confirm the validity of the proposed base change, ensuring corrections are consistent across nearby sequences and reducing the likelihood of error propagation. Further reinforcing the correction process, the Correction Constraint Tracker monitors and restricts the number of corrections allowed per k-mer. By imposing a strict threshold, this mechanism prevents overcorrection in ambiguous regions, where excessive modifications could introduce new errors rather than resolve existing ones. Finally, a voting-based refinement step validates corrections by aggregating weighted votes from overlapping k-mers, selecting the highest-voted unique base alternative to maximize accuracy while minimizing new errors.

### 2.3 Error Correction Quality Assessment

This section comprehensively analyses the effectiveness, efficiency, and resource utilization of the error correction method applied in a GPU-powered environment. By evaluating multiple aspects of performance, such as correction accuracy, computational resource consumption, and comparison with existing tools like Musket and SMusket, the researchers aim to highlight the proposed approach's strengths and areas for improvement. Each subsection addresses a unique aspect of the

evaluation to provide a holistic view of the error correction process.

2.3.1 Evaluation Metrics for Error Correction Accuracy

The evaluation of error correction algorithms in sequencing data relies on key metrics—precision, sensitivity, and gain—to assess their ability to accurately identify and rectify errors while avoiding false corrections. These metrics are derived from a confusion matrix, which classifies algorithmic outcomes into four components: True Positives (TP) represent erroneous bases correctly corrected to their true values; False Positives (FP) denote correct bases wrongly altered, introducing new errors; True Negatives (TN) reflect correctly unchanged bases (though TN is often excluded from precision and sensitivity calculations); and False Negatives (FN) indicate uncorrected errors or mis-corrections. Together, these metrics quantify the algorithm's reliability in error detection and correction while minimizing unintended modifications.

$$\frac{TP}{TP+FP+FP_{INDEL}},\ TP+FP+FP_{INDEL}>0 \quad (1)$$

$$\frac{TP}{TP+FN},\ TP+FP>0 \quad (2)$$

$$\frac{TP-(FP+FP_{INDEL})}{TP+FN},\ TP+FN>0 \quad (3)$$

The equations above show the formulas of the metrics utilized to calculate the Precision (1), Sensitivity (2) and Gain (3). Equation (1) measures the proportion of proper corrections among the total number of performed corrections. High precision means that the algorithm makes a few incorrect changes, while low precision indicates that the algorithm introduced many errors or incorrect corrections. While (2) evaluates the proportion of fixed errors among all existing errors in the data, high Sensitivity indicates that most real errors are corrected (low FP count), and low Sensitivity means many errors remain uncorrected. Lastly, (3) measures the net benefit of the correction algorithm by comparing accurate corrections (TP) against introduced errors (FP), normalized by the total existing errors (TP + FN). It represents whether an algorithm produces an overall benefit (more TP than FP) or has an adverse effect (more FP than TP). Values ranging from 1.0 to, but not including, 0.0 represent a benefit; 0.0 is neutral, and less than 0.0 is considered an adverse effect.

2.3.2 Memory and Speedup

The researchers would also like to investigate the runtime performance of GPU-based error correction algorithms compared to traditional CPU-based implementations, a critical metric for evaluating efficiency, particularly for large datasets. This analysis highlights the scalability and practical benefits of integrating GPU resources into bioinformatics workflows by demonstrating the potential reduction in processing time. To calculate the empirical speedup, the formula is as follows:

$$Speedup = \frac{Reference\ Time\ (Musket\ or\ SMusket)}{Reference\ Time\ (GPUMusket)} \quad (4)$$

Furthermore, the study will evaluate the peak resident memory usage during error correction, comparing it with other tools like Musket and SMusket, as memory footprint significantly impacts usability, especially when dealing with large-scale datasets.

### 2.4 Application to De Novo Assembly

This phase evaluates the error-corrected datasets to a chosen de novo assembler with the ability to disallow automatic trimming to determine the assembly accuracy of the preprocessing mechanism effectively. Trimming is also an optional preprocessing step that happens after error correction, which involves identifying, correcting, and removing sequencing errors or low-quality bases (e.g. base miscalls, insertions, deletions, bases with PHRED score quality < Q20) to reduce mis assemblies further and improve continuity. Metrics such as contig N50, genome fraction (completeness), mis assembly rates, and base accuracy will be explored to quantify the impact of the GPU-powered error correction algorithm on assembly outcomes. These metrics provide a comprehensive understanding of how improved read quality translates into enhanced assembly accuracy and continuity, making it possible to evaluate the downstream effectiveness of the error correction process.

## 3. RESULTS

This section examines the analysis and results of the GPU-accelerated error correction algorithm for short-read sequencing data. The researchers aimed to evaluate the algorithm's performance across various datasets by benchmarking it against traditional CPU-based error correction tools, including Musket. Finally, this chapter assesses the algorithm's quality and performance, including evaluations on real and simulated datasets, runtime comparisons between GPU and CPU implementations, memory usage metrics, and whether there is a significant improvement in error reduction against other error correction tools. The application of the corrected reads to de novo assemblers is also explored, demonstrating the broader utility of the algorithm beyond genome pre-processing.

In the testing environment, Python Ray orchestrates task distribution and scheduling across GPU workers in the cluster, managing both k-mer extraction and error correction processes. The system operates on a single node equipped with three NVIDIA Tesla V100 GPUs, each with 32GB of VRAM, alongside 128GB of host RAM and an Intel Xeon Gold 6246 CPU featuring 48 cores to support high-throughput parallel processing. Ray efficiently allocates computational tasks across the GPUs while optimizing workload distribution among workers. A key component of this setup is the object store, a shared memory space designed to facilitate efficient data transfer between tasks and workers. This store allows up to 63 objects to reside in memory outside the Python Heap, minimizing serialization and deserialization overhead. By leveraging this architecture, the system ensures rapid access to large datasets—such as k-mer spectra—during processing, enhancing overall performance in genomic analysis workflows.

### 3.1 Dataset Classification

The datasets used in this study were gathered from the online database of the National Center for Biotechnology Information—Sequence Read Archive (NCBI-SRA), an open-source biomedical repository that stores real sequencing data, such as reads with different types of

classifications generated by high-throughput sequencing technologies. The researchers mainly evaluated the effectiveness of the error correction algorithm on selected genome sizes to ensure meaningful performance comparisons in terms of runtime and memory consumption. Since error correction tools are frequently applied to large-scale sequencing projects, the study focused on datasets with accession number's *ERR022075 (E.Coli), SRR002291 (Human Dataset 1), and SRR19858939 (Human Dataset 2)* that has substantial read counts (ranging from 22.7 million to 78.5 million reads) to better reflect real-world computational demands. Smaller short-read datasets were used to test correction accuracy, such as Influenza.A with accession number SRR28464539. However, they were excluded from evaluating memory usage or runtime scalability because they did not sufficiently provide significant results. These are key factors in assessing tool efficiency but are applied in evaluating the accuracy metrics of the genomic error correction tool. Additionally, simulated datasets are derived from specific datasets *(E.Coli and H.Salinarum SRR10031102)* reference sequence, which serves as a controlled environment where true sequences are known to determine the preciseness of evaluating the error correction process. In this case, the errors introduced are only substitutions with varying rates (e.g., 1%, 2%, and 3%) which are randomly distributed into reads at a specified rate, mimicking real sequencing where errors do not occur at perfectly spaced intervals. Furthermore, the read length of each simulated read is set to 100 base pairs to ensure fair comparison for benchmarking across different error rates.

### 3.2 Evaluation of Error Correction Results

On real datasets, GPUMusket achieves higher true positives (+3.66% to +15.66%) and more false positives (14–19%), with larger genomes like E. coli benefiting most from GPU acceleration. However, its aggressive correction leads to elevated false negatives in smaller genomes (e.g., +61.33% for Influenza A). Musket demonstrates better precision (0.72–1.39% higher) and gain (1.5–3.2% higher) due to its conservative approach. GPUMusket shows more substantial precision for simulated datasets at lower coverages (+4.79–5.70%) but outperforms Musket at higher error rates. Both tools achieve high sensitivity (>98.7%), though Musket maintains a slight edge (3–5%). Notably, Musket exhibits significantly better gain (+8.5–8.6%) in E. coli, highlighting a performance gap. While GPUMusket excels in speed and lower-coverage precision, Musket remains superior in balanced error correction for most scenarios.

Table 1. Results of Precision, Sensitivity, and Gain

| | | Precision | Sensitivity (Recall) | Gain |
|---|---|---|---|---|
| **Influenza A.** | **GPUMusket** | 71.66% | 91.70% | 55.43% |
| | **Musket** | 72.88% | 91.19% | 57.27% |
| **H. Salinarum** | **GPUMusket** | 85.58% | 87.98% | 73.16% |
| | **Musket** | 86.79% | 87.99% | 74.60% |
| **E.Coli** | **GPUMusket** | 70.16% | 97.13% | 55.82% |
| | **Musket** | 70.67% | 96.89% | 56.68% |

### 3.3 Benchmarking GPUMusket with other EC Tools

This section presents a comparative performance analysis of GPUMusket against other EC tools, focusing on runtime speedup and peak resident memory usage across multiple datasets. The evaluation highlights GPUMusket's computational efficiency and resource utilization relative to CPU-based alternatives, assessing whether its GPU-accelerated approach translates into measurable performance gains. While GPUMusket leverages GPU parallelism to accelerate computationally intensive stages (e.g., k-mer frequency analysis), its design faces inherent limitations due to the algorithm's branching logic, which restricts full vectorization. Unlike purely data-parallel EC tools, GPUMusket's multistage workflow relies on conditional operations that introduce warp divergence on GPUs, preventing uniform SIMD optimization. Despite this constraint, GPUMusket strategically offloads scalable workloads to the GPU, while retaining sequential steps on the CPU. Nonetheless, results have demonstrated that GPUMusket achieves significant runtime improvements while maintaining competitive memory efficiency. This balance reinforces its advantages for large-scale genomic error correction, particularly in scenarios where GPU-friendly operations dominate the computational burden. The trade-off between partial vectorization and branching overhead is critically analyzed, providing insights into optimization for future hybrid architectures.

#### 3.3.1 GPUMusket vs. other EC Tools on Real Datasets

The real E.Coli short-read dataset is evaluated with GPUMusket under three hardware configurations on a single-node environment—a single-GPU setup and a multi-GPU setup with 3 GPUs as the highest, where each GPU requires only a single CPU thread. While Musket, the CPU threads vary where 16 threads are the lowest and 48 threads are the highest. SMusket was included as a baseline, tested under the same thread configurations as Musket.

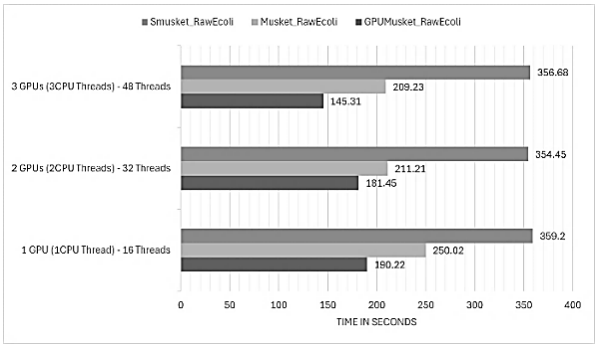


Fig. 2. Comparison of Error Correction Runtime between Musket, SMusket, and GPUMusket for Real E.Coli Dataset

GPUMusket has presented a consistent performance advantage across all tested configurations with performance advantages ranging from 14.1% - 30.6%, but it is most significant with 3 GPUs at 64.92 seconds. Moreover, GPUMusket scales from 190.22 seconds (1 GPU) to 145.31 seconds (3 GPU). Musket shows limited scaling with additional threads from 250.02 seconds (16 threads) to 209.23 seconds (48 threads) with only a 16.3% improvement. Furthermore, SMusket's runtimes

remain consistently high (~335 seconds) regardless of thread count making it slower than Musket by 16.3% - 41.8% and significantly slower for GPUMusket. This indicates that GPUMusket remains faster in absolute terms (145.31 seconds on 3 GPUs vs. 209.23 seconds on 48 threads). GPUMusket appears better optimized for higher GPU counts but it is still particularly well-optimized for single-GPU operations, making it especially valuable for users with limited hardware resources. Even with a single GPU, GPUMusket offers nearly 60 second time savings over Musket and SMusket. These results suggest that GPU acceleration, when properly implemented, can significantly reduce processing times for genomic error correction tasks.

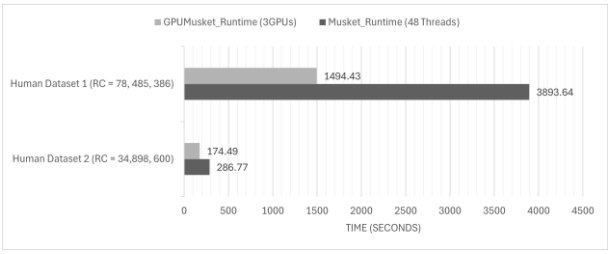


Fig. 3. Comparison of Error Correction Runtime between Musket and GPUMusket for Two (2) Human Datasets

For the runtime performance of two (2) human datasets, dataset 1 (78,485,386 reads) exhibits longer runtimes than Dataset 2 (34,898,600 reads) due to its larger data volume. SMusket resulted with a runtime of 172.43 seconds for human dataset two (2) which is slightly better than GPUMusket (174.49 seconds) and demonstrates significant efficiency gains over Musket—achieving a 52.4% faster runtime (172.43s vs. 286.77s). The most dramatic performance gains occur with GPUMusket (using 3 GPUs + 3 CPU threads). Dataset 1 finishes 61.6% faster than Musket (1,494.43s vs. 3,893.64s), while Dataset 2 achieves a 39.2% reduction (174.49s vs. 286.77s). Compared to SMusket's successful run on Dataset 2, GPUMusket's advantage is narrower but still notable (19.4–20.8% faster).

### 3.3.2 GPUMusket vs. other EC Tools on Simulated Datasets

GPUMusket's Error Correction Runtime on simulated E.Coli datasets effectively leverages GPU parallelism and demonstrates strong scalability, significantly reducing runtime as more GPUs are added compared to its CPU-based counterpart. For the 30x coverage, GPUMusket demonstrates the fastest overall runtime by ~16-22 seconds but has the smallest relative speedup which is likely due to the lower computational demand. However, the tool scales well with additional GPUs particularly for high-coverage, large datasets (70x - 100x), where it achieves ~30% - 36% speedup with 3 GPUs.

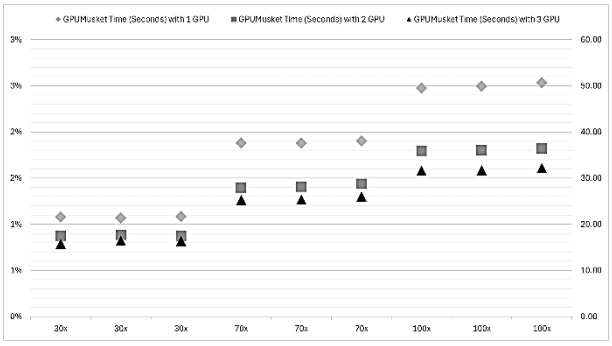


Fig. 4. GPUMusket Error Correction Runtime for Simulated E.Coli Dataset with Different GPU Numbers

GPUMusket also remains consistently efficient with minimal runtime differences across all error rates (1% - 3%) suggesting that it efficiently handles varying error complexities making it a superior choice for high-throughput error correction. In contrast, Musket shows strong dependence on coverage level but limited benefits from increased thread counts where threads beyond 16 show only limited benefit in 30x coverage, while in 100x coverage 32 threads show minimal improvement. Error rate sensitivity is also more impactful of Musket's runtime at 30x coverage by ~17.7% (13.09 seconds to 15.41 seconds) and is less pronounced at higher coverage levels. Furthermore, SMusket in higher coverage had led to longer runtimes, but it has not scaled efficiently with additional threads as runtime improvements were marginal (≤ 5%) and in some cases, its performance slightly degraded. For instance, at 70x coverage with 3% error rate for 16 threads, the runtime is only 58.90 seconds while at 48 threads has slightly increased to 60.64 seconds. This indicates that SMusket does not perform well on small datasets and especially on a single node cluster environment.

### 3.3.3 Runtime Speedup Analysis for Real Datasets

GPUMusket demonstrates superior performance and scalability compared to both Musket and SMusket, with empirical benchmarks confirming consistent speedups across multiple GPU configurations. Against Musket, a well-optimized CPU-based tool, GPUMusket achieves speedups ranging from 1.31x (1 GPU) to 1.44x (3 GPUs), though it exhibits a temporary dip in efficiency at 2 GPUs (1.16x), suggesting diminishing returns before recovering with additional GPU resources.

In contrast, GPUMusket significantly outperforms SMusket, with speedups increasing from 1.89x (1 GPU) to 2.45x (3 GPUs), highlighting SMusket's poor scalability despite its GPU-based implementation. These results indicate that GPUMusket optimizes GPU acceleration more effectively than SMusket while maintaining competitive performance against Musket's CPU efficiency. At 3 GPUs, GPUMusket reaches peak performance, being 1.44x faster than Musket and 2.45x faster than SMusket, establishing it as the preferred choice for GPU-enabled systems.

However, in resource-constrained environments where GPU availability is limited, Musket remains a viable alternative due to its CPU efficiency, whereas SMusket's inefficiencies render it the least favorable option.

Table 2. Speedup Results of GPUMusket vs. other EC Tools on E.Coli Real Dataset

| No. of GPUs to No. of CPU Threads | Speedup (GPUMusket vs. Musket) | Speedup (GPUMusket vs. SMusket) |
|---|---|---|
| *1 GPU (1 CPU Thread) - 16 Threads* | 1.31x | 1.89x |
| *2GPU (2CPU Threads) - 32 Threads* | 1.16x | 1.95x |
| *3GPU (3 CPU Threads) - 48 CPU Threads* | 1.44x | 2.45x |

For the real human datasets, GPUMusket has exhibited superior performance compared to both Musket and SMusket across different dataset sizes of the two (2) human genomes. There are significant speedup improvements for the larger Human dataset (78.5 million reads), achieving 2.61x over Musket, while maintaining a 1.64x advantage for the smaller Human dataset (34.9 million reads). Furthermore, GPUMusket also sustains approximately 1.25x speedup for Human Dataset 2 against SMusket. Even though this margin is not as considerable as the empirical speedup results from Musket, it still suggests that GPUMusket maintains a stable performance advantage.

Table 3. Speedup Results of GPUMusket vs. other EC Tools on Real Human Datasets

| Dataset | Speedup (GPUMusket vs. Musket) | Speedup (GPUMusket vs. SMusket) |
|---|---|---|
| *Human Dataset 1 (RC = 78.5M)* | 2.61x | none |
| *Human Dataset 2 (RC = 34.9M)* | 1.64x | 1.26x |

3.3.2 Peak Resident Memory

Peak resident memory refers to the maximum amount of physical RAM (Random Access Memory) a program consumes during its execution. It is a critical metric in evaluating the resource efficiency, scalability, and accessibility of a computational tool, particularly for pre-processing applications like genomic error correction. Comparing the results of GPUMusket against Musket across three real datasets of increased read counts: E.Coli Dataset 1 with 22.7 million reads, Dataset 2 with 34.9 million reads and Dataset 3 with 78.5 million reads that are both sampled from a human, Musket consistently resulted in the lowest memory footprint across all datasets, even for large datasets (~11.2GB for Dataset 3), implying strong algorithmic optimization for CPU-based executions.

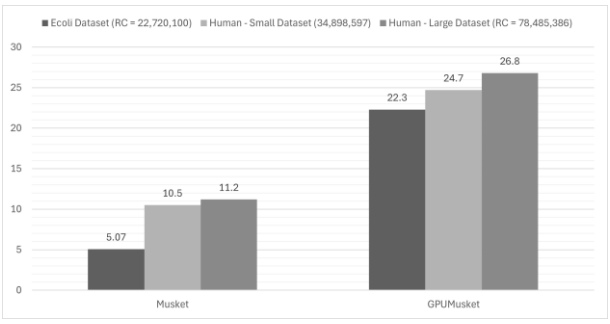


Fig. 5. Peak Resident Memory Results (in GB) for Different EC Tools on Varying Real Datasets.

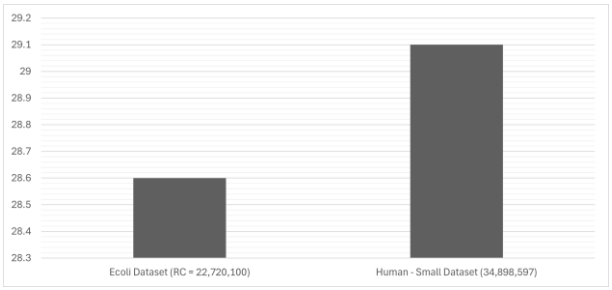


Fig. 6. Peak Resident Memory Results for SMusket (in GB) for Different EC Tools on Varying Real Datasets.

However, SMusket consistently consumes the highest memory with ~29GB regardless of dataset size and shows almost no scaling across all two (2) datasets. As mentioned previously in section 4.4.1, the largest human dataset did not produce results for SMusket due to an error in the execution process, thus being excluded in the illustration in Figure 4.8. While GPUMusket demonstrates intermediary memory usage (22.3 – 26.8 GB) scaling moderately at ~1.2x from Dataset 1 (E.Coli) to Dataset 3 (Human-Large), indicating better adaptability than SMusket but less efficiency than Musket. Thus, GPUMusket strikes a balance, offering GPU acceleration with better memory handling than SMusket but still requiring substantial resources.

### 3.4 Evaluation of Results of De Novo Assembly

The corrected H.Salinarum and E. coli datasets were assembled using the SPAdes de novo assembler, comparing performance with and without read trimming. The entire correction and assembly process was repeated a minimum of 5 times to determine variability in results. However, after every execution, there is low variability because of consistent results. Therefore, the results demonstrate that trimming is optional, as its impact varies depending on the dataset and error correction tool used.

For H.Salinarum, both GPUMusket and Musket produced identical N50 values (323,520) and contig counts (68), indicating high assembly continuity regardless of trimming. The minor differences in error rates (GPUMusket: +0.002%) and nearly identical genome fractions (~84.68%) suggest that trimming had negligible effects on assembly quality for this dataset. This consistency implies that H.Salinarum reads were already of sufficient quality, making trimming unnecessary for optimal assembly. In contrast, for E. coli, Musket outperformed GPUMusket in most metrics, achieving a higher N50 (105,630 vs. ~83,000) and fewer contigs (114 vs. 147 without trimming), indicating better contiguity. Musket also maintained a lower error rate (0.015% vs. 0.085%) and a higher genome fraction (97.99% vs. 97.07%), suggesting a more complete and accurate assembly. However, GPUMusket showed a slight improvement in genome fraction after trimming (97.76%), implying that trimming may help recover additional genomic content in some cases.

While GPUMusket may exhibit slightly higher error rates in some cases, it maintains competitive assembly continuity and genome fraction, demonstrating its reliability as a robust error correction tool. Additionally, its consistent performance on the H.Salinarum dataset—where trimming had a negligible impact—highlights its ability to handle high-quality reads effectively, reducing

the need for preprocessing overhead and streamlining the assembly workflow.

## 4. CONCLUSION

GPUMusket represents a significant advantage in bioinformatics by harnessing GPU acceleration to improve the efficiency and scalability of error correction in short-read genomic data while maintaining competitive accuracy in downstream de novo assembly. By achieving substantial speedup rates over its CPU-based tools like Musket and SMusket, GPUMusket addresses a critical challenge in genome assembly pipelines, enabling researchers to process large-scale datasets in significantly less time. This enhancement is particularly valuable in real-world scenarios where rapid turnaround is essential, such as clinical pathogen surveillance, population genomics, and large-scale comparative studies. While Musket retains an edge in precision for smaller genomes due to its conservative correction approach, GPUMusket's scalability and statistically comparable assembly results make it a potential alternative solution for high-throughput projects requiring computational efficiency.

Notably, GPUMusket displays robust assembly performance on high-quality datasets, producing near-similar assembly metrics to Musket while eliminating the need for additional trimming, streamlining workflows, and reducing preprocessing overhead. Furthermore, its efficient GPU utilization minimizes memory bottlenecks, making it more adaptable to resource-constrained environments than distributed CPU solutions like SMusket. Additionally, its scalability with multiple GPUs ensures it can keep pace with ever-increasing sequencing data volumes. Overall, GPUMusket bridges the gap between speed and accuracy in genome assembly preprocessing, offering a practical, high-performance alternative to traditional CPU-based correctors. Its ability to accelerate error correction without compromising assembly integrity positions it as a valuable tool for modern genomics research, where the demand for rapid, cost-effective, and scalable solutions continues to develop.

## 5. REFERENCES


[1] J. Hu, Z. Wang, Z. Sun et al., "NextDenovo: an efficient error correction and accurate assembly tool for noisy long reads," *Genome Biology*, vol. 25, p. 107, 2024. doi: 10.1186/s13059-024-03252-4.

[2] Y. Meng et al., "Genome sequence assembly algorithms and misassembly identification methods," *Molecular Biology Reports*, vol. 49, no. 11, pp. 11133–11148, 2022. doi: 10.1007/s11033-022-07919-8.

[3] A. Whibley, J. L. Kelley, and S. R. Narum, "The changing face of genome assemblies: Guidance on achieving high-quality reference genomes," *John Wiley & Sons LTD*, 2020. doi: 10.1111/1755-0998.13312.

[4] P. Zhang et al., "Comparison of de novo assembly strategies for bacterial genomes," *International Journal of Molecular Sciences*, vol. 22, no. 14, p. 7668, 2021. doi: 10.3390/ijms22147668.

[5] J. Wang et al., "Systematic comparison of the performances of de novo genome assemblers for Oxford Nanopore Technology reads from Piroplasm," *Frontiers in Cellular and Infection Microbiology*, vol. 11, 2021. doi: 10.3389/fcimb.2021.696669.

[6] F. Dida and G. Yi, "Empirical evaluation of methods for de novo genome assembly," *PeerJ Computer Science*, vol. 7, p. e636, 2021. doi: 10.7717/peerj-cs.636.

[7] M. Teng and R. A. Irizarry, "Accounting for GC-content bias reduces systematic errors and batch effects in ChIP-seq data," *Cold Spring Harbor Laboratory Press*, 2025. [Online]. Available: genome.cship.org.

[8] H. I. Nakaya, *Bioinformatics*. Brisbane (AU): Exon Publications, 2021. doi: 10.36255/exonpublications.bioinformatics.2021.

[9] C. Moeckel et al., "A survey of k-mer methods and applications in bioinformatics," *Computational and Structural Biotechnology Journal*, vol. 23, pp. 2289–2303, 2024. doi: 10.1016/j.csbj.2024.05.025.

[10] R. Vaser and M. Sikic, "Raven: a de novo genome assembler for long reads," *bioRxiv*, 2020. doi: 10.1101/2020.08.07.242461.

[11] R. R. Exposito, J. Gonzalez-Dominguez, and J. Tourino, "SMusket: Spark-based DNA error correction on distributed-memory systems," *Future Generation Computer Systems*, vol. 111, pp. 698–713, 2020.

[12] X. Liao et al., "Current challenges and solutions of de novo assembly," *Quantitative Biology*, vol. 7, no. 2, pp. 90–109, 2019.

[13] A. Müller, R. Membarth, R. Leißa, and S. Hack, "AnySeq/GPU: A Novel Approach for Faster Sequence Alignment on GPUs," in *Proc. 2022 Int. Conf. Supercomputing (ICS '22)*, Virtual Event, USA, 2022. doi: 10.1145/3524059.3532376.

[14] B. Schmidt and A. Hildebrandt, "From GPUs to AI and quantum: Three Waves of Acceleration in Bioinformatics," *Drug Discovery Today*, vol. 29, no. 6, 2024. doi: 10.1016/j.drudis.2024.103990.

[15] J. Li et al., "Recent technology advancements in large-scale DNA assembly," *Wiley Online Library*, vol. 1, no. 1, p. e79, 2024. doi: 10.1002/qub2.79.

[16] V. Dominguez Del Angel et al., "Ten steps to get started in Genome Assembly and Annotation," *F1000Research*, vol. 7, 2018. doi: 10.12688/f1000research.13598.1.

[17] Y. Liu, J. Schroeder, and B. Schmidt, "Musket: a multistage k-mer spectrum-based error corrector for Illumina sequence data," *Bioinformatics*, vol. 29, no. 3, pp. 308–315, 2013.

[18] R. Wittler, "General encoding of canonical k-mers," *Peer Community Journal*, vol. 3, p. e87, 2023. doi: 10.24072/pcjournal.323.

[19] H. B. Luleci, S. A. Yuka, and A. Yilmaz, "Efficient Storage and Analysis of Genomic Data: A k-mer Frequency Mapping and Image Representation Method," *Interdisciplinary Sciences*, 2024. doi: 10.1007/s12539-024-00659-2.

[20] Y. Bussi, R. Kapon, and Z. Reich, "Large scale k-mer-based analysis of the informational properties of genomes, comparative genomics, and taxonomy," *PLoS ONE*, vol. 16, no. 10, p. e0258693, 2021. doi: 10.1371/journal.pone.0258693.

[21] G. Greenberg, "Analysis and applications of k-mer based methods in bioinformatics," 2023. [Online]. Available: https://hdl.handle.net/2142/121999.

[22] A. Hijikata et al., "Exome-wide benchmark of difficult to sequence regions using short-read next-generation DNA sequencing," *Nucleic Acids Res.*, vol. 52, no. 1, pp. 114–124, Nov. 2023, doi: 10.1093/nar/gkad1140.

[23] S. Watanabe, T. Aoki, and S. Tsuzuki, "Performance tuning on GPU for granular simulation based on discrete element method saving memory usage," *Trans. Japan Soc. Comput. Eng. Sci.*, vol. 2016, no. 0, p. 20160013, 2016, doi: 10.11421/jsces.2016.20160013.