FT K-Means: A High-Performance K-Means on GPU with Fault Tolerance

***Abstract*—K-Means is a widely used algorithm in clustering, however, its efficiency is primarily constrained by the computational cost of distance computing. Existing implementations suffer from suboptimal utilization of computational units and lack resilience against soft errors. To address these challenges, we introduce FT K-Means, a high-performance GPU-accelerated implementation of K-Means with online fault tolerance. We first present a stepwise optimization strategy that achieves competitive performance compared to NVIDIA's cuML library. We further improve FT K-Means with a template-based code generation framework that supports different data types and adapts to different input shapes. A novel warp-level tensor-core error correction scheme is proposed to address the failure of existing fault tolerance methods due to memory asynchronization during copy operations. Our experimental evaluations on NVIDIA T4 and A100 GPUs demonstrate that FT K-Means without fault tolerance outperforms cuML's K-Means implementation, showing a performance increase of 10%-300% in scenarios involving irregular data shapes. Moreover, the fault tolerance feature of FT K-Means introduces only an overhead of 11%, maintaining robust performance even with tens of errors injected per second.**

I. Introduction

The K-Means, one of the top 10 algorithms in data mining [1], is widely used in image classification, vector quantization [2], knowledge discovery [3], and pattern classification [4]. However, K-Means is increasingly vulnerable to transient faults caused by high circuit density, low near-threshold voltage, and low near-threshold voltage [5, 6, 7]. Oliveira et al. [8] demonstrated an exascale system with 190,000 cutting-edge Xeon Phi processors that still suffer from daily transient errors under ECC protection. Recognizing the importance of this issue, the U.S. Department of Energy has named reliability as a major challenge for exascale computing [9].

Intel Corporation first documented a transient error leading to soft data corruption in 1978, marking a significant recognition of the impact of such faults in both academia and industry [10]. Subsequently, in 2000, Sun Microsystems reported server crashes and outages at major sites like America Online and eBay back to cosmic ray strikes on unprotected caches [11]. Similarly, Virginia Tech, in 2003, had to dismantle and sell its newly assembled Big Mac cluster of 1100 Apple Power Mac G5 computers due to a lack of ECC protection that resulting the system prone to cosmic ray-induced failures [12]. Despite advancements in ECC protection, transient faults continue to challenge system reliability. For example, a simulation by Oliveira et al. of an exascale system with 190,000 advanced Xeon Phi processors revealed daily vulnerabilities to transient errors, even with ECC [8]. These faults are not just theoretical concerns; real-world impacts have been recorded by Google, which experienced transient faults causing incorrect outputs in its production environment [13]. In 2018, faced with the ongoing risk of transient faults on its large-scale infrastructure, Meta launched an internal investigation to seek solutions [14]. Transient faults may cause either fail-stop errors, which lead to system crashes, or fail-continue errors, which produce incorrect outcomes. While fail-stop errors can often be addressed through checkpoint/restart mechanisms [15, 16, 17, 18] or algorithmic approaches [19, 20, 21], fail-continue errors are more problematic as they silently corrupt the state of applications and result in incorrect outputs [22, 23, 24, 25, 26]. These types of errors are particularly dangerous in environments where safety is critical [27]. This paper focuses on fail-continue errors within the computational logic units, assuming that errors in memory and those causing system stops are managed using error-correcting codes and checkpoint/restart techniques. We refer to these issues as soft errors.

Existing fault tolerance methods are feasible for K-Means clustering. Taamneh [28] proposes a checkpointing strategy, which involves periodically saving the centroids to stable storage during normal operation and restarting from checkpointed centroids in the event of a failure. However, this method cannot detect silent errors and requires recomputation after an error. Dual modular redundancy (DMR) verifies computational correctness by replicating instructions and comparing the results [29, 30, 31, 32, 33]. Figure 1 left illustrates the K-Means workflow. We find that DMR can protect the memory-bound routine efficiently, such as the centroids update phase in Figure 1 left step 3. It is because the memory latency of loading the data points is so high that all arithmetic operations can be duplicated without introducing an overhead over 1%, even for synchronous instructions like atomicAdd. However, DMR failed to protect the compute-bound distance calculation in Figure 1 left step 1 due to the redundant computations. In Figure 1 left step 2, nearest cluster matching is fused within the distance computation kernel to eliminate redundant memory acess. Algorithm-based fault tolerance (ABFT) reduces this redundancy by using checksums based on equivalence relationships, lowering the redundancy to $O(1/N)$, where N is the problem size [34]. Efforts have been made to apply ABFT on GPUs, designing kernel fusion schemes to minimize the overhead of ABFT by hiding it within the memory transactions and computing unit gaps [35, 36, 37, 38].

However, existing ABFT fusion strategies lack architecture-aware optimization. Kosaian [37] proposes an error detection scheme for tensor-core but the error correction requires a time-redundant recomputation. Although error correction is dis-

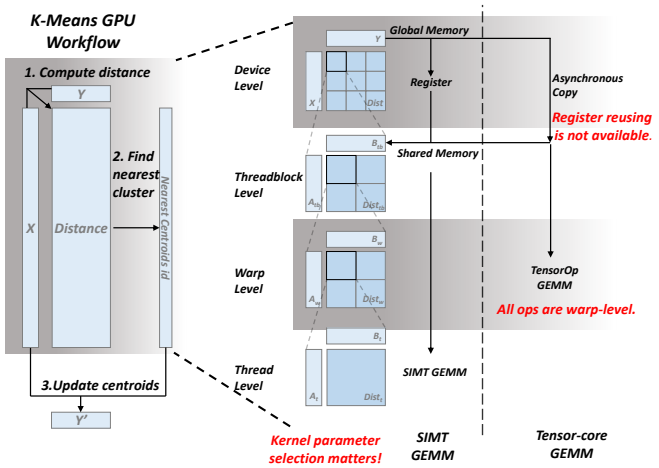

Figure 1: Workflow of K-Means. The red illustrates our motivation.

cussed in [36, 38], they only utilize a register-reusing which becomes outdated for Amepere architecture. The register reusing stage is illustrated in the SIMT GEMM part of Figure 1 and the asynchronous copy is presented in the Tensor-core part. Before Ampere architecture, the global-to-share memory transfer passes through the register file explicitly, enabling register reusing for checksum computation. However, the latest asynchronous memory copy enables a new data path from global to shared memory bypassing the register. Without register reusing, checksum computation requires additional expensive memory read, destroying the carefully designed pipeline. Additionally, these fault tolerance strategies consider matrix multiplication individually, with no aware of the underlying K-Means routine.

Furthermore, the state-of-the-art K-Means implementation, cuML suffers from fixed kernel parameters, resulting in sub-optimal and low utilization of peak computing performance over various problem sizes. For example, the K-Means in cuML obtain a performance of less than 10% of the peak performance for both single and double precisions. Due to the underlying tall-and-skinny matrix multiplication computation inside the K-Means, a code generation strategy and a parameter selection scheme are necessary to improve the K-Means performance for various of input shapes and data types.

To address these issues, we propose FT K-Means, a high-performance K-Means implementation equipped with an algorithm-based fault tolerance scheme that detects and corrects silent data corruptions at computing units on-the-fly. More specifically, our contributions include the following:

- We begin our work by optimizing a K-Means baseline without fault tolerance. Through a series of optimizations on kernel fusion, FT K-Means offers performance competitive to or faster than the state-of-the-art library, cuML. FT K-Means is available at an anonymous link.[1]
- We explore the fault tolerance K-Means designs at the warp level using both the CUDA cores and tensor cores. The combination presents a low overhead even under tens of errors injected per minute.
- A template-based code generation strategy is developed to reduce development costs. The template-based can generate K-Means kernels with or without fault tolerance for a wide range of input sizes and data types.
- Experimental results of single precision and double precision on an NVIDIA A100 server GPU and a Tesla Turing T4 GPU show that FT K-Means offers a $10\% - 300\%$ improvement compared to the state-of-the-art library cuML. The fault tolerance scheme in FT K-Means introduces an average overhead of 11%, even under tens of error injections per minut.

[1] https://github.com/ics2024artifact/artifact.git

II. Background and Related Works

The K-means algorithm aims to categorize M objects $\{X_i\}_{i=1}^{M}$ based on N-dimensional features by grouping similar ones into K clusters. Starting with initial centroids $Y_i^{(0)}{}_{i=1}^{K}$, the algorithm iteratively performs two main steps until satisfies a termination condition. Firstly, it assigns each object X_i to the nearest cluster Y_j using

$$\arg\min_{j=1,\cdots,K} \|X_i - Y_j\|_2. \tag{1}$$

Secondly, it updates the centroids of each cluster with

$$Y_j' = \frac{1}{|S_j|} \sum_{i \in S_j} X_i \tag{2}$$

where S_j represents the set of all points X_i that are assigned to cluster j, and $|S_j|$ denotes the number of points in cluster j. This formula computes the new centroid Y_j as the mean of all points assigned to the cluster j.

A. Fault Model

FT K-Means is designed to identify and correct errors in computing units that could influence the final results. It operates under the assumption that memory errors are managed by ECC [39] and communication reliability issues are addressed by FT-MPI [40]. For compute errors during runtime, a fault-tolerant strategy is implemented based on the single-event upset (SEU) assumption [41, 42], which posits that only one soft error occurs within each detection and correction interval. This assumption is supported by the low frequency of multiple soft errors, attributed to brief fault detection intervals—a concept prevalent in several studies [31, 35, 36, 43]. The fault tolerance method involves high reliability in detecting faults with minimal false alarms [44]. Specifically, each threadblock randomly selects an element to corrupt by flipping a single bit, either in its 32-bit float representation or 64-bit double representation. A checksum test with a defined threshold δ then attempts to operate the corrupted computations, followed by an application of an error correction scheme.

B. Previous Work for K-Means without Fault Tolerance

Historically, the simplicity of the k-means algorithm has led to its frequent reimplementation on GPUs. These implementations vary, starting with early GLSL-based versions [45] and progressing to initial and more recent CUDA versions [46, 47]. Lutz et al. [48] highlighted the significance of single-pass processing in GPU computations, which avoids the redundant loading of data into GPU caches. They reported a significant

performance improvement, with speeds over 50 times faster than contemporary CPU implementations and twice as fast as 'double-pass' strategies that involve separate steps. This method is particularly effective for simultaneously processing multiple small k-means instances by using kernel fusion [49]. Meanwhile, Cuomo et al. [50] conducted a detailed analysis of the performance costs associated with transferring large datasets between CPU and GPU for specialized processing. The kmcuda package provides a GPU-optimized 'YinYang' k-means algorithm on GitHub [51], which enhances processing speed on CPUs but faces significant overhead with GPU parallel processing, and is widely used in data analysis tools like R. Nelson and Palmieri [52] emphasized the significance of memory management and synchronization, discussing the trade-offs between using global versus shared memory, and different thread synchronization models that involve memory locking. Martin et al. [53] present a detailed analysis of individual computation steps and propose several optimizations that improve the overall performance. Kaiming et al. [54] introduced an integrated approach to the critical steps of the K-means algorithm, which significantly enhances performance by reducing unnecessary calculations and memory operations, outperforming the Intel DAAL K-means in both sequential and parallel environments.

However, existing works fail to provide enough optimization toward the tensor-core computing units and the asynchronous memory copy presented in the GPU architecture after SM80. cuML [55], one of the state-of-the-art machine learning libraries on GPU, has provided a K-Means implementation using the latest architecture properties. Despite using tensor core computing units and the latest architectural features, the performance of the cuML K-means implementation remains below its hardware's peak potential due to fixed kernel parameters that do not optimize for different input shapes.

C. Algorithm-Based Fault Tolerance

Soft error protection algorithms aim to identify and rectify errors that can arise in iterative or computationally demanding applications. The development of these algorithms dates back to 1984, with Huang [56] first specifically designed for matrix-matrix multiplication. The fundamental concept of these algorithms involves encoding matrices X and Y into checksums X^c and Y^r, respectively. This encoding is achieved through the following equations.

$$X \xrightarrow{encode} X^c := e^T X, \tag{3}$$

$$Y \xrightarrow{encode} Y^r := Ye, \tag{4}$$

where e is a column vector, $[1, 1, \ldots, 1]$. The combinations of input matrix and the encoded checksums, $X' = \begin{bmatrix} X \\ e^T X \end{bmatrix}$ and $Y' = \begin{bmatrix} Y & Y^r \end{bmatrix}$, are then multiplied to get a matrix D' that contains both the correct result and checksum information:

$$D' = X'Y' = \begin{bmatrix} D & De \\ e^T D & \end{bmatrix} = \begin{bmatrix} D & D^r \\ D^c & \end{bmatrix}. \tag{5}$$

The final accuracy of matrix multiplication can be confirmed by comparing the matrix D with its checksum counterparts D^r and D^c. If the difference exceeds a set threshold, it signals an error in the computation. The cost of this checksum encoding and verification is minimal compared to the matrix multiplication itself, offering a cost-effective error detection method. These algorithms can be applied to any matrix multiplication process and verification can occur either during (online) or after (offline) the computation.accumulation Chen et al. [34] introduced an outer-product matrix-matrix multiplication algorithm that maintains the checksum relationship throughout the accumulation process:

$$D' = \sum_i X'(:, i) \cdot Y'(s, :) = \sum_i \begin{bmatrix} D_i & D_i e \\ e^T D_i & \end{bmatrix}. \tag{6}$$

where, i represents the step number in the outer-product update of the matrix D, with D_i indicating the result of each outer-product, namely $X'(:, i) \cdot Y'(i, :)$. The offline version of the double-checksum approach can correct only one error in the entire computation. In contrast, the online version corrects a single error at each step of the update, allowing it to address multiple errors throughout the entire program.

In 2011, Ding et al. [35] presented an implementation of the outer-product ABFT-GEMM for GPUs. To minimize the memory latency from checksum operations in ABFT, Zhai et al. developed combined compute kernels for GEMM on AVX-512-enabled CPUs [36]. Kosaian and Rashmi [37] present a warp-level ABFT implementation capable of error detection, but not correction. Shixun et al. [38] proposed a fully-fused ABFT-GEMM that effectively detects and corrects computational errors. However, the kernel fusion strategy relies on reusing registers during transfers between global and shared memory. This approach does not extend well to GPU architectures post-Turing, e.g. Ampere, due to the introduction of asynchronous copy instructions.

III. K-Means without Fault Tolerance

The K-Means process in each iteration can be divided into two stages. First, for every sample, assign a cluster that has the closest Euclidean distance to it. Second, calculate the geometric center for all samples belonging to a cluster as the new coordinates for this cluster. Consider M samples and K clusters, where each sample has a dimension of N. The time complexity of the first stage is $O(MNK)$, while the time complexity of the second stage is $O(MN)$, so the major bottleneck lies in the first stage. In this section, we present the step-wise optimizations of K-Means. The optimizations include applying GEMM to K-Means, kernel fusion with thread-wise and thread block-wise reduction, broadcast between thread blocks, and enabling tensor core in GEMM.

A. K-Means Stepwise Optimization

1) Basic implementation: For the cluster assignment stage, we launch a kernel. Each thread in this kernel handles a line in the sample matrix (refer to the matrix definition in Fig 2), which represents a sample. The thread loads all centroids in

the centroid matrix calculates the Euclidean distance between this sample and every centroid, and chooses the one with the smallest distance as its assigned centroid. For the update centroids stage, M kernels are launched in serial. In kernel i, each thread processes one sample. If this sample belongs to kernel i, then add all dimensions of this sample to kernel i's corresponding dimensions. Finally, launch a kernel to calculate $\frac{sum\ of\ samples\ belongs\ to\ this\ centroid}{number\ of\ samples\ belongs\ to\ this\ centroid}$, and write the answer back to Centroids matrix as new centroids.

2) GEMM based K-Means: Due to the property of Euclidean distance, and we only need to find the closest centroid, we can use GEMM to speed up the cluster assignment process. Ignoring the squared root in distance computation, the distance can be computed in three parts: $\Sigma_k Samples_{ik}^2 + \Sigma_k Centroids_{jk}^2 - 2 \cdot \Sigma_k Samples_{ik} \cdot Centroids_{jk}$ As Figure 2 Step 1 depicts, the first two parts of this formula can be computed by squaring elements and summing them up in each row. This can be finished by launching two simple kernels. The third part of the formula has the highest time complexity, and it is exactly in the form of GEMM, so we can launch a GEMM kernel to handle this part and put it together with the first two square terms, and write back to GPU memory. Then we need to launch another kernel to reduce over each row to find the closest centroid for each sample. Also, as illustrated in Figure 2 Step 3, for the updating centroids stage, launching N kernels is a great waste of time, because, in kernel j, a large number of threads are idle because the sample which is handled by it doesn't belong to centroid j. So we only launch one kernel to handle N centroids all at once. Each thread still deals with one sample, but it uses atomic add to add the values of this sample in every dimension to its assigned centroid and add one to the counter of this assigned centroid (for average operation). And the last kernel of averaging over each centroid remains unchanged. Our optimization boosts the performance to 25x compared to the basic implementation.

3) Kernel Fusion in thread and thread block level: After GEMM, we write the result matrix back to GPU memory and load the matrix again in order to do a row-wise reduction. This greatly increases the amount of data movement and increases storage overhead. In this part, we apply kernel fusion to accomplish part of the reduction within the GEMM kernel. Firstly, as Figure 2 Step 2 indicates, each thread in the GEMM kernel handles a small submatrix of the result matrix. After the computation of GEMM, we can simply find the minimum column in each row within this submatrix, and write it to shared memory. After all threads in the threadblock finish this step, thread 0 reads all results in shared memory, calculates another row-wise minimum for each row as the final partial answer for this threadblock, and writes it to GPU memory. Assuming each threadblock has size $TM \times TN$, the reduction kernel only needs to load and handle TN/N of data compared to the last part, which obtains a speedup of 1.13x compared to last part.

4) Threadblock level broadcast: The optimization above cannot fully eliminate the time and space complexity of launching another reduction kernel after the GEMM kernel. So in this part, we attempt to accomplish cluster assignment within the GEMM kernel. Owing to the fact that different threadblocks cannot communicate, we use a broadcast vector and atomic operation to ensure that only one threadblock is changing the assignment answer in one row at a time, i.e. each threadblock needs to acquire for the lock of a row before changing the assignment answer in this row. With this method, the speedup increases to 1.04x compared to the last part.

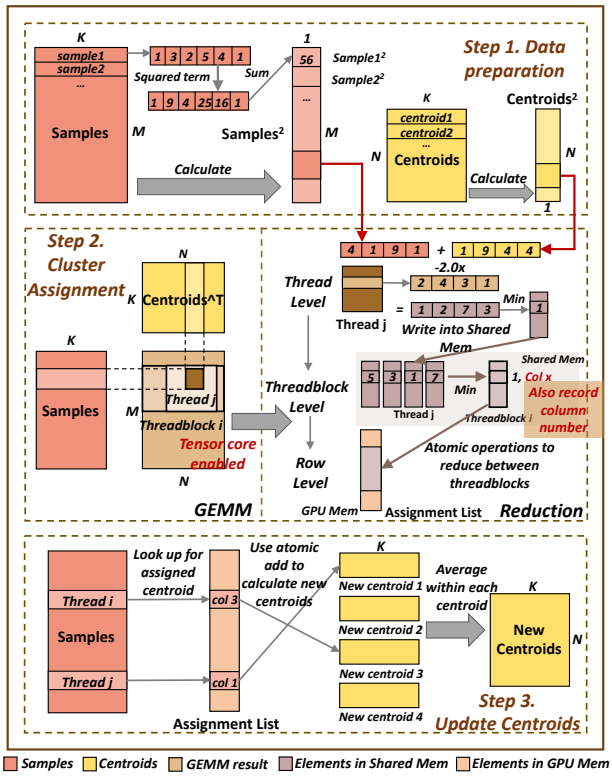

Figure 2: Overview of the optimized K-Means.

5) Enabling tensor core in GEMM: Modern Nvidia GPUs are equipped with a tensor core in each Streaming Multiprocessor (SM), designed to accelerate GEMM. Hence we use GEMM kernels in cutlass with tensor core and enable TF32 in FP32 precision to further boost performance, instead of our hand-written GEMM kernel. The reduction part in the last section is placed into the GEMM kernel as an epilogue. With this optimization, we achieve a speedup of 1.45x compared to the previous optimization. And now, the fully optimized GEMM is presented in Figure 2

B. Automatic Code Generation

cuML has a highly optimized open-source K-Means implementation based on cutlass. However, in the cluster assignment stage, it has hard-coded parameters in its GEMM kernel, which can trigger low performance in some input sizes. Moreover, the parameters for a cutlass GEMM kernel must be hard-coded in order to pass compile-time checking. So the cost of integrating GEMM kernels with customized parameters becomes unacceptable. We propose a code generation strategy to generate kernels with different parameters while minimizing code length to the greatest extent possible.

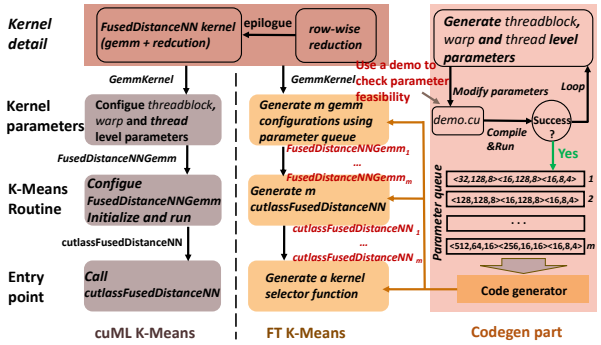

Figure 3: Overview of the code generation Strategy

Default naming of kernel parameters A group of kernel parameters in cuML and Cutlass refers to a set of parameters, threadblock level parameters, warp level parameters and thread level parameters. Each level is composed of three parameters from each dimension. For example, in warp level, the three parameters are labeled Warp.M, Warp.N, and Warp.K, and they refer to the three dimensions of M, N, and K in GEMM.

Code Generation Strategy Figure 3 demonstrates the method we used in our code generation strategy. The code structure of the cutlass GEMM kernel in cuML are as the left side of the figure. Firstly, the epilogue is integrated into the FusedDistanceNN kernel. The FusedDistanceNN kernel handles both GEMM and reduction. And then the kernel is wrapped into GemmKernel. With threadblock, warp, and thread level parameters set, it is then transferred to the next template as FusedDistanceNNGemm. Finally, with all K-Means routine sets, the whole cutlassFusedDistanceNN template function works as an interface of the cluster assignment stage. In order to generate a set of feasible kernels with customized parameters, we write a code to test all possible parameters in the search space defined by ourselves, as shown in the codegen part. And for every group of parameters, try it in a demo code. If it can compile and run, which means it is functionally correct. And then we put it in the parameters queue. With all parameters, we then run a code generator to modify the three parts of the original cuML source code, injecting corresponding functions for each parameter in each stage.

1) Kernel Parameters: The kernel parameters used in code generation is not chosen by brute forcing every possible integer in a range for every parameter in the parameter group (a parameter group means: threadblock level, warp level and thread level parameters). We follow some rules. 1) all parameters must be power of 2 2). Warp.K = Threadblock.K. 3). warp size/thread size is 8 or 16. 4). thread size is fixed for FP32 (16, 8, 4) and FP64 (8, 8, 4) owing to the size of the tensor core.

2) Code Generation Template: The code generation strategy defines the K-Means kernel using different tiling parameters. For single precision, 157 kernels with different parameter sets are defined while 145 kernels are defined for double precision. The test workflow illustrated in Figure 3 checks the feasibility of those kernels and performs the benchmark over 64 problem sizes. The benchmark result of different kernels will be employed as the kernel selection criterion. Below, we give a brief

```
01: Registers: A_t[m_w], B_t[n_w], C_t[2],
02: Shared memory: A_tb[k_stage][m_tb×k_tb ], B_tb[k_stage][n_tb×k_tb ]
03: // Load the first k_stage - 1 tiles of A into tile A_tb using
04: for stage = 0, 1, ..., k_stage - 1:
05:    asm ("cp.async.ca.shared.global [%0], [%1], 16;\n" :
            : A_tb[stage][tid],
              "l"(A[tid]));
06:    asm ("cp.async.ca.shared.global [%0], [%1], 16;\n" :
            : B_tb[stage][tid],
              "l"(B[tid]));
07:    asm volatile("cp.async.commit_group;\n" ::);
08: asm ("cp.async.wait_group 1;\n" ::);
09: __syncthreads();
10: A_t[0:m_w] ← A_tb[0][*]
11: B_t[0:n_w] ← A_tb[0][*]
12: for k = 0, 8, 16, ..., K
13:       asm ("cp.async.ca.shared.global [%0], [%1], 16;\n" :
              : A_tb[stage][tid],
                "l"(A[tid]));
14:       asm ("cp.async.ca.shared.global [%0], [%1], 16;\n" :
              : B_tb[stage][tid],
                "l"(B[tid]));
15:    for i = 0, 1, ..., m_w - 1
16:       for j = 0, 1, ..., n_w - 1
17:          asm volatile("mma.sync.aligned.m8n8k4.row.col.f64.f64.f64.f64
                {%0,%1}, {%2}, {%3}, {%4,%5};\n"
                : "=d"(c[i][j][0]), "=d"(c[i][j][1])
                : "d"(a[0]), "d"(b[0]), "d"(c[i][j][0]), "d"(c[i][j][1]));
18:    asm ("cp.async.commit_group;\n" ::);
19:    asm volatile("cp.async.wait_group %0;\n" ::"n"(1));
20:    __syncthreads();
10:    A_t[0:m_w] ← A_tb[(k / 8 + 1) % k_stage][*]
11:    B_t[0:n_w] ← A_tb[(k / 8 + 1) % k_stage][*]
21: Epilogue
22: ...
```

Optimized Dsitance Compute & Cluster Assign Kernel

Figure 4: Pseudocode: K-Means w/o FT

explanation of the K-Means kernel so that we can switch to the discussion of fault tolerance part.

The main body of the kernel is shown in Figure 4. In Figure 4, A stands for samples X, B for centroids Y, and C for the distance matrix D. From line number 04 to 07, an asynchronous multi-stage pipeline from global to shared memory starts and a group barrier is committed for each iteration. At line 08, the kernel waits for at least one group to be ready. Once there the group is prepared, the whole thread block loads data from shared memory into the register synchronously. Next comes the main loop along the number of clusters K. At the start of the main loop, the latest memory asynchronous copy is pushed into the pipeline, as shown in lines 13 - 14. After that, the warp-level matrix multiplication is performed by tensor-core units. At the end of this iteration, the latest group is committed and new data is refreshed into the register once a previous group is finished. When the main loop is finished, the memory-bound epilogue performs a reduction along the row of the C matrix. Due to limited space, we skip the detailed description and the reader is welcome to our open-sourced implementation.

IV. FT K-Means With Fault Tolerance

In this section, we present the fault tolerance scheme used in FT K-Means. Our discussion concentrates on distance computing and the reduction operation to find the nearest cluster. As mentioned before, the centroids updating stage can be handled with a negligible overhead of less than 1% by simply applying the DMR strategy.

A. Online correction with location encoding

Figure 5 compared the fault tolerance scheme in FT K-Means with existing state-of-the-art methods in detail.

One ABFT is illustrated in Figure 5 (a). To minimize the overhead associated with ABFT, encodings are applied at the

	Y	Ye_1	
X	XY	$X(Ye_1)$	
e_1^TX	$(e_1^TX)Y$		

(a) Shixun's FT-GEMM

	Y	Ye_1	
X	XY		
e_1^TX		$(e_1^TX)(Ye_1)$	

(b) Jack's FT-GEMM

	Y	Ye_1	Ye_2
X	XY		
e_1^TX		$(e_1^TX)(Ye_1)$	$(e_1^TX)(Ye_2)$
e_2^TX		$(e_2^TX)(Ye_1)$	

(c) FT K-Means

	Shixun's	Jack's	FT K-Means
Level	Threadblock	Warp	Warp
SIMT	√	√	√
Tensor Core	×	√	√
Detection	√	√	√
Correction	√	×	√

(d) Comparisons

Figure 5: Comparison of SOTA ABFT-GEMM schemes. Blue and green areas are baseline and ABFT computations, respectively.

thread block level. Aiming to avoid additional latency during GEMM accumulation, the prefetching stage in GEMM is fused with all encodings. In GPU architecture prior to Ampere, the prefetching stage incorporates all ABFT encodings. With a carefully designed prefetching strategy, a warp-level reduction can obtain each element in e^TX and Ye without requiring extra global read operations. Subsequently, the prefetching strategy facilitates the availability of e^TXY and XYe encodings within a thread, eliminating the need for thread block-level communication. Ultimately, the target checksums are accumulated via a threadblock-level reduction.

Compared to the detection scheme specifically designed for tensor-core GPU, as shown in 5 (b), our method employs a vector $e_2 = [1, 2, \cdots, N]$ to checksum the inputs again, in addition to the previous $e_1 = [1, 1, \cdots, 1]$. Our method doubles the computational cost on CUDA cores (e^TX, Ye) and triples the computational cost on tensor cores ($e_1^TXYe_2$, $e_2^TXYe_1$, and $e_1^TXYe_1$).

B. *Implementation of FT K-Means with fault tolerance*

The red part in Figure 6 illustrates the injected instructions to implement FT K-Means. From line number 15 to 18, the checksum e_1^TX, Ye_1, e_2^TX, and Ye_2 are computed. This accumulation takes place inside a thread, getting rid of inter-thread communication and additional memory operation. From line number 22 to 24, $e_1^TXYe_1, e_1^TXYe_2$, and $e_2^TXYe_1$ are computed via tensor-core MMA operation. The overhead of our ABFT method mainly comes from those three MMAs. Theoretically, they will incur a computation overhead of $\frac{3}{m_w \times n_w}$. Assume $m_w = 4$ and $n_w = 2$, the theoretical overhead is 37.5%. However, from our experimental evaluation in Section V, the overhead is only 11% on average. This theory-experiment mismatch indicates that at the thread level and warp level, there remains a 27.5% execution bubble between computation and memory, which is available for kernel fusion. Actually, we first try to get the checksums e_1^TX, Ye_1, e_2^TX, and Ye_2 using the tensor core as well, namely embedding e_1, e_2 into a new matrix operand so that we can get the checksum through several tensor core operation. However, those tensor operations cannot be hidden behind the memory footprint, resulting in a 50% overhead approximately.

```
01: Registers: A_t[m_w], B_t[n_w], C_t[2],
02: Shared memory: A_tb[k_stage][m_tb×k_tb ], B_tb[k_stage][n_tb×k_tb ]
03: // Load the first k_stage - 1 tiles of A into tile A_tb using
...
12: for k = 0, 8, 16, ..., K
13:       asm ("cp.async.ca.shared.global [%0], [%1], 16;\n" :
              : A_tb[stage][tid],
                "l"(A[tid]));
14:       asm ("cp.async.ca.shared.global [%0], [%1], 16;\n" :
              : B_tb[stage][tid],
                "l"(B[tid]));
15:    e1T_A = A_t[0] + ... + A_t[m_w - 1]
16:    Be1 = B_t[0] + ... + B_t[n_w - 1]
17:    e2T_A = A_t[0] + 2 * A_t[1] + ... + m_w * A_t[m_w - 1]
18:    Be2 = B_t[0] + 2 * B_t[1] + ... + n_w * B_t[n_w - 1]
19:    for i = 0, 1, ..., m_w - 1
20:       for j = 0, 1, ..., n_w - 1
21:          asm volatile("mma.sync.aligned.m8n8k4.row.col.f64.f64.f64.f64
                 {%0,%1}, {%2}, {%3}, {%4,%5};\n"
                 : "=d"(c[i][j][0]), "=d"(c[i][j][1])
                 : "d"(a[0]), "d"(b[0]), "d"(c[i][j][0]), "d"(c[i][j][1]));
22:    asm volatile("mma.sync.aligned.m8n8k4.row.col.f64.f64.f64.f64
          {%0,%1}, {%2}, {%3}, {%4,%5};\n"
                 : "=d"(e1TC[0]), "=d"(e1TC[1])
                 : "d"(e1T_A), "d"(Be1), "d"(e1TC[0]), "d"(e1TC[1]);
23:    asm volatile("mma.sync.aligned.m8n8k4.row.col.f64.f64.f64.f64
          {%0,%1}, {%2}, {%3}, {%4,%5};\n"
                 : "=d"(e1TC[0]), "=d"(e1TC[1])
                 : "d"(e1T_A), "d"(Be2), "d"(e1TC[0]), "d"(e1TC[1]);
24:    asm volatile("mma.sync.aligned.m8n8k4.row.col.f64.f64.f64.f64
          {%0,%1}, {%2}, {%3}, {%4,%5};\n"
                 : "=d"(e1TC[0]), "=d"(e1TC[1])
                 : "d"(e2T_A), "d"(Be1), "d"(e1TC[0]), "d"(e1TC[1]);
25:    if k % 256 == 0:
26:       for i = 0, 1, ..., m_w - 1:
27:          for j = 0, 1, ..., n_w - 1:
28:             e1TC[0] -= c[i][j][0]
29:             e1TC[1] -= c[i][j][1]
30:       if e1TC[0] + e1TC[1] > epsilon:
31:          correction
32: ...
```

Optimized Dsitance Compute & Cluster Assign Kernel w/ FT

Figure 6: Pseudocode: K-Means w/ FT

V. Performance Evaluation

We evaluate our K-means on two NVIDIA GPUs, a Tesla Turing T4 and a 40GB A100-PCIE GPU. The Tesla T4 GPU is connected to a node with two 16-core Intel Xeon Silver 4216 CPUs, whose boost frequency is up to 3.2 GHz. The associated CPU main memory system has a capacity of 512 GB at 2400 MHz. The A100 GPU is connected to a node with one 64-core AMD EPYC 7763 CPU with a boost frequency of 3.5 GHz. We compile programs using CUDA 11.6 with the −O3 optimization flag on the Tesla T4 machine, and using CUDA 12.0 on the A100 machine. A100 has a peak computational performance of 19.5 TFLOPS for single precision and 9.7 TFLOPS for double precision. The memory bandwidth is 1.55 TB/s. T4 has a peak performance of 8.1 TFLOPS for single precision and a peak performance of 0.253 TFLOPS for double precision. The bandwidth of T4 is 320 GB/s. We first demonstrate the benchmark result between FT K-Means without fault tolerance and cuML for FP32 and FP64 on A100. Next, we evaluate the FT K-Means under fault tolerance. Then, we benchmark FT K-Means under error injections with cuML, Wu's ABFT, and Jack's ABFT. Finally, we present the same performance evaluation on the T4 GPU. All experimental results are averaged over ten trials.

A. *Benchmarking K-Means without Fault Tolerance*

In this subsection, we evaluate the performance of FT K-Means w/o checksum.

1) Step-wise optimizations for K-means: Figure 7 demonstrates how our K-Means distance kernel is optimized from 5% to 182% of cuML stepwise. The performance is measured with

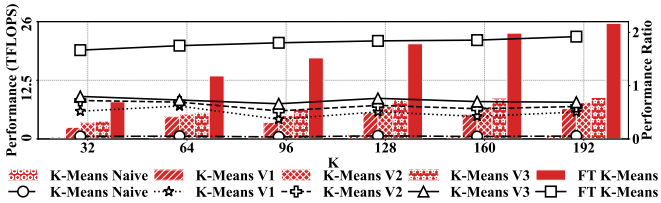

Figure 7: K-Means w/o FT stepwise optimizations on A100, FP32 (N=131072, N=128)

GFLOPS (bar plot, the left y-axis) and the performance ratio with respect to cuML (line chart, the right y-axis). Without using any optimizations, the K-Means Naive obtains a performance of 482 GFLOPS. Next K-Means V1 employs GEMM to K-Means distance calculation. The performance improved from 482 GFLOPS to 4662 GFLOPS. And then, K-Means V2 adds kernel fusion to both thread and threadblock level, reducing memory operations. The performance is improved to 5902 GFLOPS. After that, we apply threadblock level broadcast to further reduce memory bound. The performance of K-Means V3 achieves 6916 GFLOPS. And tensor core enabled and parameter selection, we finally acheive 17686 GFLOPS, which exceeded the performance of cuML (9676 GFLOPS).

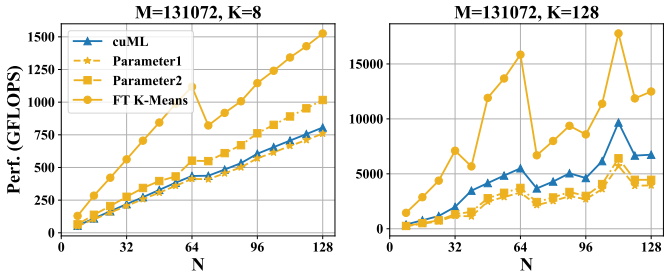

Figure 8: FP32 precision comparison of K-Means performance at distance step without fault tolerance with FT K-Means, Selected parameters and cuML on an A100 GPU, with M and K fixed.

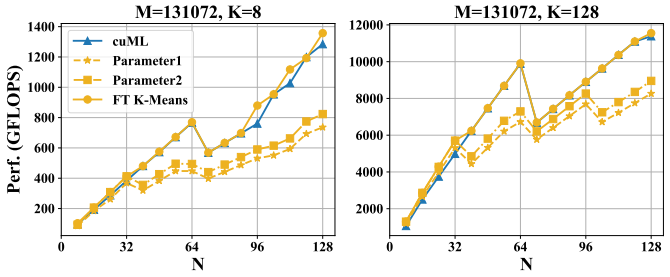

Figure 9: Double precision comparison of K-Means performance at distance step without fault tolerance with FT K-Means, Selected parameters and cuML on an A100 GPU, with M and K fixed.

2) Performance evaluation with M and K fixed: Figure 8 and 9 demonstrate a performance evaluation in distance step between FT K-Means, two selected parameters relative to cuML (all without fault tolerance) in FP32 and FP64 precision when M and K are fixed. We tested the performance of different methods under two values of K, $K = 8$ and $K = 128$. These are cases representing two situations: one with very few dimensions of data and the other one with relatively more dimensions of data. The selected parameters are labeled Parameter1 and Parameter2, and they are chosen based on experience. And the same parameter name refers to different parameters in FP32 and FP64, but their performance is similar. For both FP32 and FP64 precision, parameters selected through experience cannot achieve good performance. Parameter 1 is always slower than cuML, with an average overhead of 15%. And for parameter 2, it slightly exceeded the performance of cuML when $K = 8$ in FP32, and it achieves the performance of cuML in some data points where N is small. However, its overall performance is still 5% slower than cuML. Using code generation strategy, we achieved 235% speedup compared to cuML under FP32 precision, and the gain in performance is significant even when K is relatively larger. However, under FP64 precision, improvements in performance is minimal, with a overall speedup of 4% in these two cases. And the curves of cuML and FT K-Means are almost coincident in a large portion of data points.

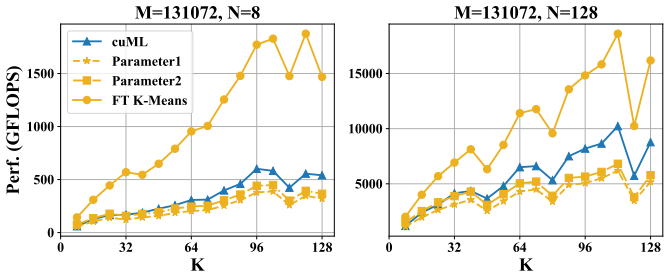

Figure 10: FP32 precision comparison of K-Means performance at distance step without fault tolerance with FT K-Means, Selected parameters and cuML on an A100 GPU, with M and N fixed

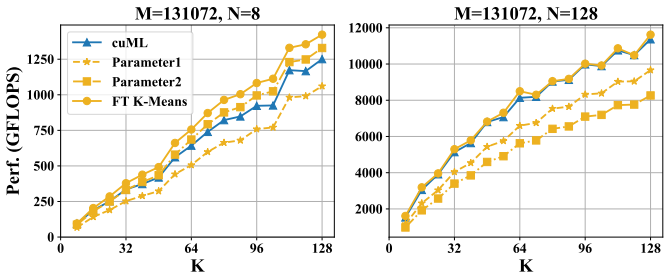

Figure 11: FP64 precision comparison of K-Means performance at distance step without fault tolerance with FT K-Means, Selected parameters and cuML on an A100 GPU, with M and N fixed

3) Performance evaluation with M and N fixed: Figure 10 and 11 offer a performance evaluation in distance step between FT K-Means, two selected parameters relative to cuML (all without fault tolerance) in FP32 and FP64 precision when M and N are fixed. The main parameter settings are similar to the previous section, and $N = 8$ and $N = 128$ indicate fewer clustering centroids and relatively more clustering centroids. For both FP32 and FP64 precision, parameters selected through experience cannot achieve good performance. Parameter 1 has an average overhead of 30%. And parameter 2 exceeded the performance of cuML when K is small in some FP32 cases, and outperforms cuML in FP64 when $N = 8$ (1.03x speedup). but its overall the performance is still 15% slower than cuML. Using code generation strategy, we obtained 239% speedup compared to cuML under FP32 precision, which is similar to fixing M and K. Under FP64 precision, improvements in

performance is higher than previous section, which is 8% in these two cases. And the speedup in small N is relatively considerable (15%).

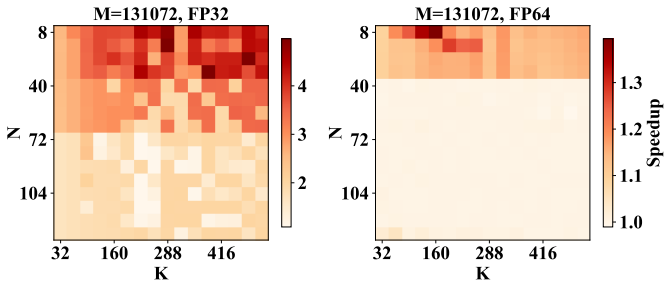

Figure 12: Speedup of FT K-Means compared to cuML on FP32 and FP64

4) Overall performance evaluation: Figure 12 illustrate an overall comparison between our code generation method and cuML K-means on both FP32 and FP64. For FP32. Our approach shows significant improvement in performance, with a maximum speedup of 4.55x and an average speedup of 2.49x. From the perspective of dimension N, the figure indicates a clear trend that the performance improvement diminishes. Furthermore, $N = 64$ serves as a clear threshold, beyond which speedup essentially decreases to below 2.0x. Such trend becomes even more manifest in the case of FP64. The average speedup is only 1.04x, with a maximum speedup of 1.39x. When N exceeds 32, the performance of our method drops to almost identical as cuML. Meanwhile, there is no apparent trend observed in dimension K for both FP32 and FP64.

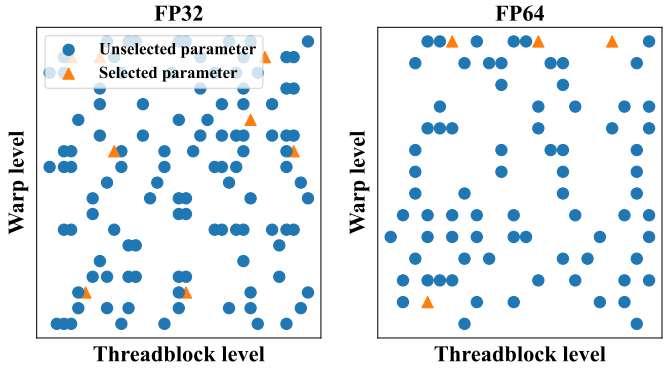

Figure 13: Parameter selection in threadblock and warp level for FP32 and FFP64

5) Evaluation of parameter selection: The experimental results from previous sections motivate us to analyze the parameters chosen for our code generation method for each data point. For the data size of Figure 12, we generated 120 groups of FP32 parameters and 80 groups of FP64 parameters to be selected from. However, only 7 groups of FP32 parameters and 4 groups of FP64 parameters are actually chosen in at least one data point. And all threadblock and warp level parameters are shown in Figure 13. And thread level parameters are fixed (FP32: 16,8,4 FP64: 8,8,4) owing to the constant size of tensor core. Moreover, we depicted figure 14 to illustrate the relationship between each data point and its corresponding selected parameters. We will analyze in detail the relationship between the optimal parameters generated by the code generator and the cuML parameters in next section.

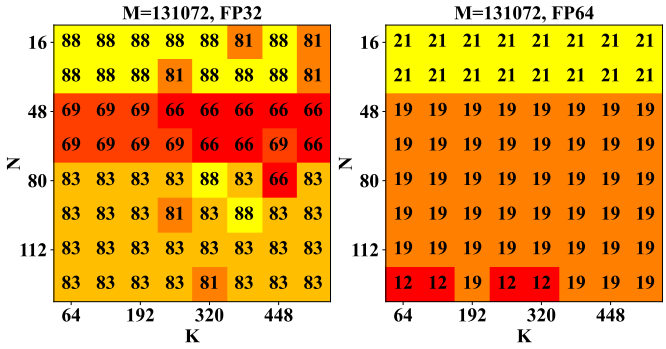

Figure 14: The selected parameter number in FP32 and FP64

6) Detailed analyse of parameters: For FP32, as we can see from figure 14, the data points can be divided into three parts: $N \leq 32$, $32 < N \leq 64$ and $64 < N$. When $N \leq 32$, one of the optimal parameters is parameter 88. As table I indicates, compared to cuML, it has a bigger threadblock and warp size on M direction and smaller size on N direction. And that is reasonable due to small N size in these data points. threadblock.N = 256 is too big that more than 87.5% of the threadblock size is containing blank data, so the occupancy is very low. Furthermore, warp.N of cuML, which is 64, exceeds the size of N. Therefore the improvement of our new parameters is significant. As N increases, i.e. $32 < N \leq 64$, the warp size of cuML becomes justified, and our parameter (parameter 69) sticks to this size. However, the threadblock size remains too big on N direction, so a more balanced version of threadblock $64, 128, 16$ outperforms. When N keeps increasing, although the occcupancy of cuML achieves a reasonable scope, a more balanced threadblock size, e.g parameter 83 can reduce the overall amount of data movement form GPU memory to shared memory, and therefore improve performance.

For FP64, the data points only have two main parts: $N \leq 32$ and $32 < N$. When N is relatively small, parameter21 has advantage in its small Threadblock.N, and increases occupancy. And as N increases, the parameters which are balanced in M and N direction has better performance. And as shown in table I, our parameter 19 is actually identical to cuML's parameter, which demonstrates that the best parameter choice is cuML's parameter in these cases. This illustrates an interesting phenomenon: the parameter choices in FP32 are much more than which in FP64. Moreover, there is also greater potential for performance improvement in FP32 than FP64. And there are several reasons for this.

First, cutlass enables TF32 in the FP32 GEMM kernel, which improves the processing speed of tensor cores. Therefore, the overhead of data movement and epilogue (row-wise reduction) becomes more critical, resulting in greater potential for alternative parameters. Second, the memory alignment requirement for FP64 is more strict than FP32 and is fixed to 1 in cutlass's implementation. So the degree of vectorization for FP64 is lower. So a balanced data fetching pattern (Threadblock.M = Threadblock.N) is crucial for increasing performance. Third,

FP32			
ID	Threadblock	Warp	Thread
88	256, 32, 16	64, 32, 16	16, 8, 4
69	128, 64, 16	32, 64, 16	16, 8, 4
83	64, 128, 16	64, 32, 16	16, 8, 4
cuML	32, 256, 16	32, 64, 16	16, 8, 4

FP64			
ID	Threadblock	Warp	Thread
21	128, 32, 16	32, 32, 16	8, 8, 4
19	64, 64, 16	32, 32, 16	8, 8, 4
cuML	64, 64, 16	32, 32, 16	8, 8, 4

TABLE I: Partial parameters of FT K-Means for FP32 and FP64

the thread level parameters of FP64 is smaller than which of FP32. Therefore even when N is relatively small, it is easier for a general parameter to obtain high occupancy.

B. FT K-Means with Fault Tolerance

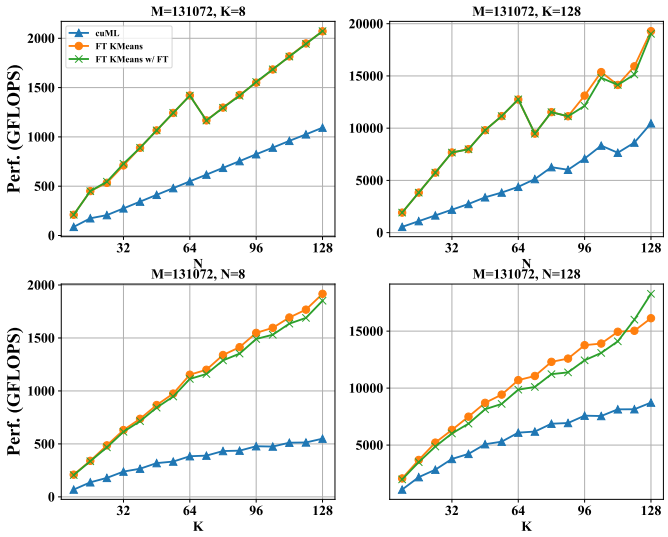

Figure 15: FT K-Means with fault tolerance on A100: FP32.

As shown in Figure 15, the experimental evaluation of the FT K-Means algorithm with fault tolerance demonstrates a remarkably low overhead across various configurations. Specifically, in configurations with $K = 8$ clusters, the overhead was maintained at a minimal -0.24%, illustrating the negligible impact of integrating fault tolerance mechanisms. Even when the number of clusters was significantly increased to $K = 128$, the overhead remained consistently low at 1.93%. For fixed N scenarios, the overhead was even lower at 0.96%. These results highlight the efficiency of the FT K-Means algorithm's fault tolerance, which effectively minimizes additional computational burdens while maintaining robustness across diverse input shapes.

In Figure 16, FT K-Means with fault tolerance presents an average overhead of 13% for double precision. For small number of clusters ($K = 8$), the overhead is 7.9%, ranging from 4.6% to 12.45%. It suggests that the algorithm maintains consistent performance even when computational precision requirements are increased. When the number of clusters

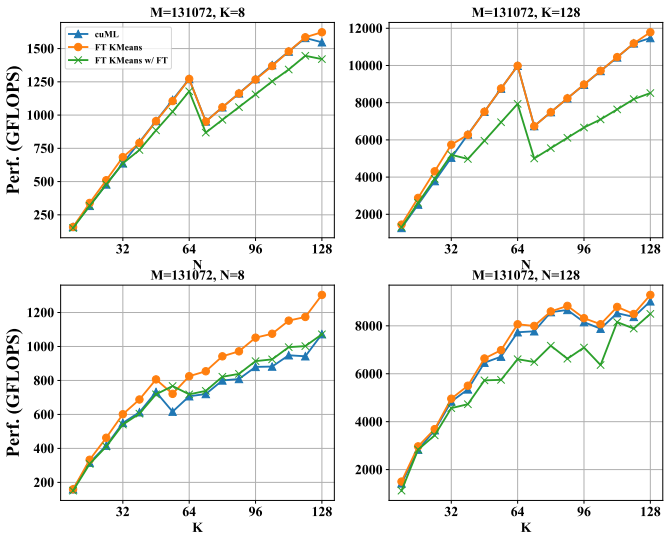

Figure 16: Benchmark FT K-Means with fault tolerance: FP64.

increased to $K = 128$, the overhead results in 20%, indicating the performance bottleneck changes to computing. For input shapes of fixed N, $N = 8$ and $N = 128$, the overhead was reduced to 0.8897%. These results underscore the FT K-Means algorithm's ability to efficiently manage fault tolerance with minimal overhead across a variety of input shapes and precision settings.

C. FT K-Means under Error Injections

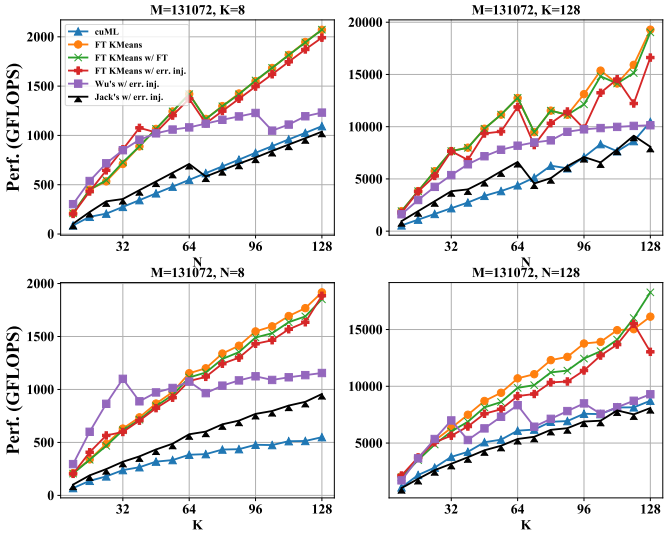

Figure 17: Benchmark FT K-Means with error injection: FP32.

As illustrated in Figure 17, the experimental evaluation of the FT K-Means algorithm under error injection reveals a minimal average overhead of approximately 2.36%, demonstrating the algorithm's efficiency in handling errors with minimal additional computational cost. The results indicate robust performance across various input shapes, with overhead percentages ranging from a slight reduction of about -0.93% to a modest increase up to 9.49%. This low overhead highlights the effectiveness of the fault tolerance mechanisms integrated into the FT-KMeans, ensuring correctness even in the presence of injected errors. Jack's FT scheme results in a 100 % overhead due to

time redundant recomputation. Wu's FT scheme introduces an overhead of 30% due to a suboptimal GEMM baseline without using the asynchronous memory copy.

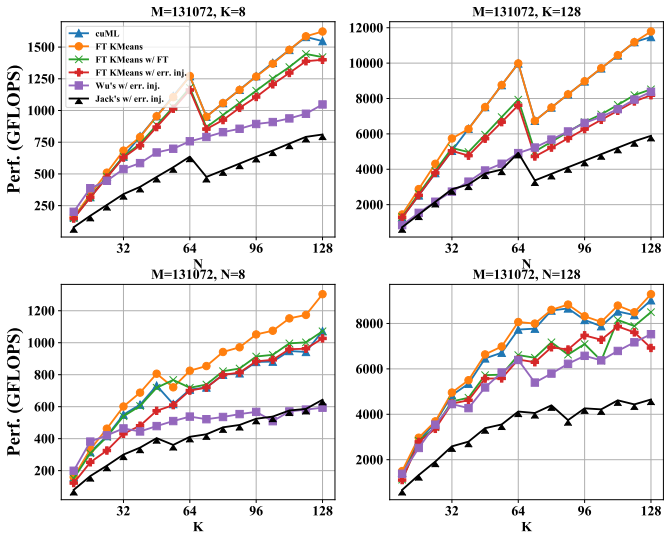

Figure 18: Benchmark FT K-Means with error injection: FP64.

In Figure 18, the evaluation of the FT-KMeans algorithm with error injection in a 64-bit floating point (fp64) environment shows a low average overhead of approximately 9.21%s. The data demonstrates that configurations with fewer features (N=8 and N=128) exhibit low overheads of 0.79% and 0.84% respectively, highlighting the algorithm's effective fault tolerance mechanisms which ensure minimal performance degradation even in high-precision settings. Meanwhile, fixed K scenarios, K=8 and K=128, show higher overheads of 10.12% and 24.07%, indicating increased computational complexity under fault conditions.

D. Performance Evaluation on T4

Figure 19 demonstrate a performance evaluation in distance step between FT K-Means, two selected parameters relative to cuML (all without fault tolerance) in FP32 precision when M and K are fixed. The definitions of Parameter1 and Parameter2 are similar to our evaluation on A100. And they are consistent with the values on A100 to the greatest extent. However, they have better performance compared to cuML's parameter in this architecture, with speed up of 184% and 208%. Using code generation strategy, we achieved 413% speedup compared to cuML under FP32 precision, and the gain in performance is significant even when K is relatively larger.

Figure 20 offers a performance evaluation in distance step between FT K-Means, two selected parameters relative to cuML (all without fault tolerance) in FP32 precision when M and N are fixed. The main parameter settings are similar to the previous section, and $N = 8$ and $N = 128$ indicate fewer clustering centroids and relatively more clustering centroids. The selected parameters acheive 183% and 206% speed up against cuML. Using code generation strategy, we obtained 381% speedup compared to cuML under FP32 precision, which is similar to fixing M and K.

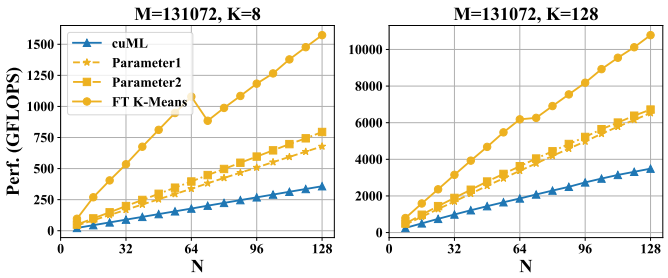

Figure 19: FP32 precision comparison of K-Means performance at distance step without fault tolerance with FT K-Means, Selected parameters and cuML on a T4 GPU, with M and K fixed.

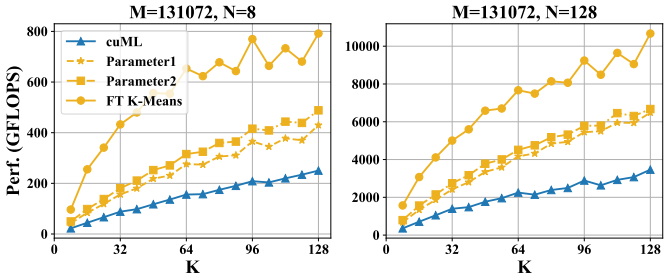

Figure 20: FP32 precision comparison of K-Means performance at distance step without fault tolerance with FT K-Means, Selected parameters and cuML on a T4 GPU, with M and N fixed

Figure 21 benchmarks the performance of FT K-Means with or without fault tolerance for single precision. FT K-Means shows an average overhead of 18% with fault tolerance and 30% under error injection. Compared with Jack's ABFT, FT K-means has a 60% improvement compared to the combination of Jack's ABFT and time-redundant recomputation. FT K-Means shows a similar performance compared to Wu's ABFT scheme.

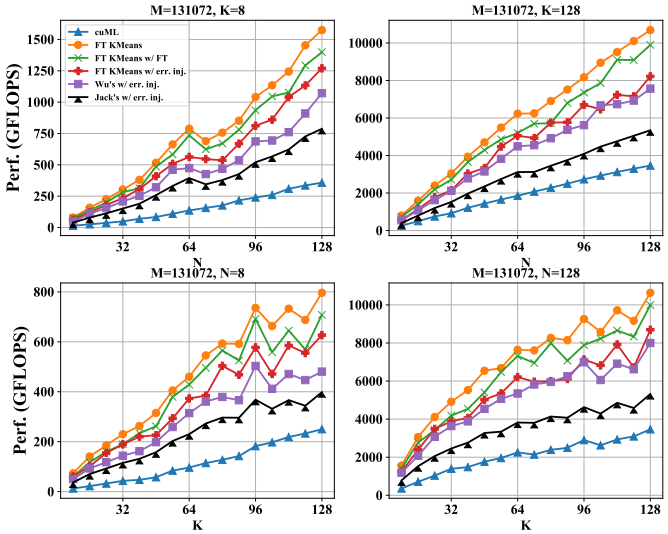

Figure 21: Benchmark FT K-Means with error injection on T4: FP32.

VI. Conclusion

In this paper, we introduce FT K-Means, a high-performance GPU-accelerated implementation of K-Means with online fault tolerance. We first present a stepwise optimization strategy that achieves competitive performance compared to NVIDIA's cuML library. We further improve FT K-Means with a

template-based code generation framework that supports different data types and adapts to different input shapes. A novel warp-level tensor-core error correction scheme is proposed to address the failure of existing fault tolerance methods due to memory asynchronization during copy operations. Our experimental evaluations on NVIDIA T4 and A100 GPUs demonstrate that FT K-Means without fault tolerance outperforms cuML's K-Means implementation, showing a performance increase of 10%-300% in scenarios involving irregular data shapes. Moreover, the fault tolerance feature of FT K-Means introduces only an overhead of 11%, maintaining robust performance even with tens of errors injected per second.

REFERENCES

[1] X. Wu, V. Kumar, J Ross Quinlan, J. Ghosh, Q. Yang, H. Motoda, G. J. McLachlan, A. Ng, B. Liu, P. S. Yu, *et al.*, "Top 10 algorithms in data mining," *Knowledge and information systems*, vol. 14, pp. 1–37, 2008.

[2] A. Gersho and R. M. Gray, *Vector quantization and signal compression*. Springer Science & Business Media, 2012, vol. 159.

[3] U. M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, "Advances in knowledge discovery and data mining," American Association for Artificial Intelligence, 1996.

[4] R. O. Duda, P. E. Hart, *et al.*, *Pattern classification and scene analysis*. Wiley New York, 1973, vol. 3.

[5] R. R. Lutz, "Analyzing software requirements errors in safety-critical, embedded systems," in *[1993] Proceedings of the IEEE International Symposium on Requirements Engineering*, IEEE, 1993, pp. 126–133.

[6] M. Nicolaidis, "Time redundancy based soft-error tolerance to rescue nanometer technologies," in *Proceedings 17th IEEE VLSI Test Symposium (Cat. No. PR00146)*, IEEE, 1999, pp. 86–94.

[7] J.-C. Laprie, "Dependable computing and fault-tolerance," *Digest of Papers FTCS-15*, pp. 2–11, 1985.

[8] D. Oliveira, L. Pilla, N. DeBardeleben, S. Blanchard, H. Quinn, I. Koren, P. Navaux, and P. Rech, "Experimental and analytical study of Xeon Phi reliability," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ACM, 2017, p. 28.

[9] R. Lucas, J. Ang, K. Bergman, S. Borkar, W. Carlson, L. Carrington, G. Chiu, R. Colwell, W. Dally, J. Dongarra, *et al.*, "DOE advanced scientific computing advisory subcommittee (ASCAC) report: Top ten exascale research challenges," USDOE Office of Science (SC)(United States), Tech. Rep., 2014.

[10] T. C. May and M. H. Woods, "Alpha-particle-induced soft errors in dynamic memories," *IEEE Transactions on Electron Devices*, vol. 26, no. 1, pp. 2–9, 1979.

[11] R. Baumann, "Soft errors in commercial semiconductor technology: Overview and scaling trends," *IEEE 2002 Reliability Physics Tutorial Notes, Reliability Fundamentals*, vol. 7, 2002.

[12] A. Geist, "Supercomputing's monster in the closet," *IEEE Spectrum*, vol. 53, no. 3, pp. 30–35, 2016.

[13] P. H. Hochschild, P. Turner, J. C. Mogul, R. Govindaraju, P. Ranganathan, D. E. Culler, and A. Vahdat, "Cores that don't count," in *Proceedings of the Workshop on Hot Topics in Operating Systems*, 2021, pp. 9–16.

[14] H. D. Dixit, S. Pendharkar, M. Beadon, C. Mason, T. Chakravarthy, B. Muthiah, and S. Sankar, "Silent data corruptions at scale," *arXiv preprint arXiv:2102.11245*, 2021.

[15] J. C. Phillips, R. Braun, W. Wang, J. Gumbart, E. Tajkhorshid, E. Villa, C. Chipot, R. D. Skeel, L. Kale, and K. Schulten, "Scalable molecular dynamics with NAMD," *Journal of computational chemistry*, vol. 26, no. 16, pp. 1781–1802, 2005.

[16] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., Curran Associates, Inc., 2019, pp. 8024–8035.

[17] D. Tao, S. Di, X. Liang, Z. Chen, and F. Cappello, "Improving performance of iterative methods by lossy checkponting," in *Proceedings of the 27th international symposium on high-performance parallel and distributed computing*, 2018, pp. 52–65.

[18] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, *TensorFlow: Large-scale machine learning on heterogeneous systems*, Software available from tensorflow.org, 2015. [Online]. Available: https://www.tensorflow.org/.

[19] D. Hakkarinen, P. Wu, and Z. Chen, "Fail-stop failure algorithm-based fault tolerance for cholesky decomposition," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 5, pp. 1323–1335, 2014.

[20] Z. Chen and J. Dongarra, "A scalable checkpoint encoding algorithm for diskless checkpointing," in *2008 11th IEEE High Assurance Systems Engineering Symposium*, IEEE, 2008, pp. 71–79.

[21] Z. Chen, "Extending algorithm-based fault tolerance to tolerate fail-stop failures in high performance distributed environments," in *2008 IEEE International Symposium on Parallel and Distributed Processing*, IEEE, 2008, pp. 1–8.

[22] S. Mitra, P. Bose, E. Cheng, C.-Y. Cher, H. Cho, R. Joshi, Y. M. Kim, C. R. Lefurgy, Y. Li, K. P. Rodbell, *et al.*, "The resilience wall: Cross-layer solution strategies," in *Proceedings of Technical Program-2014 International Symposium on VLSI Technology, Systems and Application (VLSI-TSA)*, IEEE, 2014, pp. 1–11.

[23] C.-Y. Cher, M. S. Gupta, P. Bose, and K. P. Muller, "Understanding soft error resiliency of blue gene/q compute chip through hardware proton irradiation and software fault injection," in *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, IEEE, 2014, pp. 587–596.

[24] J. Dongarra, P. Beckman, T. Moore, P. Aerts, G. Aloisio, J.-C. Andre, D. Barkai, J.-Y. Berthou, T. Boku, B. Braunschweig, *et al.*, "The international exascale software project roadmap," *International Journal of High Performance Computing Applications*, vol. 25, no. 1, pp. 3–60, 2011.

[25] J. Calhoun, M. Snir, L. N. Olson, and W. D. Gropp, "Towards a more complete understanding of SDC propagation," in *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*, ACM, 2017, pp. 131–142.

[26] M. Snir, R. W. Wisniewski, J. A. Abraham, S. V. Adve, S. Bagchi, P. Balaji, J. Belak, P. Bose, F. Cappello, B. Carlson, *et al.*, "Addressing failures in exascale computing," *The International Journal of High Performance Computing Applications*, vol. 28, no. 2, pp. 129–173, 2014.

[27] G. Li, S. K. S. Hari, M. Sullivan, T. Tsai, K. Pattabiraman, J. Emer, and S. W. Keckler, "Understanding error propagation in deep learning neural network (dnn) accelerators and applications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2017, pp. 1–12.

[28] S. Taamneh, A. Qawasmeh, and A. H. Aljammal, "Parallel and fault-tolerant k-means clustering based on the actor model," *Multiagent and Grid Systems*, vol. 16, no. 4, pp. 379–396, 2020.

[29] N. Oh, P. P. Shirvani, and E. J. McCluskey, "Error detection by duplicated instructions in super-scalar processors," *IEEE Transactions on Reliability*, vol. 51, no. 1, pp. 63–75, 2002.

[30] N. Oh, P. P. Shirvani, and McCluskey, "Control-flow checking by software signatures," *IEEE transactions on Reliability*, vol. 51, no. 1, pp. 111–122, 2002.

[31] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August, "Swift: Software implemented fault tolerance," in *Proceedings of the international symposium on Code generation and optimization*, IEEE Computer Society, 2005, pp. 243–254.

[32] J. Yu, M. J. Garzaran, and M. Snir, "Esoftcheck: Removal of non-vital checks for fault tolerance," in *2009 International Symposium on Code Generation and Optimization*, IEEE, 2009, pp. 35–46.

[33] Z. Chen, A. Nicolau, and A. V. Veidenbaum, "SIMD-based soft error detection," in *Proceedings of the ACM International Conference on Computing Frontiers*, ACM, 2016, pp. 45–54.

[34] Z. Chen and J. Dongarra, "Algorithm-based fault tolerance for fail-stop failures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 19, no. 12, pp. 1628–1641, 2008.

[35] C. Ding, C. Karlsson, H. Liu, T. Davies, and Z. Chen, "Matrix multiplication on gpus with on-line fault tolerance," in *2011 IEEE Ninth International Symposium on Parallel and Distributed Processing with Applications*, IEEE, 2011, pp. 311–317.

[36] Y. Zhai, E. Giem, Q. Fan, K. Zhao, J. Liu, and Z. Chen, "Ft-blas: A high performance blas implementation with online fault tolerance," in *Proceedings of the ACM International Conference on Supercomputing*, 2021, pp. 127–138.

[37] J. Kosaian and K. Rashmi, "Arithmetic-intensity-guided fault tolerance for neural network inference on gpus," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021, pp. 1–15.

[38] S. Wu, Y. Zhai, J. Liu, J. Huang, Z. Jian, B. Wong, and Z. Chen, "Anatomy of high-performance gemm with online fault tolerance on gpus," in *Proceedings of the 37th International Conference on Supercomputing*, 2023, pp. 360–372.

[39] J. M. Bird, M. K. Peters, T. Z. Fullem, M. J. Tostanoski, T. F. Deaton, K. Hartojo, and R. E. Strayer, "Neutron induced single event upset (seu) testing of commercial memory devices with embedded error correction codes (ecc)," in *2017 IEEE Radiation Effects Data Workshop (REDW)*, IEEE, 2017, pp. 1–8.

[40] G. E. Fagg and J. J. Dongarra, "Ft-mpi: Fault tolerant mpi, supporting dynamic applications in a dynamic world," in *European parallel virtual machine/message passing interface users' group meeting*, Springer, 2000, pp. 346–353.

[41] D. Binder, E. C. Smith, and A. Holman, "Satellite anomalies from galactic cosmic rays," *IEEE Transactions on Nuclear Science*, vol. 22, no. 6, pp. 2675–2680, 1975.

[42] E. Petersen, R Koga, M. Shoga, J. Pickel, and W. Price, "The single event revolution," *IEEE Transactions on Nuclear Science*, vol. 60, no. 3, pp. 1824–1835, 2013.

[43] P. Wu and Z. Chen, "Ft-scalapack: Correcting soft errors on-line for ScaLAPACK Cholesky, QR, and LU factorization routines," in *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*, ACM, 2014, pp. 49–60.

[44] M. Turmon, R. Granat, and D Katz, "Software-implemented fault detection for high-performance space applications," in *Proceeding International Conference on Dependable Systems and Networks. DSN 2000*, IEEE, 2000, pp. 107–116.

[45] S. A. Shalom, M. Dash, and M. Tue, "Efficient k-means clustering using accelerated graphics processors," in *Data Warehousing and Knowledge Discovery: 10th International Conference, DaWaK 2008 Turin, Italy, September 2-5, 2008 Proceedings 10*, Springer, 2008, pp. 166–175.

[46] R. Farivar, D. Rebolledo, E. Chan, and R. H. Campbell, "A parallel implementation of k-means clustering on gpus.," in *Pdpta*, vol. 13, 2008, pp. 212–312.

[47] Y. Li, K. Zhao, X. Chu, and J. Liu, "Speeding up k-means algorithm by gpus," *Journal of Computer and System Sciences*, vol. 79, no. 2, pp. 216–229, 2013.

[48] C. Lutz, S. Breß, T. Rabl, S. Zeuch, and V. Markl, "Efficient and scalable k-means on gpus," *Datenbank-Spektrum*, vol. 18, pp. 157–169, 2018.

[49] M. Kruliš, J. Lokoč, and T. Skopal, "Efficient extraction of clustering-based feature signatures using gpu architectures," *Multimedia Tools and Applications*, vol. 75, pp. 8071–8103, 2016.

[50] S. Cuomo, V. De Angelis, G. Farina, L. Marcellino, and G. Toraldo, "A gpu-accelerated parallel k-means algorithm," *Computers & Electrical Engineering*, vol. 75, pp. 262–274, 2019.

[51] Y. Ding, Y. Zhao, X. Shen, M. Musuvathi, and T. Mytkowicz, "Yinyang k-means: A drop-in replacement of the classic k-means with consistent speedup," in *International conference on machine learning*, PMLR, 2015, pp. 579–587.

[52] J. Nelson and R. Palmieri, "Don't forget about synchronization! a case study of k-means on gpu," in *Proceedings of the 10th International Workshop on Programming Models and Applications for Multicores and Manycores*, 2019, pp. 11–20.

[53] M. Kruliš and M. Kratochvíl, "Detailed analysis and optimization of cuda k-means algorithm," in *Proceedings of the 49th International Conference on Parallel Processing*, 2020, pp. 1–11.

[54] K. Ouyang, V. Tran, J. Liu, B. M. Wong, and Z. Chen, "Kf k-means: A high performance k-means implementation using kernel fusion," in *2023 IEEE International Conference on Big Data (BigData)*, IEEE, 2023, pp. 121–127.

[55] S. Raschka, J. Patterson, and C. Nolet, "Machine learning in python: Main developments and technology trends in data science, machine learning, and artificial intelligence," *Information*, vol. 11, no. 4, p. 193, 2020.

[56] K.-H. Huang and J. A. Abraham, "Algorithm-based fault tolerance for matrix operations," *IEEE transactions on computers*, vol. 100, no. 6, pp. 518–528, 1984.