Performance Evaluation of k-means using CPU and GPU with oneAPI and OpenMP for Network Intrusion Detection

Laura Caetano Costa, Luiz Fernando Antunes da Silva Frassi, Henrique Cota de Freitas

Computer Architecture and Parallel Processing Team (CArT)
Department of Computer Science
Pontifícia Universidade Católica de Minas Gerais (PUC Minas)
30535-901, Belo Horizonte, Brazil

laura.costa3141@gmail.com, luizfernandoe30@gmail.com, cota@pucminas.br

Abstract. *This article describes the application of the k-means algorithm for network intrusion detection, evaluating both sequential and parallel CPU and GPU versions using oneAPI and OpenMP. Based on a network intrusion dataset (CIC-IDS2017), which contains millions of instances of modern attacks, the experiments showed that the parallel versions achieved significant performance gains over the sequential version. The good scalability of the CPU solutions and GPU acceleration stand out, with oneAPI performing slightly better than OpenMP. The results reinforce that parallelization is essential for applying k-means in contexts where the data volume is large.*

1. Introduction

The increase in network traffic and complexity intensifies the need for efficient methods to detect cyberattacks (Andrade Maciel et al., 2020). In this context, *k-means* (MacQueen, 1967) stands out for its simplicity and ability to cluster patterns and identify anomalies in large volumes of data. However, its application to large datasets is computationally expensive and requires performance optimization strategies.

Despite the relevance of *k-means* in detecting network attacks, its timely application in high-traffic environments still represents a challenge. Processing time, especially in sequential implementations, can compromise the effectiveness of detection systems. Thus, this work aims to develop and evaluate different execution strategies: sequential and CPU- and GPU-parallel versions using the parallel programming models oneAPI (oneAPI, 2025) and OpenMP (OpenMP, 2025), which are publicly available[1].

This paper presents a detailed experimental evaluation as a contribution, comparing the performance of *k-means* across different architectures and programming models. Aspects such as execution time and scalability are analyzed. Furthermore, it provides quantitative evidence on the benefits obtained with parallelization. The results enable the selection of the most efficient approach for real-world network attack detection scenarios,

---

Laura and Luiz are undergraduate students in Computer Science at PUC Minas.

The authors would like to thank the National Council for Scientific and Technological Development of Brazil (CNPq - Codes 311697/2022-4 and 402837/2024-0), the Foundation for Research Support of Minas Gerais State (FAPEMIG - Code APQ-05058-23), and PUC Minas FIP 2025/32469.

[1] https://github.com/cart-pucminas/parallel-ML

contributing to the development of more agile and scalable security systems. To validate the experiments, we utilized the CICIDS2017 dataset (Sharafaldin et al., 2018), which provides realistic network traffic data.

This paper is organized as follows: Section 2 presents related work. Section 3 describes the methodology adopted, the data used, and the testing process. Section 4 details the implementation of the k-means algorithm's parallelization. Section 5 presents the results obtained, with performance metrics in terms of execution time and accuracy. Finally, Section 6 presents the study's conclusions.

2. Related Work

A large number of studies have explored the parallelization of the *k-means* algorithm on various platforms and in different application contexts.

Andrade Maciel et al. (2020) proposed a reconfigurable architecture for the *k-means* and K-Modes algorithms applied to intrusion detection. Implemented in VHDL and validated on an Intel Cyclone IV-E FPGA, the solution showed substantial performance gains compared to a Xeon CPU, with reductions of up to 97% in clock cycles and up to 99% in energy consumption. Furthermore, the work emphasized the flexibility of the architecture in handling both numerical and categorical data, which is fundamental in real IDS scenarios.

Yin and Zhang (2017) developed an improved version of *k-means* that introduced the automatic selection of initial centroids through the DD algorithm, the use of information entropy for attribute weighting, and PCA for dimensionality reduction. To handle large-scale datasets, the algorithm was parallelized in a cloud computing environment using the MapReduce model, enabling greater scalability. The experiments showed superior results compared to traditional *k-means*, achieving accuracy rates of up to 100% on datasets such as KDD Cup99, in addition to successful application in content-based image retrieval.

Pathak and Ananthanarayana (2012) presented a multi-threaded *k-means* approach for intrusion detection. Unlike traditional methods, each thread was configured to analyze subsets of attributes specific to certain types of attacks (DoS, Probe, R2L, U2R), while a coordinating thread consolidated the results. This division reduced computational cost and increased efficiency, achieving a detection rate of 99.18% and reducing the false positive rate to 1.34%. In addition, the authors emphasized that the proposed approach is flexible enough to detect new types of attacks. When it is difficult to precisely determine the category of a packet, the system can flag it as belonging to multiple attack types simultaneously, ensuring that ambiguous cases are handled manually, whereas traditional methods might incorrectly classify them into a single category.

The article by Bhimani et al. (2015) investigates ways to optimize the K-Means algorithm, which is widely used in applications such as image processing and data mining, but has high computational costs in large datasets. The authors compare three parallelization approaches: shared memory with OpenMP, distributed memory with MPI, and the use of GPUs with CUDA-C. The experiments show that OpenMP excels with smaller images, while GPUs perform better with larger images, achieving up to 35 times the acceleration of the sequential version. Additionally, a parallel centroid initialization

technique enhances the quality of the results and yields further speed gains.

These works highlight different strategies for accelerating and adapting the *k-means* algorithm. Each contribution addresses performance, scalability, or applicability challenges. A comparative summary of these studies is present in Table 1, which contrasts the use of parallelism, the tools employed, and the hardware platforms explored.

Table 1. Overview of related work

Article	Tool / Model	Device
Bhimani et al. (2015)	OpenMP and CUDA	GPU NVIDIA Tesla K20m GPU
Andrade Maciel et al. (2020)	VHDL (Reconfigurable Hardware)	FPGA Intel Cyclone IV-E (DE2-115)
Yin and Zhang (2017)	MapReduce (Cloud Computing)	Cloud Server Cluster
Pathak and Ananthanarayana (2012)	Multi-threading (C# .NET)	CPU Intel Core i5 (Dual-core $\times$ 2, 4 threads)
This work	OneAPI and OpenMP	Intel(R) Core(TM) i7 and NVIDIA GeForce RTX 3050

3. Methodology

This section describes materials and method used to design and test.

The following hardware configuration was used to perform the tests: 11th Gen Intel(R) Core(TM) i7-11800H processor @ 2.30GHz, 16GB of RAM, and NVIDIA GeForce RTX 3050 Ti Mobile graphics card. The tests were conducted using the following interfaces and compilers: oneAPI 2025.2.0, codeplay 2025.2.0 Codeplay Software Ltd. / Intel (2025), OpenMP, GCC 13.3.0, CUDA release 13.0, and Clang 18.1.8. All tests were run with the -O3 flag enabled for parallelism optimization.

To ensure greater reliability and reduced variance in test execution, three main measures were adopted. First, five repeated runs were performed for each metric collection. Second, controlled execution conditions were maintained through the use of standardized criteria for the number of iterations and centroid initialization. Finally, a fixed frequency policy was implemented, with Persistent mode enabled for the GPU and Performance mode for the CPU.

3.1. k-means

The *k-means* algorithm (MacQueen, 1967) is based on the definition of a central element, called *centroid*, for each of the k groups (*clusters*) previously specified. The variable k represents the number of groups formed. For each of n instance to be classified, the distance between this element and the existing centroids is calculated. Then, the object is assigned to the *cluster* whose centroid presents the smallest distance, as shown in Figure 1, that is, the most similar in terms of characteristics or attributes.

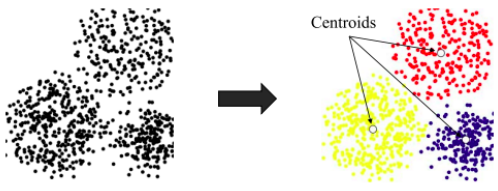

Figure 1. Example of clustering with k = 3. (Andrade Maciel et al., 2020).

3.2. Dataset

The CIC-IDS2017 (Sharafaldin et al., 2018) dataset was created by the Canadian Cyber Security Institute for research in Intrusion Detection Systems (IDS) and Intrusion Prevention Systems (IPS). The main objective was to overcome the shortcomings of existing datasets, which were considered outdated and unreliable.

The seven original CSV files, with different days and attack types, were merged into a single set with over 2 million records. Then, a thorough cleaning was performed, removing rows with zero values and records containing invalid values (NaN).

The initial balancing analysis revealed a significant imbalance between classes. To address this, six classes with low representation were excluded, resulting in nine main classes. These nine classes were subsequently mapped to four broader categories: Not an attack, DDoS, DoS, and Brute Force. To simplify the problem, the classification was reformulated as a binary problem, where "Not an attack" corresponded to Class 0 and "Any type of attack" corresponded to Class 1. Due to the still relatively small number of attacks, oversampling techniques were applied, resulting in a balanced base of 4 million instances (2 million attacks and 2 million non-attacks).

In the final step, the labels were separated, the data was normalized, and the dimensionality was reduced using Principal Component Analysis (PCA), resulting in 22 more relevant attributes. Finally, starting from the original 4 million instances, the dataset was progressively expanded to 32 million instances through eightfold replication, enabling large-scale testing and assessing performance gains. This pre-processing process ensured a clean and balanced dataset suitable for analysis of cyberattack detection.

4. Implementation of k-means

The execution flowchart of the *k*-means algorithm is shown in Figure 2. The parallel versions focus on the *Assignment* and *Update* steps.

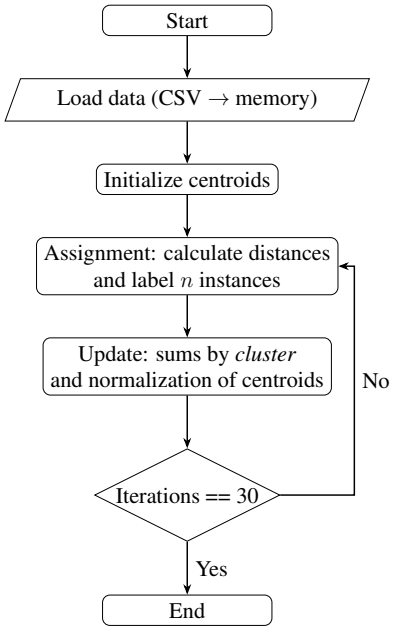

Figure 2. Flowchart of *k-means* execution.

To analyze the bottlenecks of the code that could be optimized through parallelization, the initial implementation was performed, grouping 4 million instances into two clusters and 32 million instances into 16 clusters. The application was then subjected to the Intel VTune Profiler. The results provided by the platform are shown in Table 2.

Table 2. VTune profile application comparison n= 32 million instances, k= 16 and n= 4 million instances, k= 2

Region	n = 32M, k = 16 [%]	n = 4M, k = 2 [%]
File reading	43.6	66.9
Attribution	43.2	15.1
Centroid update	6.9	13.4
Others	3.6	4.6

Thus, it was possible to ascertain that the fraction of the code responsible for performing computations was not significant for small data volumes and had a relevant portion of its execution in sequential and I/O operations. In this sense, only in large-scale scenarios would parallelism enable a significant improvement in terms of time.

Given the above, it was decided to focus the implementation of parallelism, partially, on the function responsible for the updating of centroids, whose complexity is $O(\text{number of columns in the dataset} \times \text{number of rows in the dataset})$, and on the function responsible for the grouping of instances into *clusters*, whose order of complexity is $O(\text{number of rows in the dataset} \times \text{number of centers} \times \text{number of columns in the dataset})$.

Parallelism in this implementation arises from distributing instances across threads with SYCL's `parallel_for`, so that each work-item independently assigns one instance to the nearest cluster. Data partitioning occurs naturally, while SYCL's buffer/accessor model handles synchronization and memory consistency.

5. Results

This section presents the results of the sequential and parallel versions of *k*-means. The average times in different scenarios and the respective gains in each were analyzed; scalability tests were also conducted, and model quality metrics were reported. Scalability tests were performed only on CPU, as it was not possible to configure the number of *cores*/SMs equivalently on the GPU.

5.1. Average time per scenario

According to Figure 3, the best performance scenario was $n = 32\,\text{million}$ and $k = 16$, which is why it was adopted in the scalability experiments. In addition, the quality metrics (accuracy with Hungarian, ARI, AMI, silhouette, Davies–Bouldin, and Calinski–Harabasz) were collected only for $n = 4\,\text{million}$ and $k = 2$, because the other increases in n and k were generated synthetically only for a performance study. The standard deviations were low in all executions (0.1%–3.5% of the mean), indicating consistent execution and reliable averages.

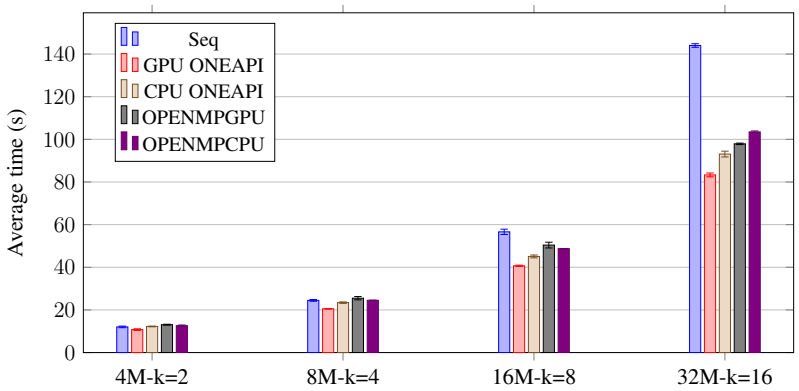

Figure 3. Average times and standard deviations by scenario.

5.1.1. Speedup of the best cases

Table 3 shows the relative speedup for the best cases in the scenario that uses 32 million instances and 16 clusters. The comparison is made pairwise for both heterogeneous code on CPU using oneAPI and OpenMP, as well as for sequential execution, and for heterogeneous code on GPU using oneAPI and OpenMP. Values > 1 indicate the row is faster.

Table 3. Speedup matrix for n = 32M, k=16

	Seq	GPU ONEAPI	CPU ONEAPI	GPU OPENMP	CPU OPENMP
Seq	1.000	0.579	0.646	0.680	0.719
GPU ONEAPI	1.729	1.000	1.117	1.175	1.242
CPU ONEAPI	1.547	0.895	1.000	1.052	1.112
OPENMP GPU	1.471	0.851	0.951	1.000	1.057
OPENMP CPU	1.392	0.805	0.899	0.946	1.000

5.1.2. Strong Scalability

Figure 4 illustrates strong scalability in CPU oneAPI (oneAPI, 2025) and OpenMP (OpenMP, 2025) for n=32M and k=16, comparing the average time (in seconds) with the number of threads (p). The graph shows that there is an improvement with increasing the number of threads, which stabilizes around eight threads. For CPU oneAPI, the standard deviations ranged from 0.1% to 2.6% of the mean, while for CPU OpenMP, they remained even lower, between 0.1% and 0.4%, confirming the stability of both implementations.

5.1.3. Weak scalability in oneAPI CPU for (n=32M).

Figure 5 illustrates the weak scalability of CPU oneAPI and OpenMP with the problem size scaled so that $N/P = 2$ (i.e., $N = 2P$). As p increases (e.g., from 2 to 16), N grows proportionally (from 4M to 32M instances). The plot reports average wall-clock time (in seconds) versus p, with the number of clusters fixed at $k = 16$. For CPU oneAPI, the standard deviations ranged from 0.8% to 5.8% of the mean, while for CPU OpenMP they varied between 0.5% and 2.2%, showing stable performance in both cases.

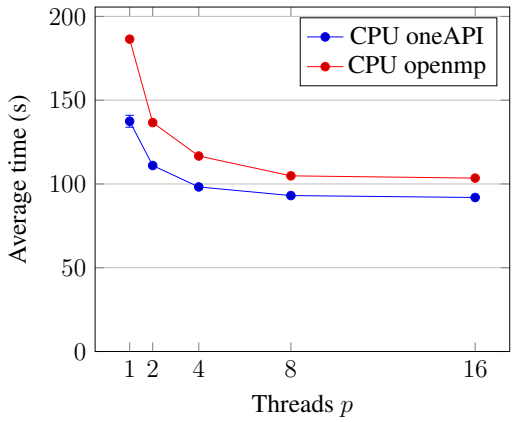

Figure 4. Strong CPU scalability using oneAPI and OpenMP for (n=32**M**, k=16)

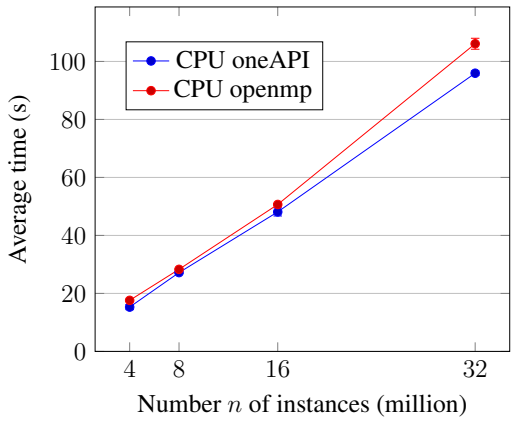

Figure 5. Weak scaling: time vs. threads with problem size scaled as $N = 2P$. Error bars: sample standard deviation over 5 runs.

5.2. Model correctness metrics

For a specific seed among those tested, after label alignment using the Hungarian algorithm, an accuracy of **88.96%** was obtained, with $\mathrm{ARI} = 0.5486$ and $\mathrm{AMI} = 0.4780$. In the internal metrics, the average silhouette was 0.1997. In short, this seed presents high external correspondence, but only moderate separation between clusters.

6. Conclusion

The results demonstrate that parallelized implementations on both CPU and GPU systems yielded significant speedups in execution time, especially in scenarios with large data volumes. Profiling analysis revealed that the fraction of code dedicated to computations becomes more significant in these contexts, allowing parallelism to deliver significant improvements in processing time. In the case of GPUs in particular, such gains require sufficiently large workloads to compensate for the device–host data transfer overhead. However, as the amount of data increases, the performance difference becomes significantly more evident. It is also observed that parallelization with *OneAPI* (oneAPI, 2025) presented slightly better performance compared to *OpenMP* (OpenMP, 2025).

On CPU, both oneAPI and OpenMP show strong scaling. The execution time decreases as the thread count increases, but the improvement stabilizes early, indicating

that memory bandwidth limitations are likely the cause. Under weak scaling, with the workload per thread held constant and the number of clusters fixed, execution times rise with p rather than remaining flat. This pattern reflects low weak-scaling efficiency and is consistent with the cost of the sequential fraction of the centroid-update and synchronization overhead. We report scalability only on CPUs, where the degree of parallelism P is directly controllable, enabling standard strong and weak scaling with reproducible results. GPUs lack a portable notion of P, and a single device does not allow proper strong scaling. Therefore, we restrict the analysis to the CPU.

As future work, we intend to evaluate GPU scalability and develop a version of the *k-means* algorithm in CUDA, in order to perform performance comparisons with the GPU implementation of OneAPI (oneAPI, 2025), using real datasets. Furthermore, we propose to investigate the algorithm's acceleration on heterogeneous architectures and evaluate its execution on reconfigurable devices, such as FPGAs (Andrade Maciel et al., 2020), or on hybrid systems that combine CPU and GPU. In this context, dynamic load balancing between different processing units may represent a promising strategy to increase the model's efficiency and scalability in real-time intrusion detection applications.

References

L. Andrade Maciel, M. Alcântara Souza, and H. Cota de Freitas. Reconfigurable fpga-based k-means/k-modes architecture for network intrusion detection. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 67(8):1459–1463, 2020. doi: 10.1109/TCSII.2019.2939826.

J. Bhimani, M. Leeser, and N. Mi. Accelerating k-means clustering with parallel implementations and gpu computing. In *2015 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6, 2015. doi: 10.1109/HPEC.2015.7322467.

Codeplay Software Ltd. / Intel. oneapi for nvidia® gpus. `https://developer.codeplay.com/products/oneapi/nvidia/home`, 2025. Access on August 21, 2025.

J. MacQueen. Multivariate observations. In *Proceedings ofthe 5th Berkeley Symposium on Mathematical Statisticsand Probability*, volume 1, pages 281–297, 1967.

oneAPI. oneapi programming guide. `https://www.intel.com/content/www/us/en/developer/tools/oneapi/overview.html`, 2025. Access on August 21, 2025.

OpenMP. The openmp api specification. `https://www.openmp.org/specifications/`, 2025. Access on August 21, 2025.

V. Pathak and V. S. Ananthanarayana. A novel multi-threaded k-means clustering approach for intrusion detection. In *2012 IEEE International Conference on Computer Science and Automation Engineering*, pages 757–760, 2012. doi: 10.1109/ICSESS.2012.6269577.

I. Sharafaldin, A. H. Lashkari, A. A. Ghorbani, et al. Toward generating a new intrusion detection dataset and intrusion traffic characterization. *ICISSp*, 1(2018):108–116, 2018.

C. Yin and S. Zhang. Parallel implementing improved k-means applied for image retrieval and anomaly detection. *Multimedia Tools Appl.*, 76(16):16911–16927, Aug. 2017. ISSN 1380-7501.