© 2024 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

CUDA Calculation of Shannon Entropy for a Sliding Window System

Gordon Velichkovski, Marjan Gusev, Dimitar Mileski
Ss. Cyril and Methodius University
Faculty of Computer Science and Engineering
1000, Skopje, North Macedonia
gordon.velichkovski@proton.me, marjan.gushev@finki.ukim.mk, dimitar.mileski@finki.ukim.mk

***Abstract*—Entropy algorithms are crucial in fields where assessing randomness, uncertainty, or complexity is vital. As datasets grow, efficient entropy calculations become important. This work explores the parallelization of Shannon entropy calculations, using GPU acceleration through CUDA for sliding window systems. By leveraging GPUs' parallel architecture, the approach achieves up to 15x speedup for large datasets. However, smaller datasets show limited improvements due to overhead, underscoring the need for optimization to harness GPU acceleration's potential.**

***Index Terms*—Shannon Entropy, CUDA, GPGPU, Parallel Processing**

I. Introduction

Entropy algorithms offer valuable insights into the structure and predictability of data [1]. There is an increasing demand for extensive processing of large-scale datasets for entropy algorithms, including sample, differential, and Shannon entropy calculation.

Since the algorithms proposed in the literature require execution time with complexity of $O(n^2)$ [2], a promising avenue for achieving this improvement is through parallelization techniques, mainly using Compute Unified Device Architecture (CUDA), a parallel computing platform and application programming interface model created by NVIDIA for Graphics Processing Units (GPUs). CUDA enables programmers to write code that can be executed on NVIDIA GPUs, leveraging their massively parallel architecture to accelerate a wide range of computational workloads [3].

This research aims to leverage GPU-based parallel computing to improve the efficiency of calculating Shannon entropy in a sliding window system. While several works have explored GPU-accelerated entropy calculations, our research on the state-of-the-art does not show extensive evidence for using CUDA for the real-time calculation of Shannon entropy over a sliding window system. Previous work has explored methods like matrix-based entropy calculations [4] and bucket-assisted algorithms [5], achieving significant speedups. However, these methods generally target different entropy algorithms or are designed for static datasets, not for sliding window systems. This work differs by implementing Shannon entropy calculation for continuous data streams, using CUDA to handle large datasets more efficiently.

Related work to calculate Shannon entropy with CUDA is in Section II, and the background of Shannon entropy is in Section III. The methodology used to calculate Shannon entropy in CUDA (Section IV) presents the solution architecture, experiments, and evaluation Methodology. Section V presents the results, and Section VI discusses their performance, comparison with other approaches, and analysis of the implications in real-world CUDA applications, along with the potential performance gains. Finally, Section VII concludes with remarks on the efficiency and recommendations for further optimizations in CUDA-based systems.

II. Related Work

Biological and medical research data is often complex. It's usually non-linear and exhibits complex behavior. It also includes some unavoidable noise [6]. Therefore, entropy estimates frequently quantify fluctuations' regularity and unpredictability over time-series data. Mullayanov et al. [4] increase the productivity of approximate entropy calculation by evaluating the algorithm's complexity and proposing an approach for approximate entropy calculation. Their approach uses matrix calculations implemented on a GPU with parallel computing, achieving a significant reduction in the calculation time by up to one order of magnitude. This solution can operate with a larger volume of information.

Sample entropy is another widely used method with high computational complexity. To improve the computational performance, [7] proposed an OpenCL solution to run a fast parallel algorithm on GPU. They used 24-hour heartbeat data and showed that the improved algorithm could reduce the execution time to $1/75^{th}$ of the base algorithm – on long signal lengths (larger than 60,000). A bucket-assisted parallelized algorithm [5] reveals experimental results for a 24-hour Holter recording achieve a speedup of around 8, using a 12-thread CPU instead of GPU.

Rough entropy is another essential tool to measure the uncertainty of information. An efficient algorithm [8] to accelerate the calculation of rough entropy with CUDA on a Tesla K40M with 2880 cores and 12GB global memory. They claim a 14.1 to 17.9 speedup ratio compared to the serial algorithm.

Two order magnitude improvements were achieved with algorithmic optimizations on the sample entropy algorithm [9]. Although this is not a parallelized optimization, it would be interesting to see the Speedup of the algorithm implemented

This is accepted version. Published version DOI: **10.1109/TELFOR63250.2024.10819103**

in CUDA. Similarly, a bucket-assisted speedup of the approximate entropy algorithm [10] is suggested as the next step to investigate parallelized approaches.

III. Background

First, we present the original method approach to understand a proposed approach of calculating Shannon entropy in parallel. Shannon entropy calculates the probability distribution of symbols in a data stream of size N by the following algorithm steps:

1) *Frequency Calculation Step* counts the occurrences n_i of each unique symbol i in the data stream.
2) *Probability Calculation Step* calculates the probability p_i of each symbol i by dividing its frequency n_i by the total number of symbols N, presented by (1).

$$p_i = \frac{n_i}{N} \tag{1}$$

3) *Entropy Calculation* uses (2) to calculate Shannon entropy $H(X)$, where k is the number of unique symbols in the data stream.

$$H(X) = -\sum_{i=1}^{k} p_i log_2(p_i) \tag{2}$$

Algorithm 1 presents the pseudo-code to calculate Shannon Entropy on the data stream $data$.

Algorithm 1 Shannon Entropy Pseudo Code

Input: $Data$

Output: $Entropy$

1: $Counts \leftarrow 0$
2: **while** $Symbol \notin Data$ **do**
3: **if** $Symbol \notin Counts$ **then**
4: $Counts[Symbol] \leftarrow 1$
5: **else if** $symbol \in Counts$ **then**
6: $Counts[Symbol] \leftarrow Counts[Symbol] + 1$
7: $P \leftarrow 0$
8: **while** $Symbol \in Counts$ **do**
9: $P[Symbol] \leftarrow Counts[Symbol]/N$
10: $Entropy \leftarrow 0$
11: **while** $Symbol \in Probabilities$ **do**
12: $Entropy \leftarrow Entropy - P[Symbol] * \log_2 P[Symbol]$

The code consists of three parts (while loops). The first one iterates the entire input data stream once and maintains a list of frequency counts for each unique symbol with a complexity of $O(n)$ where n is the size of the input data stream. The second loop calculates the probability of occurrence for each unique symbol. Since the iteration is over k unique symbols, the complexity is $O(k)$. The third loop iterates over the probabilities with k unique symbols to calculate the Shannon entropy with a complexity of $O(k)$ The algorithm has an overall time complexity of $O(n+k)$ or just $O(n)$ since $n >> k$ is the dominant factor in executing the algorithm with traversal over the n data items, and k is a predefined small constant.

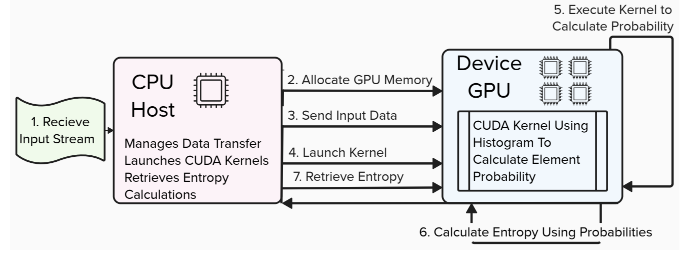

Fig. 1. GPU Solution Architecture

In the real-time scenario, the calculation repeats in the sliding window approach for the last n data elements from the continuously incoming data stream with the size of N. The sliding window algorithm executes the Shannon entropy algorithm with a $O(n+k)$ complexity for each incoming data stream, and the overall total complexity is $O(N * (n + k))$. The complexity of simplifies to $O(N * n)$, as $N >> k$. For small window sizes, $N >> n$, and the algorithm complexity is $O(N)$, while it is $O(N * n)$ for large window sizes.

IV. Methods

A. Sliding Window Real-Time Architecture

The following variables define and parameterize the calculating entropy for a real-time sliding window system.

1) *Input Stream Velocity (v)* calculates the amount of the input data items per second. In this solution, the input data stream is a large static array traversed by a sliding window.
2) *Window Size (n)* is the sliding window size representing the data array $Data$ to calculate the Shannon Entropy SE.
3) *Threads per Block* (TpB) refers to the number of threads per block launched and determines how the GPU will schedule and execute the kernel.
4) *Number of Bins* (B) is a predefined number the strategy specifies to allocate each data stream item into a specific bin.

B. Parallel Solution

Fig. 1 presents the parallel solution. The algorithm transfers input data from the host to the device and, after processing, transfers calculated results to the host. Reusing data from one kernel to the next saves an entire data transfer round trip or half a round trip if we were to compute the rest of the result on the host.

One kernel calculates the probabilities of elements from the frequency map by calculating a histogram. Each thread in the GPU is responsible for processing a portion of the input data and updating the histogram accordingly.

After parallel computing the histogram, we perform a reduction operation to combine the histograms from different threads into a single global histogram probability result. Instead of the number of occurrences, each bin stores the computed probability for that unique element index. Another

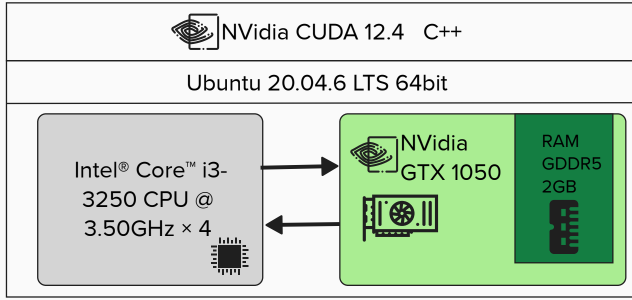

Fig. 2. System Architecture

kernel calculates the entropy from the results obtained from the previous kernel. One thread calculates the entropy value for each unique data item, and the reduction method calculates the sum.

C. Parallel Algorithm Complexity

Each thread within a block processes a portion of the input stream to calculate the algorithm's first part (local histograms). The overall complexity of $O(N * TpB)$ largely depends on the size of the incoming data stream N and TpB.

The reduction operation to combine local histograms into a global histogram takes O($\log B$) time steps. The local histogram construction complexity dominates if the input size is significantly larger than the total number of threads. The global histogram reduction complexity might become relevant if the number of bins is enormous compared to the number of threads per block. However, in practice, the number of bins is predefined to be a relatively small value compared to the input size.

The third kernel that calculates the Shannon entropy for each bin activates thread for a single bin with a per-bin complexity of $O(1)$. The overall complexity of this step is $O(B/TpB)$. Summing up the complexities in the pipelined parallel execution reveals the final complexity of $O(N*TpB)$, which can be simplified to $O(N)$ since TpB can be safely omitted from the asymptotic complexity since it is typically a constant value determined by the hardware and optimization parameters.

The other parameters will take values within a predefined range in test cases to experiment and find the best Speedup and the optimal parameters.

D. Experiments

We executed the serial implementation on a computer with an Intel i3-3250 CPU, running Ubuntu 20.04 64-bit OS. The parallel solution utilizes an NVIDIA GTX 1050 GPU, Intel i3-3250 CPU, running Ubuntu 20.04 64-bit OS (Fig. 2), and CUDA C++ to develop parallel code.

Two experiments include serial and parallel CUDA implementations. The test cases involve calculating Shannon entropy for

- window sizes n ranging from 100 to 1 million data items,
- thread-per-block configurations $TpB \in \{1, 10, 32, 256\}$,
- each test case runs ten times to ensure result consistency and report an average execution time.

TABLE I
EXECUTION TIMES FOR VARIABLE WINDOW SIZE N.

	Serial	Parallel (CUDA)			
N		$TpB = 1$	$TpB = 10$	$TpB = 32$	$TpB = 256$
100	0.00006s	0.00137s	0.00172s	0.00161s	0.00158s
500	0.00022s	0.00192s	0.00216s	0.00200s	0.00234s
1000	0.00030s	0.00229s	0.00244s	0.00235s	0.00213s
5000	0.00090s	0.00304s	0.00257s	0.00265s	0.00220s
10K	0.00130s	0.00378s	0.00281s	0.00270s	0.00225s
50K	0.00630s	0.00922s	0.00302s	0.00277s	0.00225s
100K	0.01040s	0.01661s	0.00381s	0.00294s	0.00223s
500K	0.05350s	0.07245s	0.00570s	0.00370s	0.00367s
1M	0.10120s	0.13617s	0.01416s	0.00674s	0.00660s

E. Evaluation Methodology

The primary evaluation metric is execution time (measured in seconds). We calculate Speedup by (3), where T_s is the average execution for executing the serial implementation for a particular test case, and T_p refers to the parallel CUDA implementation.

$$Speedup = \frac{T_{\mathrm{s}}}{T_{\mathrm{p}}} \quad (3)$$

We will also consider the overhead introduced by parallelization, particularly for smaller datasets, where the time spent managing GPU threads and transferring data may outweigh the benefits of parallel computation.

V. RESULTS

The C++ programming language implements both algorithms. The NVCC compiler compiles the CUDA C++ code, and the g++ compiler compiles the serial C++ algorithm. The NVIDIA 1050 GPU has Pascal architecture, 640 CUDA cores, and 2 GB of GDDR5 memory [11]. Table I presents the achieved results for various window sizes.

For larger datasets, the CUDA implementation outperforms the serial implementation, with processing times remaining the same or increasing by a small factor as the number of processed data items increases. However, for smaller datasets, the overhead associated with data transfer outweighs the benefits of the parallel implementation, resulting in lower performance than the serial implementation.

Fig. 3 presents the achieved speedup reaching up to 15x for large dataset windows, particularly for more than 50K data items. In other cases, the speedup is lower than 1.

VI. DISCUSSION

While the performance improvement over the serial implementation is noticeable for large numbers of elements (e.g., window size larger or equal to 50K), the gains are the opposite for smaller datasets. The parallelized CUDA implementation exhibits faster processing times with larger window sizes, with the Speedup at around 14 to 15 compared to the serial algorithm.

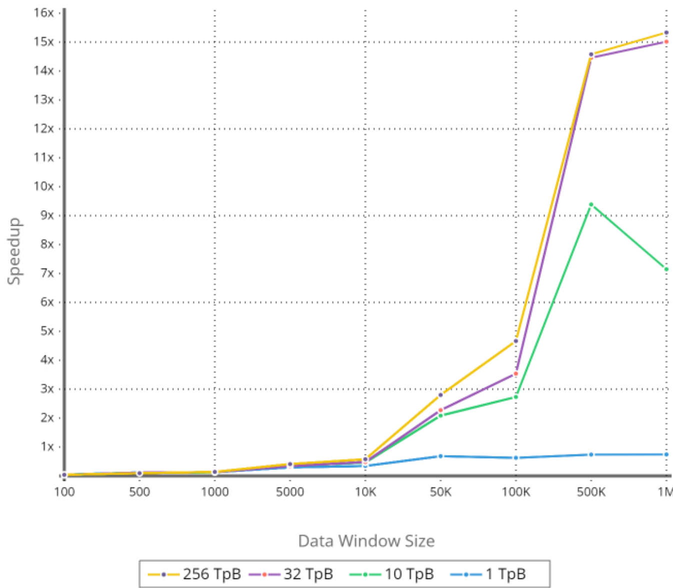

Fig. 3. Speedup Chart

Throughput scales well with increasing input size, validating the efficiency of parallelism. The Speedup saturates around 15x due to a combination of non-parallelizable operations (Amdahl's Law [12]), data transfer requirements, memory bandwidth limitations, and kernel launch overhead, which prevent further scaling despite the availability of more computational resources on the GPU.

The data transfer overhead between the CPU and GPU can significantly impact the performance of GPU-accelerated algorithms, particularly when working with smaller datasets; however, as dataset sizes increase, the advantages of the GPU's parallel processing capabilities generally compensate for the transfer time, leading to overall improved performance.

Compared to previous works, such as the approximate entropy algorithm achieving speedups between 8-10x on multi-threaded CPUs [5] or rough entropy computation on GPUs with 14.1-17.9x speedup [8], our implementation shows similar results. The 15x Speedup achieved aligns with these findings, especially given the similar dataset sizes and GPU architectures. While the data transfer bottleneck is present in our implementation, it is also noted in prior works, emphasizing that further optimizations in memory management or hybrid CPU-GPU strategies may yield additional gains.

Comparing the complexity of the serial algorithm $O(N * (n + k)$ to the $O(N * TpB)$ complexity of the parallel algorithm will compare TpB and k in addition to the window size n. Considering that k and TpB are small constants, it follows that $N >> k$ and $N >> TpB$ for the large-sized incoming data stream. The parallel algorithm's Shannon Entropy calculation per window is $O(1)$, and $O(n)$ for serial algorithm implementation.

Both algorithms are comparable for small arrays $N >> n$. Since the execution time depends on the data transfer and kernel activation times, the serial algorithm implementation is advantageous. However, the parallel algorithm implementation is advantageous for large arrays where $n >> k$. Although both algorithms scale linearly with increasing input size, the parallel implementation is optimized to leverage the GPU's architecture and can handle larger datasets more efficiently. The overhead associated with data transfer and kernel execution impacts performance for smaller data stream sizes.

While parallelization through CUDA demonstrates the potential for optimizing performance with significant data volumes, the minimal gains suggest that further optimizations or alternative parallelization strategies may be necessary to fully leverage the computational power of GPUs across a broader range of problem sizes.

VII. Conclusion

This research explores the GPU-accelerated sliding window-based streaming approach for entropy calculations, with a primary emphasis on Shannon entropy. The results show that while the parallelized algorithm offers performance improvements for large datasets, the algorithm execution for smaller datasets limits the gains due to the overhead associated with parallelization. The parallelized CUDA implementation exhibits faster processing times, especially with larger window sizes. Yet, the improvement diminishes as the complexity of the original algorithm is already decently efficient.

References

[1] A. Delgado-Bonal and A. Marshak, "Approximate entropy and sample entropy: A comprehensive tutorial," *Entropy*, vol. 21, no. 6, 2019.

[2] Y.-H. Pan, Y.-H. Wang, S.-F. Liang, and K.-T. Lee, "Fast computation of sample entropy and approximate entropy in biomedicine," *Computer Methods and Programs in Biomedicine*, vol. 104, no. 3, pp. 382–396, 2011.

[3] J. Sanders and E. Kandrot, *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010.

[4] R. Mullayanov, A. Krushkov, and R. Nasyrov, "Approximate entropy technique of calculation based on parallel computation with usage of gpu," in *Proceedings of the 8th Scientific Conference on Information Technologies for Intelligent Decision Making Support (ITIDS 2020)*, pp. 103–108, Atlantis Press, 2020.

[5] G. Manis, D. Bakalis, and R. Sassi, "A multithreaded algorithm for the computation of sample entropy," *Algorithms*, vol. 16, no. 6, 2023.

[6] M. Borowska, "Entropy-based algorithms in the analysis of biomedical signals," *Studies in Logic, Grammar and Rhetoric*, vol. 43, no. 1, pp. 21–32, 2015.

[7] X. Dong, C. Chen, Q. Geng, W. Zhang, and X. D. Zhang, "Fast algorithm based on parallel computing for sample entropy calculation," *IEEE Access*, vol. 9, pp. 20223–20234, 2021.

[8] S. Jing, C. Liu, G. Li, G. Yan, and Y. Zhang, "An efficient algorithm for parallel computation of rough entropy using cuda," in *2017 13th International Conference on Computational Intelligence and Security (CIS)*, pp. 1–5, 2017.

[9] C. Chen, C. Liu, J. Li, and B. da Silva, "Acceleration of bucket-assisted fast sample entropy for biomedical signal analysis," *IEEE Transactions on Instrumentation and Measurement*, vol. 72, pp. 1–11, 2023.

[10] G. Manis, "Fast computation of approximate entropy," *Computer Methods and Programs in Biomedicine*, vol. 91, no. 1, pp. 48–54, 2008.

[11] https://www.nvidia.com/en-gb/geforce/graphics-cards/geforce-gtx-1050/specifications/ [Accessed: 10/06/2024].

[12] M. D. Hill and M. R. Marty, "Amdahl's law in the multicore era," *Computer*, vol. 41, no. 7, pp. 33–38, 2008.