International Journal of Applied Science; Vol. 8, No. 3; 2025
ISSN 2576-7240 E-ISSN 2576-7259
https://doi.org/10.30560/ijas.v8n3p1

Optimizing Logcumsumexp on Cambricon MLU: Architecture-Aware Scheduling and Memory Management

Xiaohu Xu[1] & Yusen Zhu[2]

[1] Guizhou University of Finance and Economics, China

[2] The Woodstock Academy, US

Correspondence: Xiaohu Xu, Guizhou University of Finance and Economics, China.

Received: June 15, 2025; Accepted: July 1, 2025; Published: July 3, 2025

Abstract

The Logcumsumexp algorithm is a core method for numerically stable cumulative summation in logarithmic space, especially suitable for scenarios involving extremely small or large numerical computations. By applying logarithmic transformation, this algorithm effectively addresses the common issues of underflow and overflow in probability calculations, deep learning, and statistical modeling, making it an important high-performance computing algorithm. In recent years, China's chip industry has been continuously rising, and the domestic MLU computing platform from Cambricon Technologies has provided new options for global users. Based on the Cambricon MLU computing platform and in combination with its hardware structure, this paper constructs a set of Logcumsumexp algorithm named MLULCSE, which can perform Logcumsumexp operations on tensors of any dimension along the specified dimension and has been optimized for different types of Logcumsumexp tasks. By categorizing tasks into four types and implementing different strategies tailored to the hardware architecture, we achieved efficient logcumsumexp computation. This work enables efficient probabilistic computing on domestic AI accelerators, experimental results show that MLULCSE running on MLU 370-X4 has a hardware time that is controlled within 7 times compared to Pytorch Logcumsumexp running on Tesla V100, and in some cases, it even reaches 0.42 times.

Keywords: MLU, High-Performance Computing, logcumsumexp

1. Introduction

1.1 Logcumsumexp

The logcumsumexp algorithm is a practical numerical function primarily used for computing cumulative exponentiation in a numerically stable manner within the logarithmic space, which has been widely applied in the field of artificial intelligence.

Logcumsumexp is a stable version of the cumsum operation in log-space and can also be seen as an extension of logsumexp. Given a vector $x = [x_0, x_1, \dots, x_n]$, then $logcumsumexp(x) = [y_0, y_1, \dots, y_n]$, where:

$$y_k = \ln \sum_{i=0}^{k} e^{x_i}, k = 0,1, \dots, n \quad (1)$$

For example, given the vector [1, 2, 3, 4], we first apply the exponential function (base e) to each element, resulting in [2.718, 7.389, 20.085, 54.598]. Then, we perform a cumulative sum to obtain [2.718, 10.107, 30.192, 84.79]. Finally, we take the natural logarithm (log base e) of each element, resulting in [1.000, 2.313, 3.407, 4.441].

In machine learning and probabilistic computations, cumulative summation over very small (or logarithmic) probabilities is common. Direct computation can lead to numerical underflow or overflow—for instance, summing very small values like e^{-1000} may result in zero (underflow), while exponentiating large logarithmic values like 1000 may produce infinity (overflow). The logcumsumexp algorithm avoids these problems by operating directly in the log-space[1-5].

1.2 Cambricon MLU AI Accelerator Cards

The Cambricon MLU (Machine Learning Unit) is a dedicated processor designed for AI applications[6-8]. It features an optimized instruction set, pipelining, arithmetic units, and memory access mechanisms tailored to AI computation and memory access characteristics. Compared to general-purpose processors, MLU offers higher performance, flexibility, and energy efficiency in AI workloads.

The fundamental building block of Cambricon hardware is the MLU Core. Each MLU Core is a processor core equipped with complete computing, I/O, and control capabilities. It can execute computational tasks independently or collaborate with other MLU Cores to perform a task. Four MLU Cores form a Cluster, which also includes an additional Memory Core and a shared SRAM (Shared RAM) unit accessible by both the Memory Core and the four MLU Cores.

In addition, each MLU Core is equipped with its own private on-chip memory, namely NRAM and WRAM. NRAM is primarily used to store the input and output data of vector and tensor operations, and can also be used to hold temporary scalar data generated during computations. WRAM, on the other hand, is mainly used to store convolution kernel data for convolution operations. To enable efficient convolution execution, data in WRAM follows a specialized layout, and various matrix operation functions can only operate on matrices stored in WRAM. WRAM does not support standard scalar read/write operations and can only be accessed explicitly through data transfer instructions.

1.3 Main Content of This Paper

The primary focus of this paper is the implementation of the logcumsumexp algorithm on the Cambricon MLU platform, referred to as the MLULCSE algorithm. This algorithm is capable of processing tensors along arbitrary dimensions and performing the logcumsumexp operation along a specified dimension of the tensor. By employing task-specific allocation strategies, the algorithm achieves high efficiency when applied to different target dimensions across tensors of varying shapes and sizes.

2. Related Work

The logsumexp function, as the predecessor of the logcumsumexp function, has been extensively studied, particularly for enhancing numerical stability in softmax functions and logarithmic-domain operations. Blanchard et al. systematically analyzed the rounding errors of different logsumexp implementations and recommended subtracting the maximum input value prior to computation to improve accuracy. This technique is also applicable to the efficient and stable implementation of logcumsumexp, especially given the differences in CPU and GPU implementations, which has been widely discussed in communities such as PyTorch.

In recent years, researchers have introduced the log-sum-exp structure into convex neural networks. Calafiore et al. proposed using log-sum-exp units as smooth approximations of the maximum operation to construct convex posynomial models, demonstrating promising applications in control and optimization tasks[9]. Miyagawa and Ebihara further applied this concept to sequence density ratio matrix estimation, achieving a trade-off between speed and accuracy through log-sum-exp[10]. Chen and Gao studied a non-convex optimization problem involving quartic polynomials and log-sum-exp terms, and derived global optimal solutions using canonical duality theory, providing a theoretical foundation for the application of logcumsumexp-like structures in optimization-driven models[11].

Compared to logsumexp, logcumsumexp has been less extensively studied; however, it holds significant practical value in models requiring cumulative softmax or segmented normalization. Early frameworks such as PyTorch did not provide native support for logcumsumexp, forcing users to implement it by combining functions like cummax and exp, which poses challenges in terms of efficiency and stability, especially for long sequences or mixed-precision training.

In summary, although logsumexp has a relatively mature theoretical and engineering foundation, logcumsumexp, as its recursive extension, remains a promising direction for further exploration in sequence modeling, structured prediction, and logarithmic probability computations. Currently, frameworks like PyTorch have introduced logcumsumexp interfaces based on NVIDIA GPUs; platforms like TensorFlow and OneDNN do not even provide an explicit logcumsumexp interface; instead, similar functionality can only be achieved by combining multiple operations. Therefore, the development of logcumsumexp algorithms for domestic computing cards such as the MLU platform is still an open problem.

3. Task Allocation Strategy

3.1 Task Analysis

Based on the distinction between the target tensor's dimensionality and the target dimension for the operation, all logcumsumexp tasks are classified into four types:

(1) tasks with a one-dimensional tensor as the target;

(2) tasks where the target dimension is the highest dimension of the target tensor;

(3) tasks where the target dimension is the lowest dimension of the target tensor;

(4) tasks where the target dimension is an intermediate dimension.

These four task types are illustrated in Figure 1, where the gray area represents the portion requiring a complete logcumsumexp operation, and the arrows indicate the direction of the logcumsumexp computation.

Different task types require different handling methods. Considering the characteristics of the MLU platform, the algorithm needs to be designed with targeted optimizations. Specifically, the design must take into account: (1) whether the algorithm is SIMD-friendly; (2) whether synchronization between cores within a cluster is required; (3) whether synchronization between clusters is necessary. Addressing these factors allows the algorithm to maximize hardware computational power while minimizing the overhead caused by synchronization.

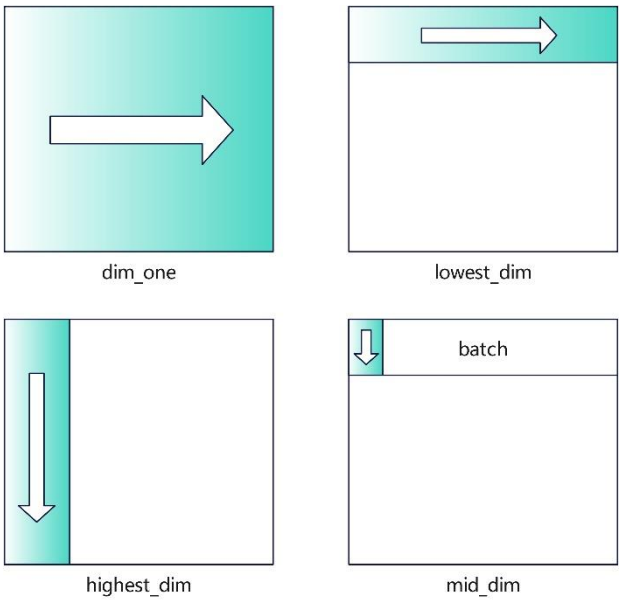

Figure 1. Four Types of Task

3.2 Tasks Targeting One-Dimensional Tensors

When the target tensor is one-dimensional (i.e., a vector), we directly partition the data into blocks and load them into different cores within different clusters. After computing the exponential of each element, we perform cumulative summation. Specifically, the target data is first transposed, then vector addition is performed row by row. The compensation value for each column is computed using the last row, and each row obtains its compensation value by vector addition with the compensation vector. Finally, the data is transposed back to its original layout. Following this, compensation is applied across cores and clusters according to the data distribution, and the final logarithm calculation yields the result.

Due to the inherent data dependency between every two elements in this task type, synchronization overhead across clusters is unavoidable when multiple clusters are engaged in processing.

3.3 Tasks with the Target Dimension as the Lowest Dimension of the Target Tensor

When the target dimension is the lowest dimension of the tensor, we treat the input data as a matrix with the target dimension as the width and the product of the higher dimensions as the height.

Intuitively, we can load several rows into one NRAM and perform row-wise cumulative addition after transposition. The advantage of this approach is that each core's task is independent, which avoids the overhead caused by inter-core synchronization. The drawback is that if the size of the target dimension exceeds the capacity of the NRAM, this strategy becomes infeasible. Even if the target dimension is somewhat smaller, if the number of rows that the NRAM can hold is limited, the row-wise cumulative addition will not fully utilize the device's parallelism, leading to wasted computational power. Therefore, this strategy is only applied to tasks with relatively small target dimensions.

For larger target dimensions, each row can be treated as a batch, and each batch independently employs the strategy for the first task type (i.e., when the target tensor is one-dimensional).

3.4 Tasks with the Target Dimension as the Highest Dimension of the Target Tensor

When the target dimension is the highest dimension of the tensor, the input data is partitioned into several column blocks. Each block is copied via 2D transfers onto an NRAM, where cumulative addition is performed row by row. If the NRAM capacity is insufficient to process the entire block at once, the block is loaded and processed in multiple passes. After computation, the results are copied back to the GDRAM through 2D transfers.

The advantage of this strategy is that each NRAM task is independent and does not require inter-core synchronization. Additionally, it is not limited by task size and can handle tensors and target dimensions of arbitrary scale. An important factor limiting the efficiency of this strategy is the frequent 2D memory transfers, which reduce the utilization rate of the memory device and increase memory access overhead. However, this limitation is dictated by the data storage structure and cannot be avoided through algorithmic design.

3.5 Tasks with the Target Dimension as an Intermediate Dimension

The target tensor is divided into multiple batches, where each batch size is the product of the target dimension size and all lower dimension sizes. Tasks are allocated with the batch as the minimum unit.Based on previous experience, two scenarios naturally arise:

(1) When the batch size is small, multiple batches can be loaded into the NRAM at once, and cumulative row-wise vector addition is performed independently for each batch. In this strategy, each core's task is independent, and both computational and memory resources are efficiently utilized.

(2) When the batch size is large, the third task type strategy (where the target dimension is the highest dimension of the target tensor) is directly applied to process each batch.

4. Accumulation Methods

4.1 Blelloch Algorithm

The Blelloch algorithm, proposed by Guy E. Blelloch in 1990, is an efficient parallel prefix sum computation method widely used in parallel computing, GPU programming, and large-scale data processing[12-14].

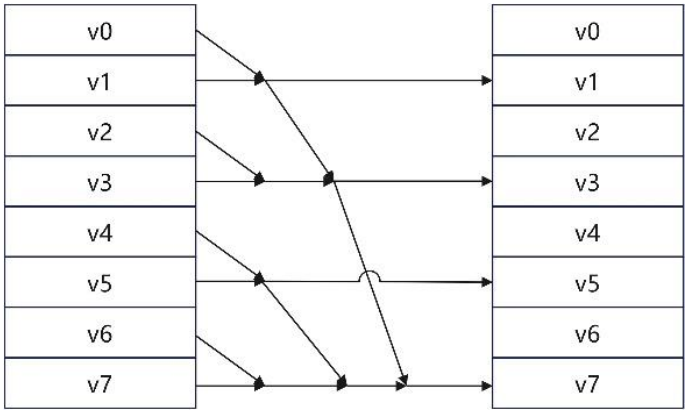

Figure 2. Blelloch up-sweep

The prefix sum operation transforms an input sequence into the cumulative sums of all its prefixes. Formally, given an input array $x = [x_0, x_1, \dots, x_n]$, it produces an output array $y = [x_0, x_0 + x_1, \dots, x_0 + x_1 + \cdots + x_{n-1}]$.The Blelloch algorithm is particularly suited for execution on parallel hardware such as SIMD units and GPUs. While maintaining a linear work complexity of $O(n)$, it achieves a logarithmic parallel time complexity of $O(\log n)$, making it a typical example of a work-efficient parallel algorithm.

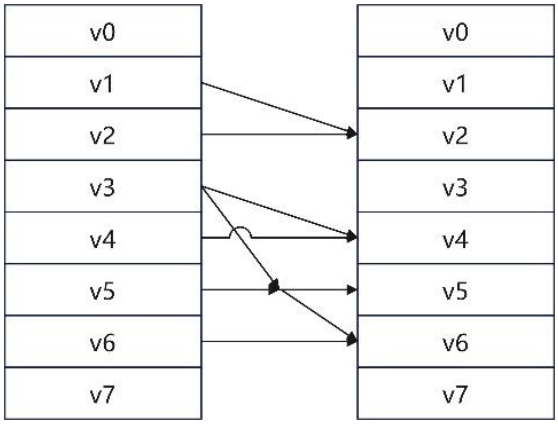

Figure 3. Blelloch down-sweep

The Blelloch algorithm consists of two phases: the up-sweep (or reduce) phase and the down-sweep phase. Figures 2 and 3 illustrate how the Blelloch algorithm processes vectors in NRAM. During the up-sweep phase, the algorithm constructs a binary tree structure and performs pairwise merging from the bottom up to compute the total sum reduction of the entire input sequence. In the subsequent down-sweep phase, prefix sums are distributed from the root node downwards, recursively updating each node's prefix value to produce the final scan result. This design ensures that multiple elements can be processed in parallel at each level, significantly improving processing efficiency.

Compared to the earlier Hillis-Steele scan algorithm, the Blelloch algorithm offers advantages in work complexity by avoiding redundant data movement and repeated computations. Consequently, it has been widely integrated into various parallel computing frameworks, such as CUDA's Thrust library, OpenCL toolchains, and modern parallel compilers, becoming one of the mainstream methods for implementing parallel prefix sums.

4.2 Analysis and Experimental Comparison between Blelloch and Direct Accumulation

The logcumsumexp algorithm designed in this paper involves a large number of row-wise vector accumulation operations. On the MLU platform, the advantage of the Blelloch algorithm lies in the fact that the two consecutive vector addition operations operate on vectors without data dependencies. Compared to direct row-wise accumulation, Blelloch allows the compiler to build instruction-level pipelines, thereby improving hardware utilization.

However, for an accumulation task with NNN rows, the total work (i.e., the number of vector addition operations) for direct accumulation is $N-1$, whereas Blelloch requires $2N-2$ operations. Therefore, whether Blelloch improves efficiency must be validated through experiments.

Experimental results show that using the Blelloch algorithm does not improve the efficiency of the logcumsumexp algorithm; instead, it reduces performance. This further indicates that on the MLU platform, the efficiency gains from instruction-level pipelining are insufficient to compensate for the doubled total workload.

5. Experimental Results and Analysis

This paper selects the PyTorch logcumsumexp running on the Nvidia Tesla V100 series as the benchmark for MLULCSE.

Table 1. Experimental Environment

Tesla V100-SXM2	**MLU 370-X4**
CUDA Version：12.1	Mluop Version：1.1.1
Pytorch Version：2.2.1	Driver Version：5.10.10

The experiments were conducted on 10 datasets with varying data sizes, tensor dimensions, and target dimensions. These datasets were provided by Cambricon and reflect the algorithm's performance across various common task

scales. All data types used were 32-bit floating-point, and the performance metric was hardware execution time, measured in microseconds (μs). Each experiment is conducted 100 times, and the average value is taken.

The results show that the hardware execution time of MLULCSE is generally kept within 7 times that of PyTorch. Considering the difference in floating-point computational power between the two platforms, this is a relatively successful outcome. In the second test case ([21, 41, 44], dim=0) and the sixth test case ([15200, 15], dim=1), MLULCSE's efficiency even surpassed PyTorch, with hardware times only 41.9% and 43.9% of PyTorch's respectively. This demonstrates the algorithm's ability to fully leverage the high parallelism of the MLU platform.

Table 2. Experimental Results

Input	V100	MLU
[2, 135, 45, 256]dim=2	52.22	275
[21, 41, 44]dim=0	21.5	9
[10, 60, 8, 43]dim=1	43.01	168
[648, 50]dim=1	19.46	117
[1160, 28]dim=1	18.43	112
[15200, 15]dim=1	368.64	162
[16, 166]dim=1	19.46	82
[9388608]dim=0	229.38	523
[4194304]dim=0	119.81	272
[1048576]dim=0	48.13	93

In the test cases where the target tensor is one-dimensional (the 8th, 9th, and 10th test cases), the execution time of MLULCSE reaches up to 2.3 times that of PyTorch at the largest scale. As the scale continues to grow, the machine parallelism on both sides is fully utilized, and the overhead of pipeline start-up and wind-down becomes proportionally smaller, causing the ratio of hardware times between the two to stabilize.

In the remaining test cases, MLULCSE performs slightly worse, with hardware times ranging from 3.9 to 6.1 times that of PyTorch. In these cases, the primary factor limiting MLULCSE's efficiency is the decrease in memory access efficiency caused by extensive use of 2D memory transfers.

6. Conclusion

To address the lack of a mature implementation of the logcumsumexp algorithm on the Cambricon MLU platform, this paper proposes a comprehensive logcumsumexp algorithm based on the MLU platform, capable of handling tensors of various sizes and different target dimensions, referred to as the MLULCSE algorithm.

MLULCSE divides tasks into four distinct types and, leveraging the hardware characteristics of the MLU platform, devises four corresponding task allocation strategies and implementation schemes. Experimental results show that MLULCSE running on the MLU 370-X4 can keep hardware execution time within seven times that of the PyTorch logcumsumexp running on the Tesla V100. In some test cases, MLULCSE even achieves faster speeds.

In the current implementation, MLULCSE still incurs some computational resource wastage when processing certain task types, due to idle cores or insufficient utilization of computing units' parallelism during vector computation after task allocation. Therefore, to further improve efficiency, constructing a finer-grained task classification system and designing more targeted strategies for logcumsumexp tasks of different types and sizes are promising directions.

Author Biography

Xiaohu Xu is currently a master's student at Guizhou University of Finance and Economics. His research interests include high-performance computing.

Yusen Zhu is currently a senior high school student at The Woodstock Academy, US.

References

[1] Cui, C., Yan, Z., Muhawenayo, G., & Kerner, H. (2024). An all-MLP sequence modeling architecture that excels at copying. *Proceedings of the ICML 2024 Workshop on Next Generation of Sequence Modeling Architectures*, Arizona State University & Colorado State University. (Accepted)

[2] Heinsen, F. A. (2023). Efficient parallelization of an ubiquitous sequential computation. *CoRR*, *abs/2311.06281*.

[3] Yang, X., Abraham, L., Kim, S., Smirnov, P., Ruan, F., Haibe-Kains, B., & Tibshirani, R. (2022, August). *FastCPH: Efficient survival analysis for neural networks*.

[4] Cui, C., Yan, Z., Muhawenayo, G., & Kerner, H. (2025). Linear-time sequence modeling with MLPs. Submitted to the *International Conference on Learning Representations (ICLR)*.

[5] Bassam, E., Zhu, D., & Bian, K. (2025). PLD: A choice-theoretic list-wise knowledge distillation. *arXiv preprint* arXiv:2506.12542.

[6] Lin, J., Wang, W., Yin, L., & Han, Y. (2025). KAITIAN: A unified communication framework for enabling efficient collaboration across heterogeneous accelerators in embodied AI systems. *arXiv preprint* arXiv:2505.10183.

[7] Ma, Z., Wang, H., Xing, J., Zheng, L., Zhang, C., Cao, H., Huang, K., Tang, S., Wang, P., & Zhai, J. (2023). PowerFusion: A tensor compiler with explicit data movement description and instruction-level graph IR. *arXiv preprint* arXiv:2307.04995.

[8] Dong, S., Wen, Y., Bi, J., Huang, D., Guo, J., Xu, J., Xu, R., Song, X., Hao, Y., Zhou, X., Chen, T., Guo, Q., & Chen, Y. (2025). QiMeng-Xpiler: Transcompiling tensor programs for deep learning systems with a neural-symbolic approach. *arXiv preprint* arXiv:2505.02146.

[9] Calafiore, G. C., Gaubert, S., & Possieri, C. (2020). Log-sum-exp neural networks and posynomial models for convex and log-log-convex data. *IEEE Transactions on Neural Networks and Learning Systems*, *31*(3), 827–838. https://doi.org/10.1109/TNNLS.2019.2910417

[10] Miyagawa, T., & Ebihara, A. F. (2021). The power of log-sum-exp: Sequential density ratio matrix estimation for speed-accuracy optimization. In *Proceedings of the 38th International Conference on Machine Learning (ICML)* (Vol. 139, pp. 7792–7804). PMLR.

[11] Chen, Y., & Gao, D. Y. (2016). Global solutions to nonconvex optimization of 4th-order polynomial and log-sum-exp functions. *Journal of Global Optimization*, *64*, 417–431.

[12] Gu, A., & Dao, T. (2023). Mamba: Linear-time sequence modeling with selective state spaces. *arXiv preprint* arXiv:2312.00752. https://doi.org/10.1093/micmic/ozad084

[13] Blelloch, G. E. (1990, November). Prefix sums and their applications. *Technical Report CMU-CS-90-1903*, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA.

[14] Dao, T., & Gu, A. (2024). Transformers are SSMs: Generalized models and efficient algorithms through structured state space duality. *arXiv preprint* arXiv:2405.21060.