

Matrix Functions Computation in Hardware Accelerators

Daniel Oliveira[1(✉)], Juan Acebrón[2], and José Monteiro[3]

[1] INESC-ID, Instituto Superior Técnico - Universidade de Lisboa, Lisbon, Portugal
daniel.virtuoso.oliveira@tecnico.ulisboa.pt
[2] Universidad Carlos III de Madrid, Getafe, Spain
juacebro@math.uc3m.es
[3] INESC-ID, Instituto Superior Técnico - Universidade de Lisboa, Lisbon, Portugal
jcm@inesc-id.pt

Abstract. Solving very large linear systems or computing a function on a large matrix remains challenging due to the memory requirements and scalability limitations of classical numerical methods. Monte Carlo–based algorithms provide an attractive alternative because they naturally enable the computation of only the output elements of interest, significantly reducing memory pressure. In this work, a GPU-oriented Monte Carlo method was developed to solve the target problem directly while addressing the architectural challenges posed by modern GPUs. In particular, many Monte Carlo sampling procedures involve branching and irregular memory access patterns, which significantly degrade GPU performance. To mitigate these limitations, a novel data structure was introduced that combines ideas from Compressed Sparse Row (CSR) representations and Alias Tables. This structure enables efficient sampling while reducing memory overhead, thus improving parallelism on GPU architectures. The proposed approach was validated on real network matrices to evaluate both its accuracy and its scalability in practical large-scale settings.

Keywords: Matrix functions · Monte Carlo · GPU · WRS

1 Introduction

The efficient solution of very large linear systems and computing a function on a large matrix remain fundamental challenges in scientific computing. Classical numerical methods, particularly those based on direct factorizations or Krylov subspace iterations [1], are often limited by memory requirements, irregular data dependencies, and communication bottlenecks when matrices grow to tens or hundreds of millions of rows. In contrast, Monte Carlo–based linear algebra methods [2] provide a compelling alternative: they allow the computation of only the output elements of interest, avoid storing dense intermediate results, and exhibit natural parallelism due to the independence of their sampling procedures. Leveraging on these advantages, this work explores a stochastic approach

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2027
O. Gervasi et al. (Eds.): ICCSA 2026 Workshops, LNCS 16760, pp. 534–551, 2027.
https://doi.org/10.1007/978-3-032-30521-3_34

that directly targets the computation of matrix inverses and resolvent-based quantities while reducing global memory pressure.

At the core of this work is a Monte Carlo algorithm grounded on the probabilistic interpretation of matrix powers [3]. Instead of explicitly forming A^k, the algorithm estimates each desired entry via ensembles of random walks defined over the sparsity pattern of the matrix. This interpretation connected the method to a Taylor polynomial expansion [4] of a matrix function expressed as a finite sum of matrix powers, while avoiding explicit matrix-vector multiplications. Methods based on Krylov subspaces typically required repeated sparse matrix-vector multiplications, global reductions, and strict data-dependency patterns that limits their scalability. In contrast, the Monte Carlo formulation decomposes the computation into a very large number of short, independent random trajectories. This property proves particularly appealing for architectures with massive parallelism, where throughput is maximized by workloads that avoided synchronization and branching.

These characteristics make Graphics Processing Units (GPUs) a natural execution platform. Modern GPUs were designed for SIMD-style workloads with high arithmetic intensity and minimal control-flow divergence, and Monte Carlo sampling initially appeared well suited for such architectures due to its embarrassingly parallel nature. However, naive implementations suffer severe performance penalties. Many Monte Carlo steps require irregular memory accesses, stochastic branching, and row-dependent transition probabilities, all of which reduced SIMD efficiency and introduced warp divergence. Additionally, sampling from rows with widely varying degrees poses load-balancing challenges, and storing transition probabilities in formats such as Compressed Sparse Row (CSR) often result in costly random indexing. Consequently, the theoretical affinity between Monte Carlo algorithms and GPUs did not automatically translate into practical performance gains.

Building on these observations, this work proposes a GPU-oriented data structure that merges the compactness of CSR representations with the constant-time sampling capabilities of Alias Tables [5]. The resulting hybrid representation enables efficient Weighted Random Sampling (WRS) while significantly reducing warp divergence and irregular memory fetches. In addition, strategies for dynamic load balancing across GPU threads were investigated, which proved particularly important for matrices with highly skewed degree distributions. Potential cache-aware optimizations [6] were also explored as a means of further reducing sampling latency and improving overall throughput.

The methods developed in this work target applications in which selected entries of matrix functions encode meaningful structural and dynamical properties of large graphs. In network science, functions over the respective adjacency matrices have been widely used to quantify the importance of nodes and to characterize information flow in complex systems. Among these functions, the matrix exponential plays a central role because its entries naturally aggregate contributions from network walks of all lengths. This property gave rise to well-established measures such as communicability and subgraph centrality, which

have been used to study biological networks, social networks, and infrastructure systems [7]. For example, Estrada demonstrated that communicability derived from the matrix exponential provides a natural measure of how efficiently information propagates between nodes in complex networks.

Focusing on the matrix exponential therefore provided both a representative test case for the proposed Monte Carlo framework and a direct connection to interpretable application-level quantities. To evaluate the accuracy and scalability of the approach, the stochastic estimates were assessed using standard error metrics. In particular, we report the mean squared error and the maximum relative error between the Monte Carlo estimates and reference values, focusing on row-wise quantities of the matrix exponential. Experiments were conducted on synthetic graph models to enable controlled benchmarking across different sparsity patterns and degree distributions, followed by evaluations on large real-world networks to assess robustness and performance under realistic conditions.

2 Background

2.1 Matrix Functions

Matrix functions form a broad class of operations that extend scalar functions to matrices, including polynomials, rational functions, and analytic functions such as the exponential or the logarithm. They appear in many areas of numerical linear algebra and arise whenever a function defined on scalars is generalized to act on matrices through spectral or series-based constructions [8]. Efficient algorithms for evaluating matrix functions depend heavily on the structure of the matrix, such as sparsity, symmetry, or conditioning. In this work, matrix functions are considered through their Taylor polynomial expansions, which express the function as a finite (or truncated) sum of matrix powers; while this framework applies to a broad class of analytic functions, particular attention is devoted to the matrix exponential.

Polynomial series provide a fundamental framework for representing analytic functions as finite or infinite sums of powers. In particular, the Taylor series expresses a sufficiently smooth scalar function $f(x)$ as a polynomial expansion about a reference point x_0, with coefficients determined by the derivatives of the function at that point. Its general form is

$$f(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(x_0)}{n!}(x - x_0)^n, \tag{1}$$

assuming the series converges in a neighborhood of x_0. This construction extends naturally to matrix arguments, allowing matrix functions to be defined through convergent power series in the matrix variable. A canonical example is the matrix exponential, defined by

$$\exp(A) = \sum_{n=0}^{\infty} \frac{A^n}{n!}, \tag{2}$$

for a square matrix A. The factorial normalization ensures convergence of the series for all matrices and causes higher-order powers to decay rapidly, making polynomial truncations an effective representation of the matrix exponential.

2.2 Matrix Representation

In the context of complex networks, matrices are typically dominated by zero entries, making sparse storage essential. For such cases, the Compressed Sparse Row (CSR) format is particularly well suited, as it stores only non-zero elements and thus provides an efficient and effective representation.

The CSR format represents a sparse matrix using three one-dimensional arrays. The first array stores all non-zero values in row-major order. The second array stores the column indices corresponding to each non-zero value. The third array, often called the row pointer array, indicates the starting position of each row within the value and column index arrays. Since the algorithm presented later processes the matrix by rows, this row-based structure is adopted.

2.3 Alias Table

The Alias Table is a probabilistic data structure designed to enable constant-time sampling from a discrete probability distribution. Originally introduced as the Walker alias method [5], it transforms an arbitrary distribution into two lookup tables that allow random sampling in $O(1)$ time.

Let W be the sum of all weights and N the number of outcomes. During the alias table construction we define a target value

$$target = \frac{W}{N}.$$

Each outcome ideally contributes exactly this amount of probability mass. Outcomes whose weight exceeds the target are referred to as heavy, while those with weight below the target are light.

The construction process repeatedly pairs a light outcome with a heavy one. For each pair, a table entry is created that assigns the light outcome its full weight, and fills the remaining probability mass (up to the target) with a reference to the heavy outcome. The weight of the heavy outcome is then reduced by the amount assigned to the table entry; if its remaining weight falls below the target, it is reclassified as a light outcome, otherwise it may be paired again as a heavy outcome. This process continues until all table entries contain exactly the target probability mass.

As a consequence of the construction, each alias table entry contains probability mass from at most two outcomes, the bucket's primary outcome and a single alias, so it is impossible for any slot to contain three or more outcomes [5] (Fig. 1).

As a result, sampling can be performed by selecting a table index uniformly at random and then choosing between the primary outcome and its alias according

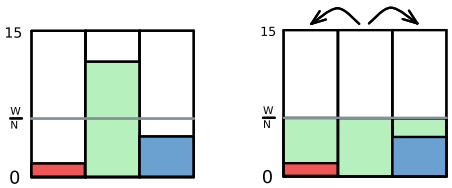

Fig. 1. Alias Table construction.

to the respective weights, using a single additional random draw. This described in Algorithm 1 and it can be observed that each sample is obtained in constant time.

Algorithm 1 Alias Method Sampling

```
i ← uniformly random integer from {0, ..., N − 1}
u ← uniformly random number in [0, 1]
if u < prob[i] then
    return i
else
    return alias[i]
```

Step-by-Step Example.

Consider the three weights , $w = (1, 11, 3)$. Then, $N = 3$, W $= 15$, $target = 5$. For each slot we store:

prob - how much of the slot is filled by the main outcome $\frac{w}{target}$ (a value in [0, 1]).
alias - the fallback outcome that fills the remaining portion of a slot beyond the main outcome.

For the example above, the alias table would be:

index	0	1	2
$prob[i]$	$1/5 = 0.2$	1	$3/5 = 0.6$
$alias[i]$	1	-	1

Interpretation:

- Index 0 contains 20% *prob*(ability) of itself (0) being the outcome and 80% of outcome being its *alias* (1)
- Index 1 contains 100% *prob*(ability) of itself (1) being the outcome

– Index 2 contains 60% *prob*(ability) of itself (2) being the outcome and 40% of outcome being its *alias* (1)

Since each slot is chosen with probability $\frac{1}{3}$, and the internal split reflects the original weights, the resulting samples follow the original distribution:

$$P(0) = \frac{1}{3} \cdot \frac{1}{5} = \frac{1}{15}, P(1) = \frac{1}{3} \cdot (\frac{4}{5} + \frac{5}{5} + \frac{2}{5}) = \frac{11}{15}, P(2) = \frac{1}{3} \cdot \frac{3}{5} = \frac{3}{15} \quad (3)$$

2.4 Network Metrics

A large number of metrics have been proposed to characterize complex networks, including degree-based measures, path-based distances, spectral indices, and centrality measures [9]. Many of these metrics are most naturally expressed in terms of matrix representations of networks, where connectivity patterns are encoded in an adjacency matrix. Among these, a particularly expressive family of metrics is derived from the matrix exponential. Because the matrix exponential naturally aggregates contributions from walks of all lengths, giving more weight to shorter walks, it provides a mathematically principled way to encode both local and global connectivity in a single operator. Several metrics can be derived from the matrix exponential, although only a subset is used in the present study.

The entries of the matrix exponential themselves have direct network interpretations. The diagonal entry $(e^A)_{ii}$ measures how strongly node i participates in closed walks, and can be interpreted as a form of subgraph centrality. Off-diagonal entries $(e^A)_{ij}$ quantify how strongly nodes i and j are connected through all possible paths in the network. Together, these entries provide fine-grained, node-level and pairwise information about network structure.

2.5 GPU

A Graphics Processing Unit (GPU) is a highly parallel processor composed of many lightweight cores designed to execute thousands of threads simultaneously. In contrast to CPUs, which optimize for low-latency execution of a few complex tasks, GPUs prioritize high throughput on large batches of similar operations.

GPUs follow a single-instruction-multiple-thread (SIMT) execution model, where groups of threads execute the same instruction concurrently on different data elements. Although each thread has its own program state, instructions are issued in lockstep for a group of threads, enabling efficient data-parallel execution. This execution model allows GPUs to exploit massive parallelism while maintaining a relatively simple control structure. GPUs typically share access to a unified global memory space, complemented by smaller, faster on-chip memories.

This architecture makes GPUs particularly effective for workloads that can be expressed as data-parallel computations, such as scientific simulations, machine learning, and image processing.

The parallel nature of GPU hardware strongly influences how algorithms must be designed and implemented. Threads on a GPU are organized into warps,

which are fixed-size groups of threads (typically 32 threads on NVIDIA GPUs) that execute instructions simultaneously in the SIMT model. Multiple warps are scheduled on a streaming multiprocessor (SM) to hide memory latency by switching execution when one warp stalls.

Because all threads within a warp execute the same instruction at a given time, branch divergence occurs when threads within a warp follow different execution paths due to conditional statements. In such cases, divergent paths are serialized, reducing effective parallelism and leading to performance loss. Similarly, inefficient memory access patterns, such as uncoalesced global memory accesses where threads in a warp access scattered memory locations, result in multiple memory transactions instead of a single efficient one, significantly increasing memory latency. Furthermore, insufficient parallel workload limits the GPU's ability to fully occupy its processing units, preventing effective latency hiding and under-utilizing available computational resources.

As a result, GPU performance depends not only on raw computational capability but also on how well memory accesses are organized, how control flow is aligned across warps, and how much parallelism can be exposed. GPU programming therefore requires careful consideration of the memory hierarchy, thread synchronization, and workload distribution to fully exploit the hardware.

3 Related Work

3.1 A Distributed Monte Carlo Based Linear Algebra Solver Applied to the Analysis of Large Complex Networks

Magalhães et al. [10] propose a distributed Monte Carlo (MC) framework for computing selected entries of matrix functions arising in large-scale network analysis, with a particular focus on matrix inverses and resolvent-based quantities. Their method builds on the classical probabilistic interpretation of matrix powers introduced by Ulam–von Neumann [3], where powers of a matrix are estimated through ensembles of random walks. By truncating the Neumann series expansion and estimating each term stochastically, the approach avoids explicit dense matrix formation and enables the computation of specific rows or entries. This selective computation is particularly well suited for network metrics such as Katz centrality and the resolvent Estrada index.

A key contribution of their work is a distributed-memory parallel implementation. The sparse matrix is partitioned row-wise across multiple compute nodes, and random walks migrate between nodes via message passing. While the independence of walks provides excellent parallel scalability, this model introduces communication overhead due to walk migration. To mitigate this, the authors propose buffering strategies, asynchronous communication, and workload-aware data partitioning. The resulting MPI/OpenMP implementation demonstrates strong scalability on large supercomputing systems, handling matrices with tens of millions of rows.

In addition, the authors introduce several optimizations, including reuse of partial random walks, simultaneous computation of multiple right-hand sides,

and memory-efficient formulations tailored to network metrics. Their results show that, although deterministic Krylov-based methods remain preferable for single linear solves, Monte Carlo methods can outperform classical approaches when computing traces, multiple inverse-vector products, or resolvent-based quantities at scale.

The present work adopts the same Monte Carlo random-walk framework, where matrix powers are estimated via Markov chains over the sparsity pattern and combined through a series expansion. The main algorithmic difference lies in the choice of expansion: while Magalhães et al. rely on Neumann (resolvent-based) series, this work targets the Taylor expansion of the matrix exponential, leading to different weighting of sampled paths.

More importantly, the two approaches differ in their computational model. Magalhães et al. focus on distributed-memory systems, where data is explicitly partitioned across nodes and communication is required to propagate random walks. In contrast, this work targets shared-memory architectures, particularly GPUs, where all data resides in a common memory space and parallelism is achieved through massive thread-level concurrency. As a result, the primary challenges shift from communication overhead to efficient memory access, load balancing, and minimizing warp divergence during sampling.

Despite these differences, several insights from the distributed setting remain relevant. In particular, strategies for managing large-scale data and irregular workloads can inform out-of-core GPU implementations, where data movement between host and device memory plays a role analogous to communication in distributed systems.

3.2 Skywalker: Efficient Alias-Method-Based Graph Sampling and Random Walk on GPUs

Wang et al. introduce Skywalker [11], a high-throughput GPU framework for large-scale graph sampling and random-walk generation. Random walks are a fundamental primitive in many graph analytics and representation learning algorithms, yet implementing them efficiently on GPUs is challenging due to irregular memory access patterns, load imbalance, and the cost of sampling from high-degree vertices. Skywalker addresses these challenges through a carefully designed system that combines alias-method-based sampling, GPU-oriented data structures, and dynamic load-balancing mechanisms to fully exploit GPU parallelism.

A key innovation of the framework is the use of the alias method for constant-time sampling of discrete probability distributions associated with graph transitions. While the alias method is widely used in probabilistic sampling, its efficient implementation on GPUs is nontrivial because constructing and accessing alias tables can incur substantial memory overhead and irregular memory accesses. Skywalker proposes a GPU-friendly alias-table representation and an optimized construction procedure that significantly reduces both preprocessing time and memory usage. This enables efficient sampling even for large graphs with highly

skewed degree distributions, where naive approaches would lead to severe performance degradation.

In addition to improving sampling efficiency, the system introduces several GPU-specific optimizations to mitigate load imbalance during random-walk execution. Because walks may vary in length and may encounter vertices with widely differing degrees, threads can experience significant divergence. Skywalker addresses this issue through dynamic workload management and warp-aware scheduling techniques that redistribute work among GPU threads to maintain high utilization. Together with memory optimizations for storing graph structures and sampling tables, these techniques enable the framework to achieve substantial performance improvements over prior CPU and GPU implementations of graph sampling workloads.

The present work shares several conceptual and algorithmic similarities with the Skywalker framework. Both approaches rely on random walks over sparse graph structures and must address similar challenges related to irregular memory access, stochastic branching, and GPU thread divergence. In particular, Skywalker's use of alias-based sampling directly relates to the hybrid CSR-alias data structures explored in this work for accelerating transition sampling in Monte Carlo estimators of matrix functions. However, the computational objectives of the two approaches differ significantly. Skywalker is designed as a general-purpose graph sampling engine supporting applications such as representation learning, whereas our work focuses on Monte Carlo estimators for matrix functions, particularly the matrix exponential, where random walks serve as a mechanism for approximating matrix-power expansions. Consequently, while Skywalker emphasizes throughput for large numbers of independent walks, the present work integrates similar sampling ideas within a stochastic linear-algebra framework aimed at estimating specific entries and traces of matrix functions.

4 Monte Carlo Approximation of the Matrix Exponential

The algorithm adopted in this work is based on the Monte Carlo matrix power method described in Sect. 3.1 While the original formulation is primarily presented in the context of matrix inversion and resolvent-based metrics, the same principles naturally extend to the computation of the matrix exponential.

4.1 Matrix Normalization

In order to ensure numerical stability of the Monte Carlo estimator, the input matrix is scaled prior to the simulation. In particular, let

$$\alpha = \max_i \sum_j |A_{ij}| \tag{4}$$

denote the maximum absolute row sum of A. We consider the normalized matrix

$$\widetilde{A} = \frac{1}{\alpha} A. \tag{5}$$

This scaling ensures that the row sums of $\widetilde{A}$ are bounded by 1, which in turn guarantees that the multiplicative factors accumulated during the random walks (cf. Algorithm 3) remain bounded. In particular, the function GetRowWeight returns values in $[0, 1]$, preventing exponential growth of the walk contributions and reducing estimator variance.

The matrix exponential of the original matrix can then be recovered as

$$e^A = e^{\alpha \widetilde{A}} = \sum_{k=0}^{\infty} \frac{\alpha^k}{k!} \widetilde{A}^k, \tag{6}$$

which corresponds to reweighting the contributions of each sampled path by the factor α^k. This normalization step is therefore essential for maintaining both numerical stability and efficient Monte Carlo estimation.

4.2 Random Walk Interpretation

Following the formulation of [12], each entry of A^k can be expressed as the expected value of a random walk of length k over the indices of the matrix.

A random walk starting at row i evolves according to transition probabilities proportional to the entries of the current row. The contribution of a single walk is represented by a multiplicative random variable whose expected value corresponds to an entry of the matrix power A^k. By averaging over p independent walks (referred to as *plays*), the expectation can be approximated by a Monte Carlo estimator whose statistical error decreases proportionally to $\mathcal{O}(p^{-1/2})$.

When computing the matrix exponential, each estimated matrix power is further scaled by the corresponding Taylor coefficient $1/k!$, ensuring consistency with the series expansion.

4.3 Algorithm Overview

The algorithm proceeds row-wise. For each row i, the approximation of the corresponding row of e^A is obtained by accumulating Monte Carlo estimates of the powers $A^k/k!$ for $k = 2, \ldots, n$.

Algorithm 2 Monte Carlo Exponential Matrix

Require: A (matrix), n (series length), p (number of plays)
Ensure: B (approximation of e^A)

```
function ExponentialMatrix(A, n, p)
    N ← size(A)
    for i = 0 to N − 1 do
        B_{i*} ← [0, ..., 0]
        for k = 2 to n do
            B_{i*} ← B_{i*} + CalculateRow(A, i, k, p)
    return B
```

Algorithm 3 Monte Carlo Row Power Computation

```
function CalculateRow(A, i, k, p)
    N ← size(A)
    r[0 : N − 1] ← [0, . . . , 0]
    for m = 1 to p do
        value ← 1
        currentRow ← i
        for l = 0 to k − 1 do
            value ← value × GetRowWeight(A, currentRow)
            selectedColumn ← RandomWeightedChoice(A, currentRow)
            currentRow ← selectedColumn
        r[currentRow] ← r[currentRow] + value
    for j = 0 to N − 1 do
        r[j] ← r[j]/p
        r[j] ← r[j]/k!
    return r
```

The core computation is performed by Algorithm 3, which estimates a single row of A^k using Monte Carlo random walks.

An inspection of Algorithm 3 reveals that the dominant computational cost arises from the repeated sampling operations(RandomWeightedChoice) required during the random walks. For each row i and each power k, the algorithm performs p independent walks of length k, resulting in $\mathcal{O}(p \cdot k)$ sampling steps per row.

4.4 Random Weighted Choice

Let $\mathcal{S} = \{s_1, s_2, \ldots, s_d\}$ be a finite set of outcomes associated with non-negative weights $w_1, w_2, \ldots, w_d$. The RandomWeightedChoice operation, as used in Algorithm 3, selects an element $s_i \in \mathcal{S}$ with probability

$$P(s_i) = \frac{w_i}{\sum_{j=1}^{d} w_j}. \tag{7}$$

A standard implementation of RandomWeightedChoice proceeds by first normalizing the weights and constructing the cumulative distribution function (CDF). More precisely, let

$$c_i = \sum_{j=1}^{i} \frac{w_j}{\sum_{k=1}^{d} w_k}, \tag{8}$$

with $c_d = 1$. Sampling is then performed by drawing a uniform random variable $u \sim \mathcal{U}(0, 1)$ and selecting the smallest index i such that $c_i \geq u$. This can be implemented efficiently using binary search over the cumulative array, resulting in a time complexity of $\mathcal{O}(\log d)$ per sample.

While this approach is simple and widely used, it introduces logarithmic overhead and requires accessing multiple memory locations during the search. In the context of Monte Carlo random walks, where this operation is executed repeatedly, such costs become a significant performance bottleneck.

The Alias Table method provides an alternative sampling strategy that enables exact RandomWeightedChoice in constant time $\mathcal{O}(1)$ after a linear preprocessing step. However, classical alias tables assume dense index sets and do not integrate naturally with sparse matrix representations such as CSR.

4.5 Implementation Details

While the previous subsection focused on the probabilistic formulation of the sampling procedures, the practical performance of the method depends strongly on how these operations are implemented on parallel hardware.

The base of our GPU implementation consists of a single kernel that is launched where each block is assigned a row. Within each block, a strided loop processes the plays, after which the result is multiplied by the vector and stored. Shared memory is used at the warp level to store partial results from each thread, followed by a warp-level reduction and a final reduction across warps to obtain the final value. As for the matrix information, it uses a read-only cache method to capitalize on this propriety that the matrix information has.

For the binary search, the values are normalized and accumulated per row, allowing sampling through a uniformly generated random number followed by a standard binary search over the cumulative row weights. For the alias version it is implemented as described in Sect. 5

5 CSR–Alias Sampling Structure

To address the sampling bottleneck described above, we propose a hybrid data structure that enables constant-time weighted sampling directly from sparse matrix rows stored in Compressed Sparse Row (CSR) format.

5.1 Hybrid CSR–Alias Representation

Our approach embeds alias tables directly into the CSR representation. The standard CSR arrays p_row and *colIdx* are preserved unchanged and continue to encode the sparse adjacency structure.

Instead of storing raw edge weights in the values array, each row is preprocessed into an Alias Table consisting of the two auxiliary arrays described in Sect. 2.3:

prob: probability threshold for the primary outcome
alias: fallback index used when the threshold test fails

Each row therefore stores a lightweight alias table corresponding to its non-zero entries. Since a row with degree d requires $\mathcal{O}(d)$ storage, the overall memory complexity remains

$$\mathcal{O}(E + N), \tag{9}$$

where E denotes the number of non-zero entries and N the number of rows. This matches the asymptotic storage complexity of standard CSR while enabling constant-time weighted sampling.

5.2 Example

Consider the first row shown in Fig. 2 containing three non-zero weights $(1, 11, 3)$. During preprocessing, these weights are converted into the new CSR-Alias representation.

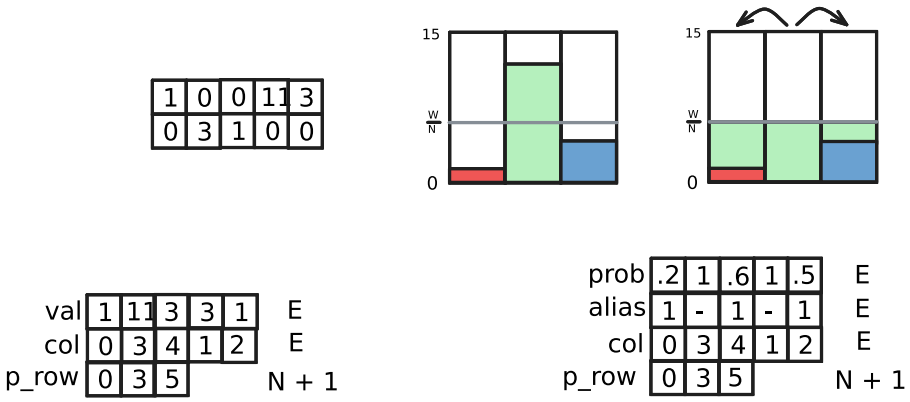

Fig. 2. Comparison between standard CSR representation (left) and the proposed CSR-Alias structure (right).

Each position in the table (right side) represents an equal share of the total probability mass. Sampling from this row proceeds as follows:

1. select an index uniformly at random,
2. compare a random number with the stored probability threshold,
3. redirect to the alias entry if necessary.

This procedure produces the correct weighted outcome in constant time.

5.3 Memory Overhead

An important aspect of the proposed representation is that it preserves the memory efficiency of the standard CSR format. For the example above, a standard

CSR2 (left side) representation of the row would store *rowPtr*, that encodes the starting and ending offsets of the row and the column and value arrays.

In the proposed CSR–Alias representation2 (right side), the structural components remain unchanged, and only the values array is replaced.

Thus, for each non-zero entry, the original scalar weight is replaced by two values: a probability threshold and an alias index. While this increases the per-entry storage by a constant factor, the overall memory complexity remains $\mathcal{O}(E)$, with E denoting the number of non-zero entries.

Importantly, the *rowPtr* and *colIdx* arrays are identical in both representations, ensuring that the sparse structure and traversal properties of CSR are fully preserved. In practice, the additional memory overhead is modest and is offset by the elimination of auxiliary sampling structures, such as cumulative distribution arrays, as well as by the improved runtime performance.

5.4 Parallel and GPU Suitability

The alias-based sampling procedure is particularly well suited for massively parallel architectures. Each sampling step consists of two random number generations, two memory accesses, and a single comparison. Unlike cumulative sum methods, the procedure avoids sequential memory access patterns and conditional branching, enabling warp-consistent execution on GPUs and improving parallel efficiency in large-scale stochastic simulations.

6 Results

This section evaluates the algorithm described in Sect. 4 in terms of numerical accuracy and computational performance on a GPU, highlighting the proposed CSR-Alias sampling structure. We first describe the execution environment, then analyze the estimator precision, and finally present runtime improvements.

6.1 Experimental Environment

All experiments were conducted on Deucalion, where the GPU is A100-40. The implementation was written in CUDA and compiled with `-O3` optimization.

For time measurements the value were taken from when the code starts building the Alias Table to when it already as the result calculated, but not printed. The Alias Table creation timer only represents a really small fraction of the time. But if for example we would be trying to get a low accuracy for a really big matrix this value could become a significant portion of the results. For the 131k nodes and 20 average degree the time to create the alias table was 30 ms. If this time becomes significant then Weighted Random Sampling on GPUs [6] offers some optimizations for this problem.

The Monte Carlo matrix exponential algorithm was executed on four sparse matrices. One matrix, `dictionary28`, was obtained from the SuiteSparse Matrix

Collection, while the remaining three were synthetically generated using graph models.

The *Watts–Strogatz* matrix was generated using the Watts–Strogatz model, consisting of approximately 65,000 nodes and 131,000 edges[1], with a rewiring probability of 0.8, resulting in a small-world structure with high randomness.

The matrix `kron_graph_u` was created using the Kronecker graph model, also with around 65,000 nodes and 171,000 edges (see footnote 1), exhibiting a mostly uniform edge distribution.

Similarly, `kron_graph_c` was generated using the same Kronecker approach and shares the same size 65,000 nodes and 171,000 edges (see footnote 1), but its nonzero entries are concentrated toward the top-left region of the matrix, reflecting a more localized structure .

The parameters used were:

- number of plays p (1M, 2M, 4M)
- maximum walk length n (30)

Unless otherwise stated, each configuration was executed multiple times and the average results are reported.

6.2 Estimator Precision

The Monte Carlo estimator used in this work follows standard convergence behavior, where the estimation error decreases proportionally to

$$\mathcal{O}(p^{-1/2}), \tag{10}$$

with p denoting the number of independent random walks. However, in our setting, the quantity of interest is not an individual entry of the matrix exponential, but rather the sum of each row of e^A. As if we were multiplying by a vector of 1. This corresponds to aggregating all estimated components of the row vector produced by the Monte Carlo procedure.

Since this is obtained as a sum of multiple Monte Carlo estimators, its variance benefits from aggregation effects. Assuming approximate independence between contributions, the variance of the sum decreases proportionally to the number of aggregated terms. In our case, each row accumulates approximately

$$N_{\text{agg}} \approx 6.5 \times 10^4 \tag{11}$$

contributions across different powers and random walks.

As a result, the overall estimation error follows

$$\mathcal{O}\left((p \cdot N_{\text{agg}})^{-1/2}\right), \tag{12}$$

[1] In the respective adjacency matrix representation, each node will have one corresponding row and column, and each undirected edge two non-zero entries in the matrix.

which provides a significant variance reduction compared to estimating individual entries.

Before our experiments, we calculated the error to be

$$\epsilon_i \leq \frac{1}{\sqrt{10^6 \times 10^5}} \tag{13}$$

To validate the accuracy of the estimator, reference values were computed using MATLAB's built-in matrix exponential function. In tests, the average relative error with respect to these reference values was

$$\bar{\epsilon} = 6.6 \times 10^{-6}. \tag{14}$$

These results confirm that the aggregation inherent in the row-wise summation significantly improves estimator stability, yielding accurate approximations even for moderate values of p.

6.3 Performance Evaluation

We compared the proposed CSR-Alias sampling method, as described in the algorithm in Sect. 4, against baseline implementations using binary normalized search. In all executions the relative error remained below 10^{-6} across all tested configurations.

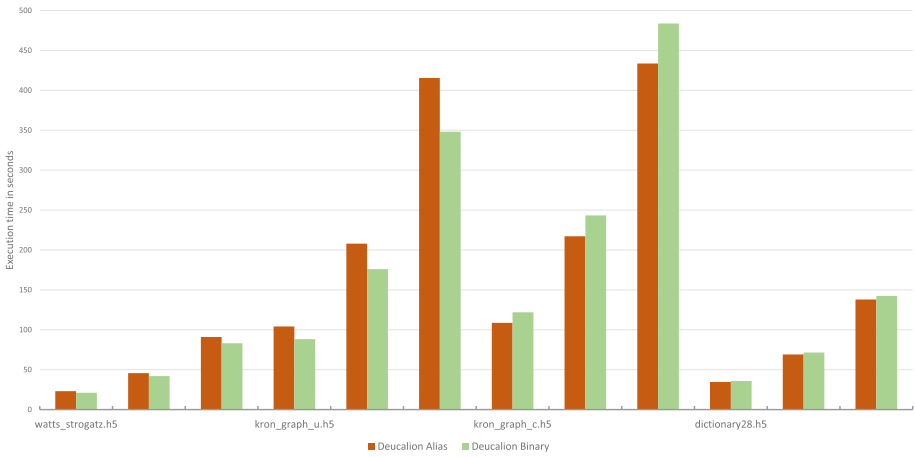

Fig. 3. Execution time comparison between the CSR–Alias implementation and the binary-search-based sampling across different datasets and configurations. Results are shown for varying numbers of plays (1M, 2M, 4M) on four graph instances: *watts_strogatz*, *kron_graph_u*, *kron_graph_c*, and *dictionary28*.

Figure 3 shows the total runtime of the Monte Carlo matrix exponential algorithm for different configurations.

For the first two instances, namely *Watts-Strogatz* and *Kronecker Uniform*, the baseline implementation based on binary search exhibits competitive performance. In these datasets, the majority of rows have very small degree (typically close to 2), which limits the depth of the binary search and reduces the number of branch operations. As a result, the baseline method benefits from low control-flow divergence and reduced memory overhead, leading to performance that is comparable to, and in some cases slightly better than, the alias-based approach.

In contrast, for the remaining instances characterized by higher average degree and more heterogeneous degree distributions, the advantages of the CSR-Alias method become significantly more pronounced. In these cases, the cost of binary search increases with the row size, both in terms of additional comparisons and irregular memory access patterns. The proposed method, by maintaining constant-time $\mathcal{O}(1)$ sampling, avoids this overhead and achieves substantially higher speedups.

Overall, these results highlight that the benefits of the CSR-Alias structure are particularly significant for graphs with moderate to high degrees, where weighted sampling constitutes a dominant portion of the computational cost.

6.4 Discussion of Speedups

The observed performance improvements can be attributed to the reduction in sampling complexity. While binary search sampling requires $\mathcal{O}(\log d)$ time per step, the proposed method achieves constant-time $\mathcal{O}(1)$ sampling.

This difference becomes particularly significant in the context of Monte Carlo simulations, where a large number of sampling operations are performed. Since each random walk consists of multiple steps, even small per-step improvements accumulate into substantial overall speedups.

The largest speedups are observed in matrices with higher average degree, where the cost of traditional sampling methods increases. In contrast, the proposed method maintains constant-time performance regardless of the row size.

7 Conclusion

This work presented a Monte Carlo approach for approximating the matrix exponential of large sparse matrices, together with a novel CSR–Alias data structure designed to accelerate the dominant sampling operation. By integrating alias tables directly into the CSR representation, the proposed method enables constant-time WRS while preserving the sparsity structure and maintaining linear memory complexity.

The experimental results demonstrate that the proposed approach achieves significant speedups in scenarios where weighted sampling becomes a computational bottleneck. In particular, for graphs with moderate to high degree or heterogeneous degree distributions, the CSR–Alias structure substantially reduces runtime by eliminating the logarithmic overhead associated with binary search and improving memory access regularity.

At the same time, the results highlight an important limitation. For graphs with very low and homogeneous degree, such as Watts–Strogatz or uniformly generated Kronecker graphs, the baseline binary-search-based method remains competitive and can even outperform the alias-based approach. In these cases, the small number of neighbors limits the depth of the search, and the reduced memory footprint of the binary representation leads to more efficient execution.

Overall, the proposed method is most beneficial for sparse matrices that are not extremely low-degree, where the cost of repeated sampling dominates the computation. These findings suggest that the choice of sampling strategy should be guided by the structural properties of the input graph.

Acknowledgements. Work supported by national funds through Fundação para a Ciência e a Tecnologia, I.P. (FCT) under projects URA-HPC - 2022.08838.PTDC (DOI: https://doi.org/10.54499/2022.08838.PTDC), UID/50021/2025 (DOI: https://doi.org/10.54499/UID/50021/2025) and UID/PRR/50021/2025 (DOI: https://doi.org/10.54499/UID/PRR/50021/2025).

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Saad, Y.: Iterative methods for sparse linear systems. In: SIAM (2003)
2. Dimov, I., Alexandrov, V., Karaivanova, A.: Parallel resolvent Monte Carlo algorithms for linear algebra problems. Math. Comput. Simul. **55**(1–3), 25–35 (2001)
3. Forsythe, G.E., Leibler, R.A.: Matrix inversion by a Monte Carlo method. Math. Comput. **4**(31), 127–129 (1950)
4. Foy, W.H.: Position-location solutions by Taylor-series estimation. IEEE Trans. Aerosp. Electron. Syst. **2**, 187–194 (2007)
5. Walker, A.J.: An efficient method for generating discrete random variables with general distributions. ACM Transactions on Mathematical Software (TOMS) **3**(3), 253–256 (1977)
6. Lehmann, H.-P., Hübschle-Schneider, L., Sanders, P.: Weighted random sampling on gpus. arXiv preprint arXiv:2106.12270 (2021)
7. Estrada, E., et al.: The structure of complex networks: theory and applications. American Chemical Society (2012)
8. Higham, N.J.: Functions of matrices: theory and computation. In: SIAM (2008)
9. Newman, M.: Networks. Oxford university press (2018)
10. Magalhaes, F., Monteiro, J., Acebron, J.A., Herrero, J.R.: A distributed Monte Carlo based linear algebra solver applied to the analysis of large complex networks. Futur. Gener. Comput. Syst. **127**, 320–330 (2022)
11. Wang, P.: Skywalker: Efficient alias-method-based graph sampling and random walk on gpus. In: 2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT). IEEE, 2021, pp. 304–317 (2021)
12. Ji, H., Mascagni, M., Li, Y.: Convergence analysis of Markov chain Monte Carlo linear solvers using Ulam-von Neumann algorithm. SIAM J. Numer. Anal. **51**(4), 2107–2122 (2013)