eIM: GPU-Accelerated Efficient Influence Maximization in Large-Scale Social Networks

Jacob Doney
Texas State University
San Marcos, TX, USA
jd1568@txstate.edu

Xin Huang
Texas State University
San Marcos, TX, USA
xhuang@txstate.edu

Chul-Ho Lee
Texas State University
San Marcos, TX, USA
chulho.lee@txstate.edu

Abstract

The influence maximization problem seeks to identify a subset of k vertices in a network that, when activated, maximizes the spread of influence under a given diffusion process. It is NP-hard to find the optimal set of influential vertices; thus, recent studies have focused on developing algorithms to find an approximate solution. The state-of-the-art parallel implementations leverage a sketch-based algorithm called influence maximization via martingales (IMM). However, IMM incurs significant memory overhead due to the storage requirements of graph traversal samples called random reverse reachable (RRR) sets. In this paper, we introduce efficient Influence Maximization (eIM), a novel GPU-accelerated IMM algorithm designed to improve the efficiency and scalability of IMM. Compared to two popular GPU implementations, eIM achieves similar accuracy with one to three orders of magnitude speedups while reducing the memory requirement to store network data and RRR sets up to 54%.

CCS Concepts

• **Computing methodologies → Massively parallel algorithms**.

ACM Reference Format:
Jacob Doney, Xin Huang, and Chul-Ho Lee. 2025. eIM: GPU-Accelerated Efficient Influence Maximization in Large-Scale Social Networks. In *Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC Workshops '25), November 16–21, 2025, St Louis, MO, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3731599.3767442

1 Introduction

The influence maximization (IM) problem originates from viral marketing applications [5, 11, 20]. Besides marketing, the IM problem has found applications in network monitoring [12], recommender systems [27], public health interventions [19], and rumor control [24]. It is based on the concept that if certain individuals in a network are selected to promote a product, they can trigger a chain reaction of influence. Their promotion of the product can spread through the network via word of mouth, thus causing a cascading effect that can lead to the product being adopted by their friends and followers. Specifically, for a given network, the IM problem seeks to identify a given number of individuals within the network who have the greatest potential to trigger the largest chain reaction of influence. The optimal solution is called the seed set.

Kempe *et al.* [10] popularized the IM problem when they proved that finding the optimal seed set for any network is NP-hard. This is because the number of possible seed sets required to check grows combinatorially as the number of vertices increases. This led to the development of several earlier polynomial-time approximation algorithms [7, 9, 10, 17]. For example, Kempe *et al.* [10] introduced a greedy hill climbing algorithm to achieve a $(1-1/e-\epsilon)$ approximate solution under the *independent cascade* (IC) and *linear threshold* (LT) models, where ϵ represents the approximation error. Their algorithm, however, requires a large number of Monte Carlo simulations to obtain the approximate solution, and thus is not scalable to large networks due to its computational inefficiency.

Borgs *et al.* [3] developed a sketch-based algorithm to overcome the inefficiencies of Monte Carlo simulations by generating a large number of random reverse reachable (RRR) sets. The RRR sets are generated based on reverse influence sampling (RIS) where each set contains the vertices activated during a single run of the diffusion process in reverse starting from a randomly selected vertex. The seed set is then selected based on the vertices with the maximum coverage over all RRR sets, as it has the potential to activate the largest number of vertices. While faster than the original greedy algorithm, this algorithm still requires a large number of graph traversals and introduces a new requirement of storing all the RRR sets. This storage demand can be memory intensive as the network grows, or when scaling for higher accuracy or larger seed sets.

Tang *et al.* proposed a two-phase influence maximization algorithm [23] to improve the RIS algorithm by using a more efficient formula to determine the required number of RRR sets to ensure the same theoretical approximation. They later improved upon this work by developing the influence maximization using martingales (IMM) algorithm [22], which is considered as a state-of-the-art algorithm. This IMM algorithm provides a framework to determine a tighter lower bound for the number of RRR sets required to guarantee a $(1-1/e-\epsilon)$ approximate solution with high probability. This algorithm iteratively produces RRR sets and evaluates whether the most influential vertices of the samples achieve coverage above a specified threshold. When the threshold is exceeded, the algorithm estimates a lower bound for the required number of RRR sets. However, it has been pointed out in [14] that the IMM algorithm inherited two limitations from the RIS algorithm. It still requires a large number of RRR sets to be generated to guarantee an approximate solution, and all RRR sets must be maintained in memory for the seed set selection phase.

SC Workshops '25, St Louis, MO, USA

ACM ISBN 979-8-4007-1871-7/25/11
https://doi.org/10.1145/3731599.3767442

Recent studies [15, 16, 21] have focused on leveraging parallel computing techniques to further accelerate the IMM algorithm under the IC and LT models. These state-of-the-art parallel algorithms and implementations have demonstrated remarkable speedups in execution time compared to their serial counterparts. However, despite these advancements, significant challenges in computational efficiency remain. The IMM algorithm faces high memory requirements for storing RRR sets, which limit the scalability and effectiveness in solving the IM problem on large-scale networks. Additionally, the creation of RRR sets introduces further challenges to efficient parallelization, including the need to perform thousands of graph traversals of unpredictable lengths. The varying lengths of traversals complicate workload balancing as well as the synchronization required to store the RRR sets in memory.

Our main contribution is the introduction of efficient Influence Maximization (eIM), a GPU-accelerated IMM algorithm designed to address the critical challenges IMM faces with efficiency and scalability under the IC and LT models. First, we propose to use a thread-safe logarithmic (log) encoding method that allows for fast decompression for storing network data and RRR sets. This allows us to save up to 54% of the required memory for storing these two components with minimal impact on the running time. Second, we optimize the generation of RRR sets by leveraging the GPU's global memory, eliminating additional memory allocation that can occur when shared memory becomes exhausted during the graph traversal process. Third, we propose a novel heuristic of removing source vertices from RRR sets to improve efficiency. Lastly, we analyze workload distribution strategies for scanning through RRR sets and develop a customized strategy.

To validate eIM's performance, we conduct extensive experiments on 16 real-world networks, comparing it against two of the most popular GPU accelerated IMM algorithms, cuRipples [15] and gIM [21]. Our results show that eIM outperforms cuRipples and gIM for most networks and parameter settings of the diffusion models. Compared to gIM, eIM achieves speedups of up to 23× for the IC model and 30× for the LT model. Additionally, eIM demonstrates improved scalability compared to gIM, as it successfully produces results in cases where gIM returns out-of-memory errors. When compared to cuRipples, eIM achieves speedups of up to 5746× for the IC model and 9224× for the LT model.

2 Background and Related Work

In this section, we present the IM problem, outline the IC and LT diffusion models, and provide an overview of related work such as IMM algorithm and two popular parallel IMM implementations, gIM and cuRipples.

2.1 Problem Definition and Diffusion Models

Consider a directed graph $G = (V, E)$ that represents a social network where V is a set of vertices with $n = |V|$ and E is a set of edges with $m = |E|$. $(u, v) \in E$ indicates the presence of a directed edge from u to v. We use $N^-(v)$ to indicate the set of in-neighbors of v, i.e., $N^-(v) = \{u : (u, v) \in E\}$, meaning all the vertices that have an outgoing edge to v. Let d_v^- denote the in-degree of v.

If S is a subset of V, then the function $I(S)$ represents the spread of influence by S under a given diffusion model, which is the total number of activated vertices when the diffusion starts from S. The IM problem aims to find a subset of k vertices such that when they are activated, they have the *maximum* expected influence, i.e.,

$$S^* = \arg \max_{S:|S|=k} \mathbb{E}[I(S)].$$

This optimal subset S^* is denoted as the seed set.

We consider two commonly studied diffusion models, namely, the IC and LT models. These models simulate how information may spread through a social network. Both models take place in discrete time steps, where at any step a vertex is either considered *active* or *inactive*. The IC model is used to simulate the spread of information via word of mouth through a network [6]. It assumes that any newly activated vertices have a single chance to activate each of their inactive neighbors. For a given graph G, each edge $(u, v) \in E$ is assigned a random weight p_{uv}, such that u may be able to activate v with probability p_{uv}. During each time step of the diffusion process, every vertex activated in the last time step has a single chance to activate each of its inactive neighbors. The process completes when no neighbors are further activated.

The IC model presents greater memory requirements compared to other diffusion models due to its tendency to propagate deeper into the network and generate larger RRR sets when using the IMM algorithm [4, 15, 18]. Thus, recent works [4, 10, 21, 25] consider assigning edge probabilities in a way that effectively limits the size of each graph traversal. This leads to both a reduction in memory requirements and a speedup in computation time compared to the pure random assignment. Specifically, it assigns the probability of an edge $(u, v) \in E$ to $1/d_v^-$, where d_v^- is the in-degree of vertex v [10]. This probability assignment results in fewer activated vertices and makes the probability that a vertex is activated less dependent on its incoming edges. In this work, we focus on this probability assignment for the IC model.

The LT model [8] simulates the idea that an individual in a network decides whether to "*do a thing or not*" based on whether enough of its neighbors have decided to do "*the thing*." In addition to the IC model, the LT model is another popular diffusion model for the IM problem. In the LT model, given a directed graph $G = (V, E)$, each vertex $v \in V$ in the network is assigned a threshold $\tau_v \in [0, 1]$ uniformly at random. Each edge $(u, v) \in E$ is associated with a weight p_{uv} such that $\sum_{u \in N^-(v)} p_{uv} \leq 1$, where $N^-(v)$ is the set of in-neighbors of v. The diffusion process proceeds in discrete time steps. Letting A_t be the set of activated vertices at the end of time t, vertex $v \in V$ becomes activated if the sum of the weights of its activated in-neighbors at time $t-1$ reaches or exceeds its threshold τ_v, i.e., $\sum_{u \in N^-(v) \cap A_{t-1}} p_{uv} \geq \tau_v$.

2.2 Influence Maximization via Martingales

The IMM algorithm proposed by Tang *et al.* [22] improves the RIS algorithm by calculating a tighter lower bound for the number of RRR sets required to ensure a $(1 - 1/e - \epsilon)$-approximate solution. Their proposed method expands the RIS algorithm by introducing a new step that estimates the required number of RRR sets, θ, to achieve the approximation. The process starts with an initial estimate for the number of RRR set, denoted as $\hat{\theta}$. It generates $\hat{\theta}$ RRR sets and identifies a set of k vertices with maximum coverage. The process continues by computing a new $\hat{\theta}$ until a set of k vertices

Algorithm 1: Influence Maximization via Martingales

Input : G, k, ε
Output: S
1 $\theta \leftarrow \text{EstimateTheta}(G, k, \varepsilon)$
2 $\mathbb{R} \leftarrow \text{Sample}(\theta, G)$
3 $S \leftarrow \text{SeedSelection}(\mathbb{R}, k)$
4 **return** S

with sufficient coverage is found. This coverage is used to calculate the value of θ. The algorithm ends by using θ to generate RRR sets and find the final seed set S. Note that the estimation process can be adjusted using an approximation factor, ε, that, when decreased, will increase the estimation accuracy of $\hat{\theta}$. The procedure of the IMM algorithm is summarized in Algorithm 1.

2.3 Accelerated IMM

To improve the computational efficiency, the use of CPUs and GPUs to parallelize IMM has recently been studied. Two of the most popular accelerated algorithms are cuRipples and gIM [15, 21]. Both implementations leverage GPUs to improve the efficiency of IMM. cuRipples is an updated version of the CPU-only Ripples [16] that allows for the utilization of both CPUs and GPUs simultaneously and is designed for distributed systems. cuRipples shows excellent scaling to larger networks. This is because cuRipples does not store all RRR sets on the GPU, and each CPU-GPU pair creates a subset of the RRR sets, which are offloaded to system memory. When finding a seed set, the RRR sets are moved back on to the GPU's memory until it is full. Any remaining sets that cannot fit within the GPU's memory space are operated on by the CPU. This transfer of data between the CPU and GPU incurs significant overhead and results in higher computation time.

On the other hand, gIM is designed to be run on a singular GPU. The data for the RRR sets is now stored in the GPU's global memory, which removes any communication overhead between the CPU and GPU and eliminates the need to transfer data. gIM's performance enhancements also come from its edge-level parallelization of the breadth first search (BFS) traversal. This works by assigning each warp (32 threads) to generate a single RRR set. When a warp visits a vertex during the BFS traversal, each thread within the warp attempts to activate a neighboring vertex. If the neighbor is activated, it will be atomically added to the warp's BFS queue located in shared memory. However, due to the limited size of shared memory, when this memory reaches capacity, it must be offloaded into global memory. In addition, global memory is used to store RRR set data. The active vertices are written from the queue to a temporary RRR set in global memory, which is eventually copied to the final collection of RRR sets. The main drawback of this implementation is repeated dynamic memory allocations, which introduce overhead and can eventually exhaust the GPU's memory.

Recent studies have focused on improving gIM, Ripples, and cuRipples by optimizing the BFS traversal process [18], improving workload balance on multi-CPU systems [26], or reducing the space requirement for RRR sets in system memory [4]. In contrast, this work aims to address the challenges of using GPU memory, making it complementary to these recent studies. Specifically, we introduce eIM, an optimized parallel implementation of the IMM algorithm

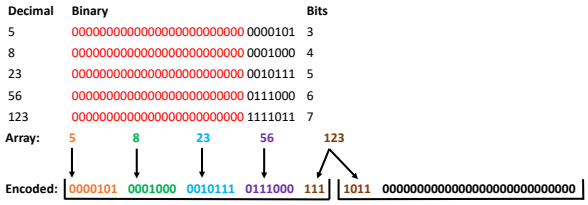

Figure 1: Example of log encoding on an array of five integers.

by leveraging log encoding, employing a novel heuristic of removing source vertices, and judiciously using the GPU's memory and resources. We demonstrate that eIM substantially reduces memory overhead and improves computational efficiency on a GPU.

3 eIM: Proposed Optimization

In this section, we describe the integral components of eIM. Here we go into detail on log encoding, removal of source vertices, and our methods for generating RRR sets and selecting the seed set.

3.1 Log Encoding

To reduce the memory requirements of IMM, our eIM algorithm utilizes a thread-safe implementation of log encoding. This encoding is applied to both the network data and the RRR sets. The network data is initially represented in the compressed sparse column (CSC) format since IMM uses the RIS method to generate RRR sets. The CSC representation consists of three arrays, each of which has offsets, in-neighbors, and edge weights, respectively. The RRR sets are stored in a single array, denoted as $\mathbb{R}$, that holds all the vertices in each of the RRR sets. An offset array, denoted as O, is used to indicate the starting index for each RRR set in $\mathbb{R}$. While other studies have used different compression techniques such as Huffman coding and bitmap coding that result in a reduction in the memory footprint of $\mathbb{R}$, they have only been used on CPUs [4]. In this work, we leverage log encoding (or bit-packing) [1, 2] on GPUs due to its fast decompression and reduced cache misses.

Log encoding is a simple technique that removes the leading zero bits from all values within an array and concatenates the remaining bits [2]. To determine the number of bits that are required to represent each value in the array, the largest value must be identified. Specifically, the number of required bits, say, n_b, needed to represent each value in the array can be determined based on $n_b = \lceil \log_2(x_{\max}) \rceil$, where $x_{\max}$ denotes the maximum element in the array. Figure 1 illustrates this operation on an array of integers. In the example, $x_{\max}$ is 123, which can be represented using 7 bits. The leading zeros are removed, and the values are compressed into two 32-bit containers. This encoding reduces the number of bits representing the array of five integers from 160 bits to 64 bits. Since the number of bits required to represent all elements might not be a multiple of 32, some bits may need to span two containers.

3.2 RRR Set Sampling under the IC Model

The set sampling phase of the IMM algorithm generates θ RRR sets, where θ is the estimated number of required RRR sets. As mentioned in Section 2.2, an RRR set consists of all vertices activated during a backward run of a given diffusion model. The generation of individual RRR sets is independent and can be created with minimal communication between threads. However, several challenges

Algorithm 2: Parallel RRR set sampling under the IC model

```
   Input  : G(V, E), ℝ, θ, C, O, offset, count
   Output : set of RRR sets R, array of counts C
1  for each block in parallel do
2      for each v ∈ V in parallel do
3          M[v] ← 0;
4      while count < θ do
5          if tid = 0 then
6              s ← RandomElement(V);
7              M[s] ← 1;
8              Q[0] ← s;
9              q_head ← 0;
10             q_tail ← 1;
11         while q_head < q_tail do
12             if tid = 0 then
13                 u ← Q.front();
14                 q_head ← q_head + 1;
15             for each neighbor v ∈ N⁻(u) in parallel do
16                 r ← Random(0, 1);
17                 if r ≤ p_vu and M[v] = 0 then
18                     M[v] ← 1;
19                     Q.atomicEnqueue(v);
20                     atomicAdd(q_tail, 1);
21         old_offset ← atomicAdd(offset, |Q|);
22         O[count + 1] ← old_offset + |Q|;
23         for i ← tid; i < |Q|; i ← i + warpSize do
24             v ← Q[i];
25             atomicAdd(C[v], 1);
26             ℝ[old_offset + i] ← v;
27             M[v] ← 0;
28         atomicAdd(count, 1);
```

exist when trying to parallelize this algorithm. For example, load imbalances between blocks of threads can occur as traversal lengths can drastically vary. This drastic size differences of RRR sets introduce complications when adding them to the collection of RRR sets $\mathbb{R}$. Communication is necessary while adding sets into $\mathbb{R}$, to avoid overwriting values or wasting space between sets.

Similar to gIM under the IC model [21], in our eIM algorithm, each block performs a single 'warp-wide' BFS, but it has several key differences in its design to efficiently manage larger graph traversals. eIM is designed to better accommodate for larger and deeper graph traversals through the implementation of a global memory pool. While accessing shared memory is faster than going to global memory, it tends to not be able to fit entire RRR sets, resulting in the need to dynamically allocate additional space. Our algorithm uses global memory for the BFS traversal queue. This offers several advantages over using shared memory. It eliminates the need for dynamic memory allocation as the queue grows and for additional storage to hold the RRR set while it is being generated. Since we maintain all activated vertices in the queue, when the traversal finishes we can transfer the queue directly to $\mathbb{R}$.

Algorithm 2 shows the pseudocode for the RRR set sampling phase in eIM under the IC model. Each block within the GPU

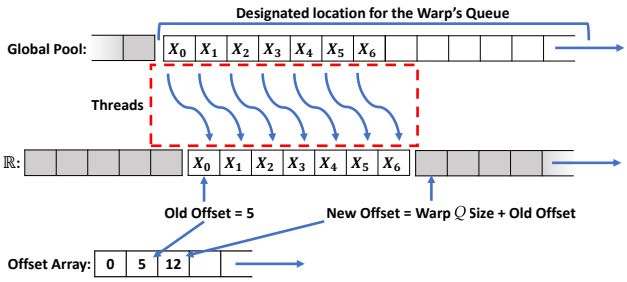

Figure 2: Example of transferring data from Q to $\mathbb{R}$.

launches a single warp, or 32 threads. These warps are responsible for creating the RRR sets and adding them to $\mathbb{R}$. The estimated number of required RRR sets, θ, and the current number of generated RRR sets, denoted as count, are shared between all the blocks. Each block is allocated a section within the global memory for storing its own BFS queue Q.

Each warp starts by initializing a shared binary array, say, M, that indicates if a vertex within G has been activated or not. After this initialization, thread 0 within each warp randomly selects a starting vertex, marks it as activated, and adds it to its queue Q. It also initializes its queue's head and tail pointers. Thread 0 starts the BFS traversal by indicating the currently visited vertex u by reading the front of the queue (without removal) and updating the head pointer. The block proceeds by performing a warp-wide probabilistic BFS, where each thread within the warp visits an unvisited in-neighbor v of u. Each thread generates a number from $[0, 1]$ uniformly at random. The neighbor v is added to Q and marked as visited if the random number is less than or equal to its activation probability p_{vu}. In other words, the neighbor is activated with probability p_{vu} and added to Q. We mark the neighbor as visited in M before atomically adding it to Q. This ordering is important to avoid adding the same vertex to Q multiple times. The last step is to atomically increase the queue's tail pointer. The BFS traversal will stop adding vertices to Q, and the head pointer will reach the end.

Since an RRR set is equivalent to the visited array of a probabilistic BFS traversal [18], we can copy Q directly to $\mathbb{R}$. Each warp shares a running sum of the total number of activated vertices within $\mathbb{R}$, denoted as offset in Algorithm 2. This value can also be used to indicate the index within $\mathbb{R}$ where the next RRR set should be placed. Thus, to copy Q to $\mathbb{R}$, we retrieve its current value and store it in shared memory, under the variable old_offset, while we atomically increase the value of offset by the size of Q (line 22). This operation returns the value of offset before the atomic addition and updates offset with the new value. The updated value tells the next block where to start adding vertices. Next, we update an offset array, denoted as O, maintaining the starting point for each RRR set in $\mathbb{R}$. Lastly, in parallel each thread within the warp copies a vertex over from Q to $\mathbb{R}$, atomically updates the vertex's count in C, and resets its value in M. The process of copying the vertices from Q to $\mathbb{R}$ is illustrated in Figure 2.

When transferring the elements (vertices) from Q to $\mathbb{R}$, we add them in ascending order by vertex ID. This ordering enables us to use a binary search operation during the seed selection phase when updating vertex frequency counts. We empirically observed that the benefit of using binary search in the seed selection phase

outweighs the overhead of sorting the elements in each queue Q when they are added into $\mathbb{R}$. After copying the values, we atomically update the count of currently generated RRR sets. If this count is less than θ, we continue and generate a new RRR set. This will result in a round robin assignment of RRR set creation between the GPU blocks, balancing the workload.

3.3 RRR Set Sampling under the LT Model

The RRR set sampling algorithm under the LT model is similar to that of the IC model in its use of global memory for storing the traversal data. The algorithm also adds vertices in each RRR set to $\mathbb{R}$ in ascending order to leverage binary search in the seed selection phase. However, the LT model's algorithm differs in how it forms RRR sets. Unlike the IC model, where a vertex v can potentially activate all of its inactivated in-neighbors, the set of neighbors that v can activate in the LT model is either $\emptyset$ or contains a single in-neighbor [21–23]. In each iteration, if the sum of edge weights of v's incoming neighbors exceeds its threshold τ_v, the first incoming neighbor whose edge weight causes the running sum to exceed τ_v is activated.

In our eIM algorithm for the LT model, we modify the second while loop (lines 11–20) in Algorithm 2. Thread 0 reads an activated vertex u from the front of the queue Q and assigns it a random threshold τ_u uniformly in the range [0,1]. This threshold is shared across all threads within the warp. Each thread retrieves a weight p_{vu} from an inactivated in-neighbor v of u. The thread calculates both the inclusive sum, i.e., the sum of weights from ranks smaller than or equal to itself, and the exclusive sum, i.e., the sum of weights from ranks strictly smaller than itself. A neighbor v is activated and added to Q if its inclusive sum is greater than τ_u and its exclusive sum is less than τ_u, indicating that it is the first neighbor to exceed the threshold. The while loop terminates if no neighbor is activated during an iteration.

Two algorithms for the LT model were explored for identifying the activating vertex. The first uses atomic addition, where each thread is assigned a neighbor and adds their edge weight to a shared sum. This method was slow as it introduced serialization into the algorithm as each thread will wait until they are able to update the shared sum. The second method uses parallel prefix sum (scan), which enables parallel calculation of cumulative weights. Instead of threads waiting to update a single shared variable, each thread maintains their own running sum and uses __shfl_up_sync to share their weight with other threads within its warp. Each thread first adds the value from the thread before it to its own, then adds the sum of two positions before its own, then four, and so on, doubling the distance each time. This effectively reduces the number of iterations from $O(d_u^-)$, which is when using atomic addition, to $O(\log d_u^-)$. The vertex whose running sum first exceeds the threshold is selected to be activated and is added to Q.

3.4 Source Vertex Elimination

We observe that many singleton sets appear within $\mathbb{R}$, where a singleton set is a set that contains only one element. These singleton sets offer limited value for identifying the most influential vertices because the influence of a seed set cannot cover a singleton set unless the seed set includes that specific vertex. This leads to a lower overall coverage ratio and necessitates the generation of additional RRR sets, slowing down the entire process. Another disadvantage of having long sequences of singleton sets is that they can introduce thread divergence and serialization within GPU blocks during the seed selection phase.

Algorithm 3: Parallel update of counts for IC and LT

```
Input  : G(V, E), ℝ, θ, C, F, O, v
Output : updated array of counts C
1  for each thread tid in parallel do
2      stride ← bdim + gdim;
3      for i ← tid; i < θ; i ← i + stride do
4          if F[i] = 0 then
5              start ← O[i];
6              end ← O[i + 1];
7              found ← binary_search(start, end)
8              if found then
9                  F[i] ← 1;
10                 for j ← start; j < end; j ← j+1 do
11                     u ← ℝ[j];
12                     atomicSub(C[u], 1);
```

To address this issue, we introduce a heuristic of eliminating the source vertex for each set. The source vertex is the randomly selected vertex from which the diffusion process in reverse begins. Since source vertices are selected uniformly at random, removing them does not introduce any bias into the sampling process. Instead, it significantly reduces the number of singleton sets while preserving the overall structure and quality of the RRR sets. This is because each RRR set still contains all vertices that have the potential to activate the source vertex (in the forward process), preserving the influence information. As shown in Section 4.3, this optimization not only reduces the number of generated RRR sets for most networks, but also improves running time performance. However, for some networks, it may result in a greater total number of elements in $\mathbb{R}$, despite the reduction in set count.

3.5 Parallelization of Seed Selection

Once RRR sets are generated, what remains is to select the seed set S. The seed selection algorithm uses a greedy strategy to select a set of k vertices that maximize the coverage over $\mathbb{R}$. It iteratively adds vertices to the seed set S until k vertices have been selected. In each iteration, the vertex that appears in the most RRR sets is added to S, and all the RRR sets in $\mathbb{R}$ covered by the selected vertex are removed. This allows for the algorithm in sequential iterations to focus on only the RRR sets that are uncovered.

During the sampling phase, each vertex is added to its corresponding RRR set in ascending order by vertex ID, and its count is updated in C, an array that tracks how frequently each vertex appears across all RRR sets. This count correlates to a vertex's influence. In the seed selection algorithm, we first select the vertex with the highest count, i.e., $\arg\max_u C[u]$, and add it to the seed set S. To find the next vertex that adds the most marginal gain to S, we update the count for each vertex by removing the influence of the previously selected vertex, say, v. This includes identifying all RRR sets containing v and decrementing the counts of other vertices in

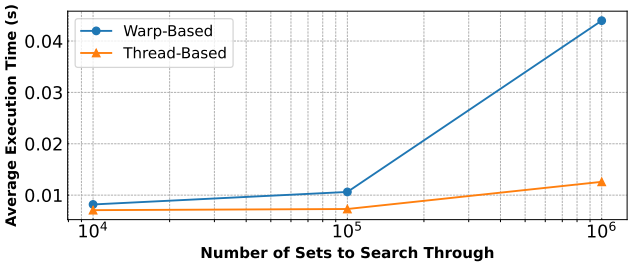

Figure 3: Scalability of the thread-based approach and the warp-based approach as N increases ($k = 100$).

those sets. We assign one thread per RRR set to determine if v is present using binary search. To track which sets are covered by S, we use a binary array F, where each bit indicates whether its corresponding RRR set has been covered. The procedure is outlined in Algorithm 3.

We observe that by assigning a single thread to search for v scales more efficiently than its warp-based counterpart to scan through the RRR sets. In other words, as the number of RRR sets increases, the thread-based approach outperforms the warp-based approach. For smaller numbers of RRR sets, warps benefit from memory coalescing, but serialization increases as the number of sets grows. This is because warps are limited by hardware constraints, as the number of sets grows, the fixed number of warps becomes a bottleneck. This results in the work being processed in more iterations, leading to greater serialization. In contrast, the larger number of threads allows workload to be distributed more evenly, increasing parallelization and improving scalability.

Let W_n and T_n denote the numbers of launchable warps and threads, respectively. Also, let C_w and C_t represent the cost (running time) to process a single set using a warp and a thread, respectively. It is clear that $W_n < T_n$ and $C_w < C_t$. While processing a single set is cheaper using a warp, as the number of sets increases, the warp-based approach must complete more iterations because it can only process W_n sets in parallel at a time. In contrast, the thread-based approach divides the workload into smaller chunks, increasing overall parallelization. As a result, the thread-based approach becomes more efficient as the number of sets grows. More precisely, assuming that each thread (or warp) takes the same cost as C_t (or C_w) to process a set in each iteration, if the number of RRR sets, denoted as N, is sufficiently large enough, we have

$$\left\lceil \frac{N}{W_n} \right\rceil C_w > \left\lceil \frac{N}{T_n} \right\rceil C_t.$$

Figure 3 shows experiment results on the scalability of both approaches as N increases and confirms that our thread-based approach achieves improved scalability.

4 Performance Evaluation

In this section, we present extensive experiment results to demonstrate performance of eIM compared to cuRipples and gIM in terms of memory efficiency and running time.

4.1 Setup

Datasets. We consider 16 real-world unweighted network datasets from SNAP [13], which are listed in an ascending order of graph size (number of vertices) in Table 1. Since the datasets are unweighted, we preprocessed them by assigning edge weights according to the given diffusion model, as explained in Section 2.1.

Table 1: Graph statistics

Dataset	# Vertices	# Edges
wiki-vote (WV)	8,298	103,689
p2p-gnutella32 (PG)	62,586	147,892
soc-Epinions1 (SE)	75,888	508,837
slashdot0902 (SD)	82,168	870,161
email-EuAll (EE)	265,214	418,956
web-Stanford (WS)	281,904	2,312,497
web-NotreDame (WN)	325,729	1,469,679
com-dblp (CD)	425,957	1,049,866
com-Amazon (CA)	548,552	925,872
web-Berkstan (WB)	685,231	7,600,595
web-Google (WG)	916,428	5,105,039
com-Youtube (CY)	1,157,828	2,987,624
soc-pokec-relationships (SPR)	1,632,804	30,622,564
wiki-topcats (WT)	1,791,489	28,508,141
com-Orkut (CO)	3,072,627	117,185,083
soc-LiveJournal1 (SL)	4,847,571	68,475,391

Software and hardware. We conduct the experiments on a Linux server equipped with a 16-core, 2.9 GHz Intel Xeon CPU, 96 GB RAM, and an NVIDIA RTX A6000 48 GB GPU. We implement eIM[1] in C++ and CUDA, and compile it via the nvcc compiler version 11.8 with "-O3" optimization enabled. For baselines, i.e., cuRipples [15] and gIM [21], we use the C++/CUDA code from their corresponding papers and run the code following their instructions. For cuRipples, tests were conducted on a single GPU and 16 CPU cores.

Parameter setting. We select the parameters k and ε for eIM and the baselines in our experiments as follows. By default, we set $k = 50$ and $\varepsilon = 0.05$, unless otherwise specified. We selected these parameters for comparability with the experiments done in [15, 21] for cuRipples and gIM. Additionally, we analyze the impact of varying k and ε on each algorithm's performance. We observed that quality of solutions, or the number of vertices activated by the selected seed set, provided by eIM remains the same as the one by cuRipples and gIM; thus, below we only present experiment results for memory efficiency and running time due to the space limit. All experiment results reported here are obtained by taking the average over ten runs.

4.2 Memory Reduction Evaluation

The network datasets are initially represented in CSC format. To evaluate the memory saving capabilities of log encoding, we compare the memory required to store a dataset in its original CSC format with the memory required when the CSC components are compressed using log encoding. Our results indicate that for smaller networks, up to 28.8% of memory can be saved. When the network size increases, the percentage of memory saved decreases but remains greater than 14%, indicating the benefit of log encoding.

To compare the memory reduction in storing and processing the RRR sets, we record the amount of memory used in storing and processing the RRR sets for various networks before and after the log encoding is applied to eIM under the IC model. The amount

[1] https://github.com/JKDNY/eIM

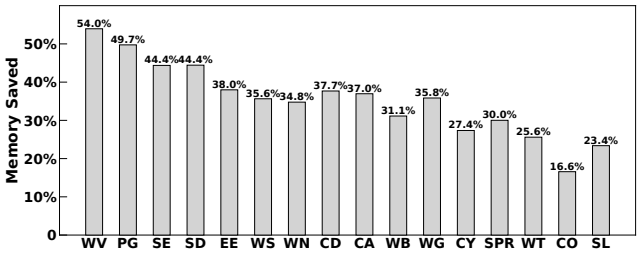

Figure 4: Memory saved by applying log encoding to store RRR sets and network data.

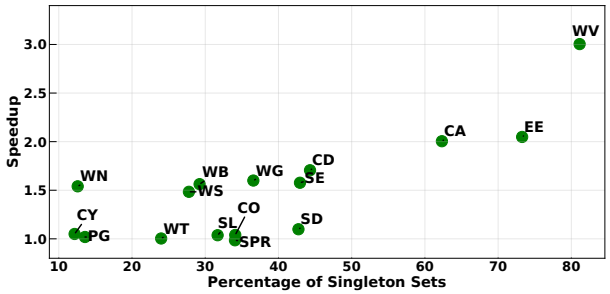

Figure 5: Speedup vs. percent of sets with only source vertices.

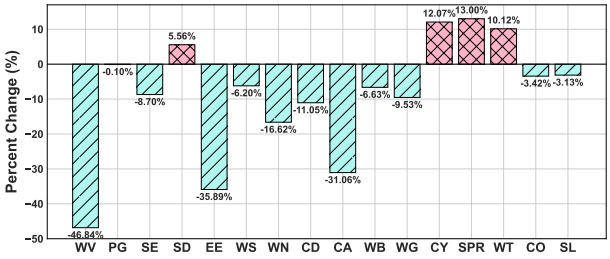

Figure 6: Percent change in memory for storing RRR sets when source vertices are removed.

of saved memory depends on both the size of the network and its underlying structure. Figure 4 shows the total memory saved for both the RRR sets and the network data. It shows that log encoding can reduce the memory required to store this data by up to 54% for small networks and up to 16.6% for larger networks.

4.3 Performance of Source Vertex Elimination

The removal of source vertices generally leads to greater speedups for networks that create a substantial number of RRR sets that only contain the source vertex, as shown in Figure 5. This improvement is achieved by eliminating these singleton sets, which allows the algorithm to generate more informative sets that contribute to meaningful coverage. As a result, the fraction of RRR sets covered by the seed set S increases more rapidly, reducing the total number of RRR sets that need to be generated.

The impact of eliminating source vertices on memory is shown in Figure 6. To measure this impact, we ran eIM under the IC model for each network ten times, both including and excluding the source vertices, and obtained the average size of $\mathbb{R}$, which is a collection of RRR sets, at the end of the algorithm execution. An average reduction of 8.65% to $\mathbb{R}$ is achieved across all networks when the

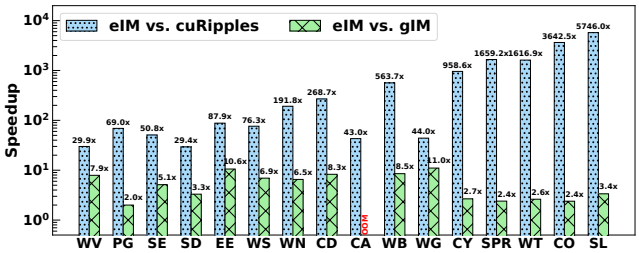

Figure 7: Speedups of eIM under IC with $k=50$ and $\varepsilon=0.05$.

Table 2: Speedup achieved by eIM over gIM under the IC model when increasing the values of k with $\varepsilon = 0.05$

Dataset	$k=20$	$k=40$	$k=60$	$k=80$	$k=100$
WV	8.04	7.82	9.02	10.68	19.23
PG	1.43	1.74	2.18	2.72	3.05
SE	3.28	4.92	6.52	8.0	7.32
SD	2.73	3.29	4.96	6.23	7.44
EE	7.19	14.32	15.11	19.45	23.02
WS	3.02	5.84	8.03	10.62	12.89
WN	2.79	5.28	4.66	5.42	6.08
CD	3.63	7.02	9.54	12.48	14.25
CA	OOM/0.18	OOM/0.21	OOM/0.27	OOM/0.35	OOM/0.32
WB	3.42	6.94	10.07	8.07	8.94
WG	5.68	8.83	OOM/0.37	OOM/0.46	OOM/0.61
CY	1.98	2.49	3.13	5.46	6.44
SPR	1.75	2.20	3.33	4.32	5.46
WT	1.56	2.39	3.56	4.62	5.83
CO	1.62	2.23	2.89	3.35	3.76
SL	2.19	4.07	3.90	OOM/1.42	OOM/1.83

Table 3: Speedup achieved by eIM over gIM under the IC model when decreasing ε values with k fixed at 100

Dataset	$\varepsilon=0.5$	$\varepsilon=0.45$	$\varepsilon=0.4$	$\varepsilon=0.35$	$\varepsilon=0.3$	$\varepsilon=0.25$	$\varepsilon=0.2$	$\varepsilon=0.15$	$\varepsilon=0.1$	$\varepsilon=0.05$
WV	1.93	2.16	2.07	2.39	2.63	2.91	3.67	5.07	10.69	19.23
PG	0.86	0.75	0.76	0.90	0.88	0.87	1.13	1.36	1.92	3.05
SE	1.25	1.18	1.21	1.37	1.54	1.56	1.78	2.68	4.26	7.32
SD	1.05	1.15	1.20	1.25	1.37	1.49	1.78	2.67	4.07	7.44
EE	1.74	1.95	2.11	2.24	2.56	2.94	3.68	6.46	10.73	23.02
WS	0.99	1.20	1.14	1.27	1.72	2.46	3.16	4.50	7.27	12.89
WN	1.07	1.03	1.04	1.14	1.17	1.21	1.42	1.94	3.04	6.08
CD	1.07	1.17	1.13	1.26	1.56	1.88	2.19	3.10	4.85	14.25
CA	4.16	4.84	5.57	6.45	7.94	10.02	13.08	17.95	24.44	OOM/0.32
WB	0.94	1.43	1.52	1.68	2.06	2.35	3.02	4.11	5.95	8.94
WG	2.23	2.48	2.81	3.16	3.92	4.71	5.71	7.20	9.12	OOM/0.61
CY	1.29	1.38	1.48	1.63	1.78	1.69	2.00	2.43	3.47	6.44
SPR	1.11	1.18	1.28	1.30	1.51	1.75	2.06	2.62	3.64	5.46
WT	1.23	1.24	1.31	1.38	1.50	1.75	2.13	2.76	3.92	5.83
CO	1.15	1.12	1.05	1.18	1.22	1.31	1.55	1.92	2.52	3.76
SL	1.62	1.77	1.93	2.18	2.43	2.37	2.80	3.32	4.09	OOM/1.83

source vertices are excluded. Networks that generate more than 50% sets that only contain source vertices achieve a greater reduction in the size of $\mathbb{R}$ compared to other networks. Since the removal of source vertices leads to quicker convergence in finding a lower bound for θ, it results in the generation of fewer but larger RRR sets. Due to these larger RRR sets, we observe that some networks may be impacted negatively and the size of $\mathbb{R}$ is slightly increased.

4.4 Performance under the IC Model

We now turn our attention to the performance comparison of eIM against cuRipples and gIM. We first consider the IC model. Figure 7 shows the speedups achieved by eIM over cuRipples and gIM under the IC model. eIM significantly outperforms both cuRipples and gIM on all datasets, and eIM achieves a speedup of up to 5746× compared to cuRipples, and up to 11× compared to gIM. In addition, as the network size increases, the speedup achieved by eIM over

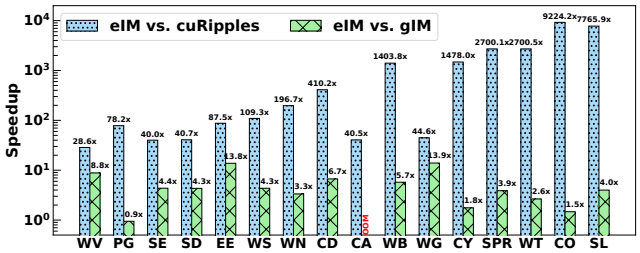

Figure 8: Speedups of eIM under LT with $k = 50$ and $\varepsilon = 0.05$.

cuRipples trends upwards. This performance advantage can be attributed to eIM maintaining RRR sets within GPU memory, whereas cuRipples incurs substantial overhead from repeated transfer of data from host to device. When compared to gIM, eIM achieves higher speedups when more RRR sets are required to be generated. These speedups are a result from using our thread-based approach when scanning through RRR sets. By log encoding (bit-packing), we also reduce the memory footprint and improve cache utilization. Furthermore, networks that tend to produce a significant number of sets containing only source vertices, such as Wiki-Votes and Email-Eu, exhibit the highest speedups, suggesting the performance benefits of their removal.

We also evaluate the impact of parameters k and ε on the performance. Here we focus on the comparison of eIM to gIM. Table 2 shows the speedup of eIM compared to gIM when increasing the values of k while having $\varepsilon = 0.05$. The amount of speedup generally increases as the value of k increases. Additionally, for some networks, eIM is able to handle larger values of k, in which case gIM returns an OOM error. In such cases, the table entries indicate OOM for gIM and the running time of eIM in seconds. The results indicate that eIM tends to scale more consistently than gIM. In addition, Table 3 shows the speedup of eIM over gIM when decreasing ε values with k fixed at 100. The results show that compared to gIM, eIM is more adapted to handle lower values of ε and avoids OOM errors. When scaling for a larger seed set (i.e., a higher value of k) or for higher accuracy (i.e., a smaller value of ε), eIM shows greater performance improvement for all datasets, although the amount of the performance improvement still depends on the network structure. Note that gIM runs slightly faster than eIM for a few cases in which the number of generated RRR sets is relatively small. In such case, the benefit of using shared memory in gIM outweighs its overhead of dynamically allocating additional memory space.

4.5 Performance under the LT Model

Next, we conduct the experiments to compare the performance of eIM to cuRipples and gIM under the LT model. As can be seen from Figure 8, eIM outperforms both cuRipples and gIM on all datasets, except for p2p-gnutella32. The speedup achieved by eIM is up to 9224× when compared to cuRipples and 13× when compared to gIM. Notably, gIM encountered an OOM error on com-Amazon, whereas eIM remained successful due to its more efficient memory usage when generating RRR sets. We also observe the same performance improvements and trends as in the IC model, including increasing speedups with the network size, and the advantages of our thread-based search and source vertex elimination.

Table 4: Speedup achieved by eIM over gIM under the LT model when increasing the values of k with $\varepsilon = 0.05$

Dataset	$k=20$	$k=40$	$k=60$	$k=80$	$k=100$
WV	6.90	9.27	10.03	7.94	9.62
PG	0.71	0.90	0.96	1.05	1.46
SE	2.42	5.09	7.00	5.28	6.20
SD	2.14	4.13	5.89	7.54	5.27
EE	9.24	19.62	20.21	25.66	30.39
WS	2.37	3.91	5.17	6.43	7.20
WN	2.60	3.08	4.05	4.82	5.66
CD	2.97	5.83	7.98	10.20	6.15
CA	OOM/0.18	OOM/0.21	OOM/0.26	OOM/0.33	OOM/0.41
WB	2.79	4.69	6.38	10.54	7.05
WG	4.66	10.44	10.03	11.99	17.11
CY	0.99	1.49	1.91	2.35	2.51
SPR	1.71	2.68	3.00	3.69	4.34
WT	1.77	2.25	2.97	3.59	4.07
CO	1.42	1.49	1.80	2.12	1.92
SL	2.19	3.54	6.00	4.81	4.95

Table 5: Speedup achieved by eIM over gIM under the LT model when decreasing ε values with k fixed at 100

Dataset	$\varepsilon=0.5$	$\varepsilon=0.45$	$\varepsilon=0.4$	$\varepsilon=0.35$	$\varepsilon=0.3$	$\varepsilon=0.25$	$\varepsilon=0.2$	$\varepsilon=0.15$	$\varepsilon=0.1$	$\varepsilon=0.05$
WV	1.44	1.88	1.93	1.52	1.68	2.02	2.21	3.22	6.34	9.62
PG	0.85	0.84	0.94	0.94	0.85	0.85	0.96	0.95	1.15	1.46
SE	1.03	1.04	1.04	1.11	1.19	1.16	1.52	2.07	3.24	6.20
SD	0.95	1.07	1.03	1.15	1.10	1.24	1.49	2.03	2.93	5.27
EE	1.76	1.98	2.14	2.60	2.80	3.45	5.40	8.02	14.24	30.39
WS	1.06	1.08	1.08	1.23	1.26	1.57	2.03	2.78	4.39	7.20
WN	1.14	1.14	1.10	1.24	1.10	1.20	1.46	2.17	3.13	5.66
CD	0.94	1.17	1.17	1.09	1.28	1.46	1.74	2.22	3.42	6.15
CA	5.30	6.02	6.77	8.52	9.17	11.77	15.17	20.59	27.73	OOM/0.41
WB	0.85	1.10	1.22	1.44	1.55	1.80	2.34	3.06	4.37	7.05
WG	2.97	3.26	3.76	4.34	5.08	6.35	7.88	10.21	13.45	17.11
CY	1.39	1.36	1.32	1.28	1.44	1.20	1.02	1.25	1.61	2.51
SPR	1.03	1.13	1.23	1.33	1.51	1.82	2.12	2.61	3.35	4.34
WT	0.94	0.98	1.02	1.13	1.21	1.39	1.69	2.10	2.82	4.07
CO	1.42	1.32	1.32	1.22	1.34	1.56	1.50	1.57	1.79	1.92
SL	1.47	1.64	1.76	1.91	2.20	2.45	2.85	3.46	4.21	4.95

We further report the speedups achieved by eIM over gIM under the LT model with different choices of k and ε. Table 4 presents the results when increasing the values of k with $\varepsilon = 0.05$, and Table 5 shows the results when decreasing ε values with k fixed at 100. We observe that eIM outperforms gIM for most cases, and the speedup is up to 30×. It is worth noting that similar to the results under the IC model, gIM is slightly faster than eIM for a few cases where the number of generated RRR sets is relatively small. Nonetheless, eIM greatly improves the scalability of finding the seed set on large networks, especially when the size of the seed set is greater and/or the accuracy is higher. All the above results indicate the high effectiveness and efficiency of eIM.

5 Conclusion

The scalability of the IMM algorithm is significantly limited due to its large memory overhead. To address this challenge, we introduced eIM, a GPU-accelerated IMM algorithm with several key innovations, including log encoding, efficient GPU memory usage, and optimized workload distribution. Through extensive experiments, we demonstrated that eIM reduces memory footprint and improves running time compared to gIM and cuRipples, while maintaining solution quality. Looking ahead, we plan to extend eIM to support multi-GPU execution to further improve scalability and to expand support for the IC model with random edge weights, which covers different influence propagation scenarios.

Acknowledgments

This work was supported by the National Science Foundation under Grant Nos. 2209921 and 2209922, and an equipment donation from NVIDIA Corporation.

References

[1] Noushin Azami and Martin Burtscher. 2022. Compressed in-memory graphs for accelerating GPU-based analytics. In *2022 IEEE/ACM Workshop on Irregular Applications: Architectures and Algorithms*. IEEE, 32–40.

[2] Maciej Besta, Dimitri Stanojevic, Tijana Zivic, Jagpreet Singh, Maurice Hoerold, and Torsten Hoefler. 2018. Log(graph): A near-optimal high-performance graph representation. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*. ACM, 1–13.

[3] Christian Borgs, Michael Brautbar, Jennifer Chayes, and Brendan Lucier. 2013. Maximizing social influence in nearly optimal time. In *Proceedings of the 25th Annual ACM-SIAM Symposium on Discrete Algorithms*. Society for Industrial and Applied Mathematics, 946–957.

[4] Xinyu Chen, Marco Minutoli, Jiannan Tian, Mahantesh Halappanavar, Ananth Kalyanaraman, and Dingwen Tao. 2022. HBMax: Optimizing memory efficiency for parallel influence maximization on multicore architectures. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*. ACM, 412–425.

[5] Pedro Domingos and Matt Richardson. 2001. Mining the network value of customers. In *Proceedings of the 7th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 57–66.

[6] Jacob Goldenberg, Barak Libai, and Eitan Muller. 2001. Talk of the network: A complex systems look at the underlying process of word-of-mouth. *Marketing Letters* 12 (2001), 211–223.

[7] Amit Goyal, Wei Lu, and Laks VS Lakshmanan. 2011. CELF++: Optimizing the greedy algorithm for influence maximization in social networks. In *Proceedings of the 20th International Conference Companion on World Wide Web*. ACM, 47–48.

[8] Mark Granovetter. 1978. Threshold models of collective behavior. *Amer. J. Sociology* 83, 6 (1978), 1420–1443.

[9] David Kempe, Jon Kleinberg, and Éva Tardos. 2005. Influential nodes in a diffusion model for social networks. In *Automata, Languages and Programming: 32nd International Colloquium, ICALP 2005, Lisbon, Portugal, July 11-15, 2005. Proceedings 32*. Springer-Verlag, 1127–1138.

[10] David Kempe, Jon Kleinberg, and Éva Tardos. 2003. Maximizing the spread of influence through a social network. In *Proceedings of the 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 137–146.

[11] Jure Leskovec, Lada A Adamic, and Bernardo A Huberman. 2007. The dynamics of viral marketing. *ACM Transactions on the Web* 1, 1 (2007), 5:1–5:39.

[12] Jure Leskovec, Andreas Krause, Carlos Guestrin, Christos Faloutsos, Jeanne VanBriesen, and Natalie Glance. 2007. Cost-effective outbreak detection in networks. In *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 420–429.

[13] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[14] Yuchen Li, Ju Fan, Yanhao Wang, and Kian-Lee Tan. 2018. Influence maximization on social graphs: A survey. *IEEE Transactions on Knowledge and Data Engineering* 30, 10 (2018), 1852–1872.

[15] Marco Minutoli, Maurizio Drocco, Mahantesh Halappanavar, Antonino Tumeo, and Ananth Kalyanaraman. 2020. cuRipples: Influence maximization on multi-GPU systems. In *Proceedings of the 34th ACM International Conference on Supercomputing*. ACM, 1–11.

[16] Marco Minutoli, Mahantesh Halappanavar, Ananth Kalyanaraman, Arun Sathanur, Ryan Mcclure, and Jason McDermott. 2019. Fast and scalable implementations of influence maximization algorithms. In *2019 IEEE International Conference on Cluster Computing*. IEEE, 1–12.

[17] Elchanan Mossel and Sebastien Roch. 2007. On the submodularity of influence in social networks. In *Proceedings of the 39th Annual ACM Symposium on Theory of Computing*. ACM, 128–134.

[18] Reece Neff, Mostafa Eghbali Zarch, Marco Minutoli, Mahantesh Halappanavar, Antonino Tumeo, Ananth Kalyanaraman, and Michela Becchi. 2024. FuseIM: Fusing probabilistic traversals for influence maximization on exascale systems. In *Proceedings of the 38th ACM International Conference on Supercomputing*. ACM, 38–49.

[19] Aida Rahmattalabi, Shahin Jabbari, Himabindu Lakkaraju, Phebe Vayanos, Max Izenberg, Ryan Brown, Eric Rice, and Milind Tambe. 2021. Fair influence maximization: A welfare optimization approach. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 35. 11630–11638.

[20] Matthew Richardson and Pedro Domingos. 2002. Mining knowledge-sharing sites for viral marketing. In *Proceedings of the eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 61–70.

[21] Soheil Shahrouz, Saber Salehkaleybar, and Matin Hashemi. 2021. gIM: GPU accelerated RIS-based influence maximization algorithm. *IEEE Transactions on Parallel and Distributed Systems* 32, 10 (Oct. 2021), 2386–2399.

[22] Youze Tang, Yanchen Shi, and Xiaokui Xiao. 2015. Influence maximization in near-linear time: A martingale approach. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 1539–1554.

[23] Youze Tang, Xiaokui Xiao, and Yanchen Shi. 2014. Influence maximization: Near-optimal time complexity meets practical efficiency. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. ACM, 75–86.

[24] Didier A Vega-Oliveros, Luciano da Fontoura Costa, and Francisco Aparecido Rodrigues. 2020. Influence maximization by rumor spreading on correlated networks through community identification. *Communications in Nonlinear Science and Numerical Simulation* 83 (2020), 105094.

[25] Chi Wang, Wei Chen, and Yajun Wang. 2012. Scalable influence maximization for independent cascade model in large-scale social networks. *Data Mining and Knowledge Discovery* 25 (2012), 545–576.

[26] Hanjiang Wu, Huan Xu, Joongun Park, Jesmin Jahan Tithi, Fabio Checconi, Jordi Wolfson-Pou, Fabrizio Petrini, and Tushar Krishna. 2024. Enhancing scalability and performance in influence maximization with optimized parallel processing. In *SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 667–676.

[27] Mao Ye, Xingjie Liu, and Wang-Chien Lee. 2012. Exploring social influence for recommendation: a generative model approach. In *Proceedings of the 35th International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM, 671–680.