

**ULTRA-FAST INFLUENCE MAXIMIZATION WITH FUSED
SAMPLING AND SKETCHES**

by
GÖKHAN GÖKTÜRK

Submitted to the Graduate School of Engineering and Natural Sciences
in partial fulfillment of
the requirements for the degree of
Doctor of Philosophy

Sabancı University
July 2021

**ULTRA-FAST INFLUENCE MAXIMIZATION WITH FUSED
SAMPLING AND SKETCHES**

Approved by:

[Redacted signature]

[Redacted signature]

[Redacted signature]

[Redacted signature]

[Redacted signature]

Date of Approval: July 14, 2021

GÖKHAN GÖKTÜRK 2021 ©

All Rights Reserved

ABSTRACT

ULTRA-FAST INFLUENCE MAXIMIZATION WITH FUSED SAMPLING AND SKETCHES

GÖKHAN GÖKTÜRK

COMPUTER SCIENCE AND ENGINEERING, Ph.D. DISSERTATION,
JULY 2021

Dissertation Supervisor: Asst. Prof. Kamer Kaya

Keywords: Influence maximization, Fused sampling, Parallel graph algorithms,
High-performance computing

Influence Maximization (IM) is the problem of finding a subset of vertices in a social network whose influence reaches the maximum reachability according to a diffusion model. Due to the NP-Hardness of the problem, often, greedy approximation algorithms are applied. However, irregular memory access patterns and the probabilistic nature of the problem make it a challenging yet rewarding optimization target.

This thesis proposes three high-performance IM methods and explores their performance considerations for implementation; first, we propose INFUSER-MG, an IM algorithm that uses a *hash-based, direction-oblivious* pseudo-random number generator and *fused sampling* to sample edges in undirected networks. Second, we propose HYPERFUSER for directed, generic networks; HYPERFUSER uses modified Flajolet-Martin sketches to estimate the cardinality of large reachability sets efficiently. Finally, we propose SUPERFUSER, a sketch-based IM algorithm that is specifically designed for the multi-GPU setting. SUPERFUSER uses a sampling-aware sample-space split mechanism to distribute the graph to multiple devices.

Also in this work, we discuss performance considerations at each step of the proposed algorithms and provide their high-performance implementations. For each algorithm, we provide detailed experimental results, including performance, quality, and scaling benchmarks.

ÖZET

ÖRNEKLEM BİRLEŞTİRME VE VERİ ÖZETLERİ İLE YÜKSEK PERFORMANSLI ETKİ ENİYİLEMESİ

GÖKHAN GÖKTÜRK

BİLGİSAYAR BİLİMİ VE MÜHENDİSLİĞİ, DOKTORA TEZİ,
TEMMUZ 2021

Tez Danışmanı: Dr. Öğr. Üyesi Kamer Kaya

Anahtar Kelimeler: Etki eniyilemesi, Örneklem birleştirme, Paralel çizge algoritmaları, Yüksek başarılı hesaplama

Etki Eniyileme (EE), bir sosyal ağda etkisinin bir yayılma modeline göre maksimum erişilebilirliğe ulaştığı bir düğüm alt kümesi bulma problemidir. Problemin NP-Zor olması nedeniyle, bu problem için genellikle açgözlü yaklaşım algoritmaları uygulanır. Düzensiz bellek erişim kalıpları ve sorunun olasılıksal doğası, onu zorlu ancak ödüllendirici bir optimizasyon hedefi haline getirmektedir.

Bu tez, üç yüksek performanslı EE yöntemi önermekte ve gerçekleştirme için performans değerlendirmelerini araştırmaktadır; ilk olarak, özüt tabanlı, yön bağımsız rasgele sayı üretici ve örneklem birleştirme kullanarak, yönlendirilmemiş bağları örnekleyen bir EE algoritması olan INFUSER-MG'yi öneriyoruz. İkinci olarak, yönlendirilmiş ağlar için, büyük erişilebilirlik kümelerinin boyutlarını tahmin etmek için değiştirilmiş Flajolet-Martin eskizleri kullanan bir EE yönetimi olan HYPERFUSER'ı anlatacağız. Son olarak, akıllı örneklem-uzay bölünmesi ile birden fazla Grafik İşleme Ünitesi kullanmak için özel olarak tasarlanmış, eskiz tabanlı bir EE algoritması olan SUPERFUSER önerilmiştir.

Bu tezde algoritmaların her adımının performans değerlendirmeleri de tartışılmakta ve önerilen algoritmaların yüksek performanslı gerçeklemleri de verilmektedir. Her algoritma için, literatürdeki en iyi yöntemler ile performans, kalite ve ölçekleme karşılaştırmaları da dahil olmak üzere ayrıntılı deney sonuçları bulunmaktadır.

TABLE OF CONTENTS

LIST OF TABLES	viii
LIST OF FIGURES	x
1. INTRODUCTION	1
2. NOTATION AND BACKGROUND	5
2.1. Notation	5
2.2. Influence Maximization	6
2.3. Related Work	12
2.4. The Importance of Memory Hierarchy.....	14
2.5. Amdahl’s Law	15
2.6. Memory Wall	16
2.7. Single Instruction-Multiple Data	17
2.8. Compute Unified Device Architecture (CUDA)	20
3. FUSED SAMPLING	23
3.1. Hash-Based Sampling.....	23
3.2. Fused Sampling	25
3.2.1. Hash-based Fused Sampling.....	25
3.2.2. Direction oblivious hash-based sampling	28
4. BOOSTING INFLUENCE-MAXIMIZATION KERNELS WITH FUSED SAMPLING	30
4.1. Vectorized Monte-Carlo graph traversal	31
4.1.1. A vectorized NEWGREEDY step	31
4.2. Finding marginal gains with memoization	36
4.3. Implementation Details	38
4.4. Experimental Results	38
4.4.1. Network datasets used in the experiments	38

4.4.2.	Metrics used to evaluate the performance.....	39
4.4.3.	Algorithms evaluated in the experiments	40
4.4.4.	Comparing INFUSER-MG with MIXGREEDY	41
4.4.5.	Comparing INFUSER-MG with State-of-the-Art	42
4.4.6.	Scalability with multi-threaded parallelism	46
4.4.6.1.	Performance with Wider AVX Registers	47
5.	FAST AND ERROR-ADAPTIVE INFLUENCE MAXIMIZATION BASED ON COUNT-DISTINCT SKETCHES	48
5.1.	Count-Distinct Sketches	49
5.2.	Estimating Reachability Set Cardinality.....	50
5.3.	Error-adaptive sketch rebuilding.....	54
5.4.	Implementation Details	56
5.5.	Experimental Results	56
5.5.1.	Experiment Settings.....	57
5.5.2.	Performance Metrics	58
5.5.3.	Algorithms evaluated in the experiments	58
5.5.4.	Comparing HYPERFUSER with State-of-art	59
5.5.5.	Scalability with multi-threaded parallelism	64
6.	A MULTI-GPU APPROACH TO FUSED-SAMPLING AND INFLUENCE MAXIMIZATION	65
6.1.	Count-distinct Sketches for GPUs	65
6.2.	Influence Cascade on GPU	66
6.3.	Fused-Sampling Aware Sample-Space Split	69
6.4.	The SUPERFUSER algorithm	71
6.5.	Experimental Results	74
6.5.1.	Algorithms evaluated in the experiments	76
6.5.2.	Comparing SUPERFUSER with GIM	77
6.5.3.	Comparing Multi-GPU SUPERFUSER with GIM.....	78
7.	CONCLUSION AND FUTURE WORK.....	80
	BIBLIOGRAPHY.....	82

LIST OF TABLES

Table 2.1. Table of notations	6
Table 2.2. AVX2 intrinsics used in the implementation.	19
Table 2.3. GPU Memory Hierarchy	21
Table 4.1. Properties of networks used in the experiments	39
Table 4.2. Execution times (in secs), memory use (in GBs), and influence scores (higher is better) of the algorithms on the networks with $K = 50$ seeds and constant edge weights with $p = 0.01$	41
Table 4.3. Execution times (in secs) of the algorithms with $K = 50$ seeds and $\tau = 16$ threads in different simulation settings.	43
Table 4.4. Memory use (in GBs) of the algorithms on the networks with $K = 50$ seeds in different simulation settings.....	44
Table 4.5. Influence scores of the algorithms on the networks with $K = 50$ seeds evaluated on $\mathcal{R} = 20000$ samples in different simulation settings (higher is better).	45
Table 4.6. Execution times (in secs) on the AVX512-capable architecture, networks with $K = 50$ seeds, using $\mathcal{R} = 2048$ for INFUSER-MG and constant edge weights with $p \in \{0.01, 0.1\}$	47
Table 5.1. Properties of networks used in the experiments	57
Table 5.2. HYPERFUSER execution times (in secs), influence scores, and memory use (in GBs) on the networks with $K = 50$ seeds using $\tau = 18$ threads and constant edge weights $w = 0.005$. Influence scores are given relative to HYPERFUSER. The runs that did not finish due to high memory use shown as "-". The values in the last 4 rows are normalized w.r.t. to those of HYPERFUSER.	60

Table 5.3. HYPERFUSER execution times (in secs), influence scores, and memory use (in GBs) on the networks with $K = 50$ seeds using $\tau = 18$ threads and constant edge weights $w = 0.01$. Influence scores are given relative to HYPERFUSER. The runs that did not finish due to high memory use shown as "-". The values in the last 4 rows are normalized w.r.t. to those of HYPERFUSER.	60
Table 5.4. HYPERFUSER execution times (in secs), influence scores, and memory use (in GBs) on the networks with $K = 50$ seeds using $\tau = 18$ threads and constant edge weights $w = 0.1$. Influence scores are given relative to HYPERFUSER. The runs that did not finish due to high memory use shown as "-". The values in the last 4 rows are normalized w.r.t. to those of HYPERFUSER.	61
Table 6.1. Average size and maximum achievable speed-up using fused-sample aware split method. Selected datasets are split into two and four batches using sampling probability $p = 0.01$	70
Table 6.2. Properties of networks used in the experiments	75
Table 6.3. Single GPU SUPERFUSER performance compared to state-of-art. The values in the last three rows are normalized w.r.t. to those of SUPERFUSER ($\mathcal{J} = 64, \text{FLOAT}$). The Friendster dataset is excluded due to missing table values (i.e., too large to handle via a single GPU).	77
Table 6.4. Multi-GPU SUPERFUSER execution time compared to GIM. The values in the last 3 rows are normalized w.r.t. to those of SUPERFUSER ($\mathcal{J} = 64, \text{FLOAT}, 2 \text{ GPUs}$). The missing values are shown with "-".	79

LIST OF FIGURES

Figure 2.1. (a) The directed graph $G = (V, E)$ for IC with independent diffusion probabilities. (b) The directed graph for WC is obtained by setting the diffusion probabilities of incoming edges to $1/ \Gamma_G^-(v) $ for each vertex $v \in V$.	7
Figure 2.2. Difference between serial addition(a) and SIMD addition(b) ..	18
Figure 3.1. (a) Two sampled subgraphs of a toy graph with 4 vertices and 6 edges. (b) The simulations performed are fused with sampling. Each edge is labeled with the corresponding sample/simulation IDs.	26
Figure 3.2. Cumulative probability function of hash-based sampling probabilities on various real-life networks.	27
Figure 3.3. Bias distribution of hash-based sampling probabilities on various real-life networks.	27
Figure 4.1. (a) Two sampled subgraphs of the toy graph from Figure 2.1a with 5 vertices and 3 and 5 edges, respectively. (b) The simulations are performed in a way to be fused with sampling. Each edge is labeled with the corresponding sample/simulation IDs.	32
Figure 4.2. (a) The initial state on a toy graph for connected-component labeling; all vertices are labeled with their ids. (b) First, the edges of A are processed; the edge to C is in both samples. C 's labels are updated. (c) B 's edges are processed. The edge to C exists in the second sample. C 's second label is smaller, hence no update is performed. (d) C 's edges are being processed. It has edges to A , B , D , and E in the samples. The labels $\langle A, A \rangle$ are propagated to D and E since the edges are in both samples. B 's second label is updated because only sample 2 contains the corresponding (C, B) edge. (e)(f) D and E edges in the samples. However, there are no updates.	33
Figure 4.3. Speedup obtained by INFUSER-MG($\mathcal{R} = 2048, \tau = 16$) over IMM($\epsilon = 0.13, \tau = 16$).	44

Figure 4.4. INFUSER-MG speedup with multiple threads.	46
Figure 5.1. (a) The initial state on the toy graph for HYPERFUSER; all vertices are set as live (green), and their registers are initialized with the length of the zero prefix of their hashes. (b) For the (1,4) edge which is live for both simulations, 1's registers are set to the maximum of both 1's and 4's registers. The (4,3) edge is live only in the second simulation. Hence, the second register of 4 is updated to 5. For the second iteration, vertices 1 and 4 are live (green) since their registers have changed. (c) For the live (1,4) edge, 1's second is updated and 1 is set as live again. (d) All the registers converged. As no live vertices exist, the process stops.	52
Figure 5.2. Effect of register saturation on Amazon dataset($w = 0.01$) using HYPERFUSER ($\mathcal{J} = 256$) without rebuilding against Greedy($\mathcal{R} = 20000$) method(Kempe, Kleinberg, & Tardos, 2003).	54
Figure 5.3. Effect of ϵ parameters on HYPERFUSER ($\mathcal{J} = 256, \epsilon_c = 0.02$) performance, using $\tau = 18$ threads. Lighter shades are better.	55
Figure 5.4. Speedups obtained by HYPERFUSER ($\mathcal{J} = 256$) over IMM ($\epsilon=0.5$) using $\tau = 18$ threads.	63
Figure 5.5. Speedups obtained by HYPERFUSER ($\mathcal{J} = 256$) over SKIM ($r=64, l=64$) using $\tau = 18$ threads.	63
Figure 5.6. Scaling of HYPERFUSER with multiple threads on some of largest datasets in the benchmarks.	64
Figure 6.1. An example run of a seed selection in SUPERFUSER; The example scheme uses 8 GPUs and $\mathcal{J} = 256$ registers. $M[j]$ represents a sketch with j registers. Timeline goes from top to bottom.	72
Figure 6.2. The reduction scheme of SUPERFUSER to find a seed vertex using sketch registers. The scheme uses 8 GPUs and $\mathcal{J} = 256$ registers. $M[j]$ represents a sketch with j registers. The symbol + represents the element-wise sum of registers. Yellow boxes show active devices. The timeline goes from left to right.	73

1. INTRODUCTION

When we are implementing algorithms, we often focus on correctness first, but we can hit a performance wall as the problem gets bigger. Sometimes we realize that our software is too slow; the computation might be finished too late that results are now irrelevant, or computation time/cost is too high to spend for the results. Even though the constants are removed in asymptotic analysis, and all algorithms with different constants are considered equivalent, it is hardly the case for real-life computation. Luckily, by adapting the code to how computer hardware works, several methods can speed it faster. Improving locality is one of those methods since we observe that most performance bottlenecks are caused by memory latency/bandwidth. A simple logical operation waiting for memory access can take $100\times$ more time due to the naive implementation. Details of the locality and memory hierarchy will be mentioned in Section 2.4.

The algorithms are often designed for simplicity, then special cases and heuristics are added for performance. We are interested in improving the performance of graph algorithms due to their irregular memory access patterns. And, we selected *probabilistic* graph algorithms for their wasteful nature. In these algorithms, temporal and spatial locality are often disregarded; the same edge is traversed multiple times, irregularly, even for throwing away results.

In this work, we show that the memory accesses can be accumulated to improve the performance by reducing the total amount of work and improving locality. Even though *Influence Maximization* is selected as the case, the methods proposed in this thesis should apply to many other probabilistic graph algorithms.

With their rapid growth, the study of efficient information/influence dissemination in networks has become an important and fruitful research area. It has applications in many fields including viral marketing (Leskovec, Adamic, & Huberman, 2007; Trusov, Bucklin, & Pauwels, 2009), social media analysis (Zeng et al., 2010; Moreno, Nekovee, & Pacheco, 2004), and recommendation systems (Lü et al., 2012). Hence, novel approaches to find good vertex sets which can effectively spread the

information are vital in practice. As the study of these networks is imperative for educational, political, economic, and social purposes, a high-quality seed set to initiate the diffusion may have vital importance. Furthermore, since the diffusion analysis may be time-critical, or increasing the influence coverage may be too expensive, novel and efficient approaches to find good vertex sets that propagate the information effectively are essential.

The Influence Maximization (IM) problem is introduced by Kempe, Kleinberg, and Tardos (2003). Formally, it focuses on finding the most promising seed (vertex) set with a given cardinality that increases the expected number of influenced vertices. IM is proven to be NP-hard (Kempe, Kleinberg, & Tardos, 2003) and there exist various simplifications and heuristics proposed in the literature (Chen, Wang, & Yang, 2009; Narayanam & Narahari, 2010; Kimura, Saito, & Nakano, 2007; Chen, Wang, & Wang, 2010; Chen, Yuan, & Zhang, 2010; Kim, Kim, & Yu, 2013a; Goyal, Lu, & Lakshmanan, 2011b; Jung, Heo, & Chen, 2012; Cheng et al., 2014; Liu et al., 2014; Galhotra, Arora, & Roy, 2016). It has also been shown that a greedy Monte-Carlo approach provides a constant approximation for the optimal solution (Kempe, Kleinberg, & Tardos, 2003).

For a graph with n vertices, the expected complexity of this greedy algorithm, estimating an influence score σ , running R simulations, and selecting K seed vertices is $\mathcal{O}(KRn\sigma)$. Hence, for real-life networks with hundreds of thousands of vertices, the approach is expensive. However, these simulation-based, greedy algorithms provide the best possible approximation guarantees. Therefore they are considered as the gold standard for IM.

Performing the simulations of a greedy algorithm in parallel is an immediate and straightforward remedy to reduce the execution time of IM kernels and make them scalable for large-scale networks. However, restructuring the kernels to leverage instruction-level parallelism has not been investigated before. Although modern compilers can efficiently and automatically utilize instruction-level parallelism for applications with regular memory access patterns, it is not a straightforward task for graph processing kernels due to their irregular memory accesses. Furthermore, vectorization attempts on such kernels usually fail to provide significant performance improvements.

Simulating a greedy algorithm in parallel is a straightforward workaround to reduce the execution time of IM kernels and make them scalable for large-scale networks. However, for large networks, a parallel, greedy approach with a good approximation guarantee does not come cheap on networks with billions of vertices and edges, even if many processing units/cores are available. In this work;

- We propose several ultra-fast high-quality Influence Maximization methods. Unlike the traditional greedy approach, the proposed approach samples the edges as traversed in multiple simulations. Hence, for a single simulation, sampling and diffusion processes are *fused*. All methods mentioned are made public ¹,
- By running concurrent simulations at once, we reduce the amount of connectivity information read from memory. When the simulations traverse the same edge within close time intervals, the edge is accessed only once as long as the simulation timelines allow methods to do so. Thus, the proposed approaches *reduce the pressure on the memory sub-system*. Furthermore, we utilize hardware parallelism with high efficiency, i.e., we usually leverage the full SIMD register or GPU warp and make use of all the arithmetic operations performed.
- In addition to instruction-level parallelism, we parallelize the methods on multi-core architectures and GPUs.
- All the algorithms designed in this thesis have high-performance implementations; our implementations are often orders of magnitude faster than their state-of-art counterparts.
- To better position the performance of our methods in the IM literature, we compare the performance, memory usage, and influence score with state-of-the-art approximation algorithms for Influence Maximization.

The dissertation is organized as follows: In Chapter 2, we present the background information and introduce the mathematical notation including Influence Maximization, Fused Sampling, and Count-Distinct Sketches. In addition, we provide preliminary information on Performance, Memory Hierarchies, and instruction sets and accelerators such as AVX2, AVX512, and GPUs.

Chapter 3 describes hash-based fused sampling, a fundamental technique we apply in our solutions, and its direction-oblivious variant.

Chapter 4 describes the connected-component-based Influence Maximization approach (Göktürk & Kaya, 2021), including details on the AVX2 implementation and its multi-core parallelization. Also, we added experimental results to compare them to the state-of-art methods (excluding sketches and heuristics).

In Chapter 5, we propose a new sketch-based Influence Maximization method (Gokturk & Kaya, 2021), HyperFuser. Similar to the previous chapter, details including AVX2/AVX-512 implementation, multi-core parallelization, and experimental results against the state-of-art methods will be given.

¹<https://github.com/ggokturk/infuser>

In Chapter 6, a blazing-fast, multi-GPU Influence Maximization algorithm is proposed. Furthermore, a better count-distinct sketch specifically designed for GPUs and a fused-sampling aware sample-space split method are described. The experimental results and their discussions are added to this chapter as well.

Finally, in Chapter 7, we discuss the contributions, present a final comparative overview, and discuss the future work.

2. NOTATION AND BACKGROUND

2.1 Notation

Let $G = (V, E)$ be a graph where the n vertices in V represent the agents, and m edges in E represent the relations among them. For *directed* graphs, an edge $(u, v) \in E$ is an *incoming* edge for v and an *outgoing* edge of u . The *incoming* neighborhood of a vertex $v \in V$ is denoted as $\Gamma_G^-(v) = \{u : (u, v) \in E\}$. Similarly, the *outgoing* neighborhood of a vertex $v \in V$ is denoted as $\Gamma_G^+(v) = \{u : (v, u) \in E\}$. When the graph is *undirected*, $\Gamma_G(v) = \{u : \{u, v\} \in E\}$ denotes the neighborhood of v .

A graph $G' = (V', E')$ is a sub-graph of G if $V' \subseteq V$ and $E' \subseteq E$. The diffusion probability on the edge $(u, v) \in G$ is noted as $w_{u,v}$, where $w_{u,v}$ can be determined either by the diffusion model or according to the strength of u and v 's relationship. In practice, $w_{u,v}$ can be determined by the strength of u and v 's relationship (Kempe, Kleinberg, & Tardos, 2003). For undirected graphs, $w_{u,v} = w_{v,u}$, and for simplicity, self edges (interactions), i.e., (u, u) or $\{u, u\}$, are not allowed.

Two vertices $u, v \in V$ are *connected* if there exists a *simple* $u \rightsquigarrow v$ path

$$\{u = v_0, v_1\}, \{v_1, v_2\}, \dots, \{v_{\ell-1}, v_\ell = v\}$$

of length ℓ in G , where all v_i s are unique. For directed graphs, the edges are oriented and the $u \rightsquigarrow v$ path is

$$(u, v_1), (v_1, v_2), \dots, (v_{\ell-1}, v).$$

An undirected graph $G' = (V', E')$ is a subgraph of G if $V' \subseteq V$ and $E' \subseteq E$. If all the vertices in G' are connected G' is called as a *connected subgraph* of G . If the subgraph

is a maximally connected subgraph of G it is called a *connected component* (CC) of G . For directed graphs, if \vec{G}' is a subgraph of \vec{G} and all the vertices in \vec{G}' are connected it is called as a *strongly connected subgraph* of \vec{G} . If the subgraph is maximal and strongly connected, it is called a *strongly connected component* of \vec{G} .

Table 2.1 Table of notations

Variable	Definition
$G = (V, E)$	Graph G with vertices V and edges E
$\Gamma_G(v)$	Neighborhood of vertex v in graph G
$\Gamma_G^+(v)$	<i>outgoing</i> neighborhood of a vertex (directed graphs)
$\Gamma_G^-(v)$	<i>incoming</i> neighborhood of a vertex (directed graphs)
$w_{u,v}$	Probability of u directly influencing v
$R_G(v)$	Reachability set of vertex v on graph G
S	Seed set to maximize influence
K	Size of the seed set
\mathcal{R}	Number of Monte-Carlo simulations performed
$\sigma_G(S)$	Influence score of S in G , i.e., expected number of vertices reached from S in G
$\sigma_G(S, v)$	Marginal influence gain by adding vertex v to seed set S
$h(u, v)$	Hash function for edge $\{u, v\}$
$h_j(x)$	j 'th hash function for number x
h_{max}	Maximum value hash function h can return
B	Batch size, number of simultaneous simulations ran.
$[a, \dots, a]_B$	Vector of size B , contains all a
$A(v)$	Out-going edges coming from vertex v
\mathcal{J}	Number of registers for sketches
$clz(..)$	Count leading zeros function
$H(..)$	Harmonic mean function
$\lll B, T \ggg$	CUDA kernel launch parameters; B blocks, T threads per block

2.2 Influence Maximization

Influence maximization aims to find a seed set $S \subseteq V$ among all possible size K subsets of V that maximizes an *influence spread function* $\sigma_{G,M}$ on G under a diffusion model M when the diffusion process is initiated from S . Among the algorithms proposed in this thesis, While INFUSER-MG focuses on graphs with undirected edges, HYPERFUSER and SUPERFUSER focus on graphs with directed edges which is the generic case for graph computations. The influence spread function $\sigma_{G,M}(\cdot)$

computes the *expected* number of agents (or nodes, vertices, etc.) influenced (activated) through a diffusion model M . For the sake of simplicity, we drop M from the notation; in the rest of the text, σ_G refers to $\sigma_{G,M}$. Some of the popular diffusion models for IM in the literature are *independent* and *weighted cascade* (IC and WC), and *linear threshold* (LT) (Kempe, Kleinberg, & Tardos, 2003).

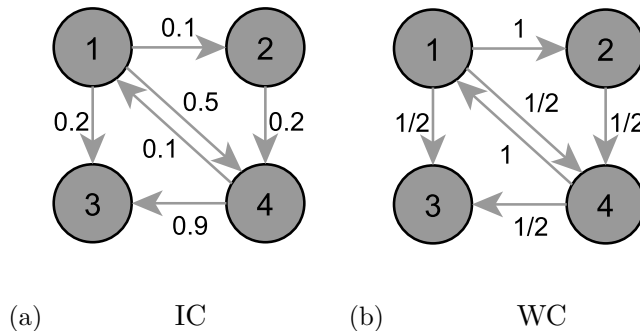


Figure 2.1 (a) The directed graph $G = (V, E)$ for IC with independent diffusion probabilities. (b) The directed graph for WC is obtained by setting the diffusion probabilities of incoming edges to $1/|\Gamma_G^-(v)|$ for each vertex $v \in V$.

- The **Independent Cascade** model works in rounds and activates a vertex v in the current round if one of v 's incoming edges (u, v) is used during the diffusion round, which happens with the activation probability $w_{u,v}$, given that u has already been influenced in the previous rounds. The activation probabilities are independent (from each other and previous activations) in the *independent cascade* model, which we focus on in this thesis. A toy graph with activation probabilities on the edges is shown in Figure 2.1a. In theory, there can exist parallel and independent $\{u, v\}$ edges in E . In practice, they are merged to a single $\{u, v\}$ edge with compound probability via preprocessing. For example, the probability of vertex u activating vertex v through two parallel edges with $w = 0.1$ is $w_{u,w} = 1 - (1 - 0.1)^2 = 0.19$.
- The **Weighted Cascade** model is a variant of IC and uses the structural properties of vertices to set the edge weights as shown in Figure 2.1b. The method, as described in (Kempe, Kleinberg, & Tardos, 2003), sets $w_{u,v} = 1/d_v$ where d_v is the number of incoming edges of v (which in the original graph is equal to $|\Gamma_G^-(v)|$). Therefore, if v has ℓ neighbors activated in the last round, its probability of activation in the new round is $1 - (1 - 1/d_v)^\ell$.
- **Linear threshold** generalizes the IC model and activates the vertex v once the cumulative activation coming from its neighbors exceeds a given threshold θ_v . All the (u, v) edges with active u vertices are taken into account in the process. Vertex v is activated when the total activation probability through

these edges exceeds θ_v (Kempe, Kleinberg, & Tardos, 2003). Each edge $e = (s, v) \in E$ has a weight $w_{s,v}$. Each vertex v has incoming edge list $Adj^{-1}(v)$ and $\sum_{s \in Adj^{-1}(v)} w_{s,v} \leq 1$. Also, each vertex v has associated with a threshold θ_v . Initially, θ is sampled uniformly, and seed set is set as activated. Then, until convergence is reached simulation ran to transfer w_{ij} values from activated vertices to their neighborhoods, activating each vertex v that reach θ_v value. Weights can be uniform i.e 0.1, 0.01, or 0.001 for all edges, as we observed in many work.

- **Triggering** proposed by Kempe, Kleinberg, and Tardos (2003) to generalize of previous models mentioned above. In the triggering model, each neighbor has a probability of influencing vertex v . The diffusion process chooses a random subset of vertices called "triggering set" to activate vertices at each instance.

The IM problem is NP-hard under the cascade and linear threshold models. This being said, the influence function is non-negative, monotone and sub-modular, which means that adding a single vertex to the current seed set can only increase the overall influence and decreases the marginal influence scores for the remaining vertices that are not in the set. Due to these properties, the influence score of a greedy solution, which always adds the most promising vertex with the highest marginal gain to a seed set of final size K is at least $1 - (1 - 1/K)^K \geq 63\%$ of the optimal solution (Nemhauser, Wolsey, & Fisher, 1978).

There exist simulation-based (Kempe, Kleinberg, & Tardos, 2003; Chen, Wang, & Yang, 2009), sketch-based (Cohen et al., 2014), and proxy-based (Chen, Wang, & Yang, 2009; Jung, Heo, & Chen, 2012) approaches in the literature to find a seed set S that maximizes the influence spread in a graph. Simulation-based approaches run Monte-Carlo simulations, whereas sketch-based ones utilize approximate data structures. On the other hand, proxy-based approaches simplify the IM problem and utilize simpler heuristics.

For the simulation-based approaches, the complexity analysis stays consistent for many diffusion models, including *Independent Cascade*, *Weighted Cascade*, and *Linear Threshold*. The time complexity of the greedy algorithm, estimating the σ influence score, running R simulations, and selecting K seed vertices is $\mathcal{O}(KRn\sigma)$ for a graph with n vertices. We concentrate on the IC model in this thesis, however, although their adaptation requires some work, the proposed methods are also relevant to other models in the literature.

Kempe et al. (2003) proposed the greedy Monte-Carlo-based algorithm using the approach as mentioned earlier and set the foundations. At each step, the greedy algorithm finds the vertex that increases the influence the most. As Feige's optimal

inapproximability result shows (Feige, 1998), the guaranteed approximation ratio is the *gold standard* for problems with a non-trivial size, both asymptotically and practically. On the contrary, the other, sketch- and proxy-based approaches do not guarantee this approximation ratio. This is why, in this thesis, we first target the greedy, simulation-based algorithms that use the proposed techniques to boost their performance. To the best of our knowledge, due to their complexity, these algorithms have been experimented only on small-scale graphs in the literature.

Since IM is an expensive problem, studies in the literature focus on improving algorithmic complexity. Instead of trying all the vertices at each step, the Cost-effective Lazy Forward (CELF) algorithm of Leskovec et al. (Leskovec, Adamic, & Huberman, 2007) keeps the vertices in a priority queue w.r.t. their marginal influence gains. Due to the submodularity property of the influence spread function, these values set upper bounds for the current marginal gains. When a vertex is visited, its current marginal gain is updated, i.e., its exact value is computed, and the vertex is replaced further down in the queue. When a vertex is seen twice, the remaining vertices are guaranteed to have smaller marginal gains. Hence, the greedy decision can be immediately taken. The bottleneck of CELF is in its initialization; the (marginal) influence scores for all the vertices must be computed, which is the most time-consuming part and makes the approach expensive for large-scale graphs. This approach is improved by Goyal, Lu, and Lakshmanan (2011a) by further exploiting the submodularity of the influence spread function.

Chen, Wang, and Yang (2009) improve CELF with MIXGREEDY. Instead of running Monte-Carlo simulations from each vertex to find the initial marginal gains, MIXGREEDY uses one iteration of another IM algorithm NEWGREEDY whose pseudocode is given in Algorithm 1. In NEWGREEDY, the algorithm greedily chooses K vertices to form the seed set S and uses \mathcal{R} graph samples to choose each seed vertex. For each inner-iteration (lines 6–12), the algorithm samples a subgraph G' from G . The pseudocode of the sampling algorithm, SAMPLE, is given in Algorithm 2 where each edge $\{u, v\}$ is included with probability $w_{u,v}$. Then the marginal gains of G' 's vertices for this sample are computed by using the reachability sets. For undirected graphs, computing $|R_{G'}(S \cup v)|$ for each vertex v takes $\mathcal{O}(m+n)$ time since a number of graph traversals which touch each edge only once is sufficient.

The pseudocode of MIXGREEDY is given in Algorithm 3. Note that MIXGREEDY uses only a single iteration of NEWGREEDY with parameters $(G, 1, \mathcal{R})$. Even though NEWGREEDY can be used to find each of the K vertices in S one by one, Chen et al.'s experiments revealed that NEWGREEDY is only faster in the initialization stage. The experiments show that performing the CELF approach and adding a vertex to the seed set in case of a revisit in the queue is faster for consequent vertices.

Algorithm 1 NEWGREEDY(G, K, \mathcal{R})

Input: $G = (V, E)$: the influence graph K : number of seed vertices \mathcal{R} : number of MC simulations per seed vertex**Output:** S : a seed set that maximizes influence on G mg : marginal influence scores

```
1:  $S \leftarrow \emptyset$ 
2: for  $k = 1 \dots K$  do
3:   for  $v \in V$  do
4:      $mg_v \leftarrow 0$ 
5:   for  $r = 1 \dots \mathcal{R}$  do
6:      $G' = (V, E') \leftarrow \text{SAMPLE}(G)$ 
7:     Compute  $R_{G'}(S)$ 
8:     Compute  $|R_{G'}(\{v\})|$  for all  $v \in V$ 
9:     for  $v \in V \setminus S$  do
10:      if  $v \notin R_{G'}(S)$  then
11:         $\sigma_{G'}(S, v) \leftarrow |R_{G'}(v)|$ 
12:         $mg_v \leftarrow mg_v + \sigma_{G'}(S, v)$ 
13:    $mg_v \leftarrow \frac{mg_v}{\mathcal{R}}$  for all  $v \in V \setminus S$ 
14:    $S \leftarrow S \cup \{\text{argmax}_{v \in V} \{mg_v\}\}$ 
15: return  $S, mg$ 
```

Algorithm 2 SAMPLE(G)

Input: $G = (V, E)$: the original graph**Output:** $G' = (V, E')$: a subgraph of G

```
1:  $E' \leftarrow \emptyset$ 
2: for each  $\{u, v\}$  in  $E$  do
3:   Randomly choose  $r \in_R [0, 1]$  from a uniform dist.
4:   if  $r \leq w_{u,v}$  then
5:      $E' \leftarrow E' \cup \{u, v\}$ 
6: Construct  $G' = (V, E')$ 
7: return  $G'$ 
```

The subgraph $G' = (V', E')$ in the algorithm is sampled from G by using SAMPLE. where $\forall e \in E, P(e \in E') = p$. At first, all vertices in the seed set are marked to be activated. Then at each step, out-going edges from the activated set are marked to be activated as well. The algorithm is terminated when there are no vertices that can be activated. The number of activated vertices is $\sigma_G(S)$, influence spread of seed set S .

In this thesis, we will first focus on INFUSER-MG (Göktürk & Kaya, 2021), the fused and restructured form of MIXGREEDY. The memory accesses and floating-point operations performed by the existing algorithm are restructured to reduce the memory pressure for the marginal gain computations. This enables fused sampling

Algorithm 3 MIXGREEDY(G, K, \mathcal{R})

Input: $G = (V, E)$: the influence graph K : number of seed vertices \mathcal{R} : number of MC simulations per seed vertex**Output:** S : a seed set that maximizes influence on G

```
1:  $S, mg \leftarrow \text{NEWGREEDY}(G, 1, \mathcal{R})$ 
2:  $\sigma_G(S) \leftarrow \max_{v \in V} \{mg_v\}$   $\triangleright mg_v = \sigma_G(\emptyset, v)$ 
3:  $Q \leftarrow \text{PriorityQueue}()$ 
4: for  $v \in V \setminus S$  do
5:    $Q.\text{enqueue}(v, \text{priority}=mg_v)$ 
6:  $iter_v \leftarrow 0, \forall v \in V$ 
7: while  $|S| < K$  do
8:    $u \leftarrow Q.\text{top}()$ 
9:   if  $iter_u = |S|$  then
10:     $S \leftarrow S \cup \{u\}$ 
11:     $Q.\text{dequeue}(u)$ 
12:     $\sigma_G(S) \leftarrow \sigma_G(S) + mg_u$ 
13:   else
14:     $mg_u \leftarrow \text{RANDCAS}(G, S \cup \{u\}, \mathcal{R}) - \sigma_G(S)$ 
15:     $iter_u \leftarrow |S|$ 
16:     $Q.\text{updatePriority}(u, \text{priority}=mg_u)$ 
17: return  $S$ 
```

Algorithm 4 RANDCAS(G, S, R)

Input: $G = (V, E)$: the influence graph S : the seed set**Output:** $\sigma_G(S)$: influence score of seed set S on G

```
1:  $\sigma_S \leftarrow 0$ 
2: for  $r = 1 \dots \mathcal{R}$  do
3:    $G' = (V, E') \leftarrow \text{SAMPLE}(G)$ 
4:   Compute  $R_{G'}(S)$ 
5:    $\sigma_G(S) \leftarrow \sigma_G(S) + \frac{|R_{G'}(S)|}{\mathcal{R}}$ 
6: return  $\sigma_G(S)$ 
```

and vectorization. Furthermore, memoization is applied to reduce the cost of the CELF phase. The proposed techniques in this paper can be adopted by other probabilistic graph algorithms and other IM kernels to boost their performance. Although they are not focusing on probabilistic algorithms and fusing, SIMD-based alterations of graph kernels to regularize memory accesses have been studied before, e.g., to perform BFS, compute centrality metrics, or connected components (Sariyüce et al., 2014, 2015; Peng et al., 2018). We note that unlike our proposal, MIXGREEDY does not produce the same \mathcal{R} samples for all iterations. That is our approach uses the same sample graph G'_r for simulation r at each iteration. The experimental results show that the differences in results in both cases are negligible.

Our next proposal, `HYPERFUSER`, borrows much from `INFUSER-MG` (Göktürk & Kaya, 2021), including hash-based fused sampling. `INFUSER-MG` computes the influence by memoizing connected components for all vertices and can work only on *undirected* datasets. It also employs the CELF optimization to reduce the cost of cardinality computations. On the other hand, `HYPERFUSER` can process both directed and undirected graphs and uses the Flajolet–Martin sketches in a novel way to estimate cardinality and choose seed candidates. As the experiments in the thesis show, its seed set quality is on par with that of `INFUSER-MG`. Our next algorithm, `SUPERFUSER`, is a multi-GPU based approach and inspires from `HYPERFUSER` by adopting its sample-space-split technique to boost the performance and distribute the work in an efficient way to multiple GPUs.

2.3 Related Work

In addition to the work mentioned above, which form the base of this thesis, there exist other Influence Maximization methods that can be considered as the state-of-art or have a component similar to our work. Even though studies using sketches, parallelism, or utilizing GPUs for IM exist in the literature, the proposed methods in this thesis are distinct from them in most ways.

The Independent Path Algorithm (IPA) (Kim, Kim, & Yu, 2013b) runs a proxy model and prunes paths with probabilities smaller than a given threshold in parallel. The approach only keeps a dense but small part of the network and scalable to only sparse networks. Liu et al. (2013) proposed IMGPU, an IM estimation method by utilizing a bottom-up traversal algorithm. It performs a single Monte-Carlo simulation on many GPU threads to find the reachability of the seed set. It is $5.1\times$ faster than MIXGREEDY on a CPU. The GPU implementation is up to $60\times$ faster with an average speedup of $24.8\times$. For comparison purposes, the techniques we propose and parallelization make the same algorithm faster around three orders of magnitude on multicore CPUs.

Although they can be inferior in terms of influence, modern IM algorithms are shown to be quite fast compared to conventional simulation-based approaches such as MIXGREEDY. Techniques such as using GPUs (Liu et al., 2013; Minutoli et al., 2020), sketches for finding set intersections (Cohen et al., 2014; Kim, Kim, & Yu, 2013b), reverse sampling to estimate the influence (Borgs et al., 2014; Minutoli et

al., 2019), and estimating the necessary number of simulations/samples required for each step (Leskovec et al., 2009) greatly reduces the execution times.

Sketch-based IM methods are cheaper compared to simulation-based methods. They usually pre-compute the sketches by processing the graph for evaluating the influence spread instead of running simulations repetitively. A popular method for sketch-based IM is SKIM by Cohen et al. (2014). SKIM uses combined bottom- k min-hash reachability sketches (Cohen & Kaplan, 2007; Cohen, 2015), built on ℓ sampled subgraphs, to estimate the influence scores of the seed sets. It is parallel in the sense that it uses `OpenMP` parallelization during sketch utilization. However, the sketch building step is single-threaded. SKIM treats vertex/sample pairs as distinct elements and reduces edge traversals via their smallest ranks in bottom- k sketches. SKIM builds its sketches on randomly assigned ranks to vertex/sample pairs. First, each vertex/sample pairs, (u, i) , are sorted by their random ranks for processing. Then, vertex/sampled pairs, (u, i) are processed in the order. For every incoming edge, (v, u) , of u in sample graph G_i , sketch X_u is updated by appending (v, i) 's rank to X_u , until any vertices' sketch size reaches j . After, the vertex with maximum estimated cardinality according to its sketch is added to the seed set S and reachability set on S is removed from all sketches and samples. The process is repeated until K vertices are select, continuing from the last rank processed. Unlike SKIM, our sketch-based proposals, HYPERFUSER and SUPERFUSER, leverages Flajolet-Martin sketches (Flajolet & Martin, 1985) for their simplicity, suitability for vectorization and fused sampling, and hence, execution-time performance.

Borgs et al. (2014) proposed Reverse Influence Sampling (RIS) which samples a fraction of all random reverse reachable sets. Then it computes a set of K seeds that covers the maximum number of those. The number of samples is calculated with respect to the number of visited vertices. The algorithm has an approximation guarantee of $(1 - 1/e - \epsilon)$. Minutoli et al. (2019) improved RIS and proposed IMM that works on multi-threaded and distributed architectures. Recently, the authors extended the algorithm to work on GPUs (Minutoli et al., 2020). IMM uses LeapFrog method to generate random numbers.

Ohsaka et al. (2014) proposed an approach that utilizes connected components and memoization. The proposed method selects the vertex with the highest degree and memoizes the connected components for all \mathcal{R} Monte-Carlo samples. Then, the marginal gains are estimated with the pruned-BFS method that prunes ancestor and recently processed vertices.

The Two-phased Influence Maximization (TIM+) algorithm borrows ideas from RIS but overcomes its limitations with a novel sampling strategy (Tang, Xiao, & Shi,

2014). Its first phase computes a lower bound of the maximum expected influence over all size- K node sets. It then uses this bound to derive a parameter θ . In the second phase, it samples θ random RR sets from G , and then derives a size- K node-set that covers a large number of RR sets.

Cohen (2015) presented the Historic Inverse Probability (HIP) estimators which, when applied to All-Distance Sketches, outperforms the HyperLogLog sketches (Flajolet et al., 2007) while estimating the number of distinct elements on data streams. As mentioned before, for the proposed algorithms in this thesis, we preferred Flajolet-Martin (Flajolet & Martin, 1985) instead of bottom- k min-hash sketches for simplicity and execution-time performance. In the future work, other estimators such as HIP can be used to improve the quality of our sketch-based solutions.

Kumar and Calders (2017) proposed the Time Constrained Information Cascade Model and a kernel that works on the model using versioned HyperLogLog sketches. The algorithm computes the influence for all vertices in G while performing a single pass over the data. The sketches are used for each time window to estimate active edges.

2.4 The Importance of Memory Hierarchy

In an ideal computing device, programmers would want more memory than the problem size which is also expected to be immediately available regardless of how data is stored. Unfortunately, both the solution complexity and the penalty due to access latencies force a multi-level memory hierarchy which contains faster yet smaller memory storage units. Hence, while architecture-agnostic implementations can suffer, when they are modified/refactored to promote data locality, we can access a large-amount of data in an effective and efficient manner. The principle of locality stems from the fact that programs tend to reuse data and instructions they have used recently. A conclusion we can draw from the locality principle is that it's possible to reorder the operations in a way which makes the local data fetched next within a reasonable accuracy. The most prevalent types of the locality are *temporal* and *spatial* locality. Temporal locality refers to accessing the same set of resources multiple times in a short time interval. On the other hand, spatial locality refers to the data access patterns that access in close proximity on memory.

Memory hierarchies are organized in several levels; each level below is slower yet

larger and cheaper. Locality allows multi-level memory hierarchies to be exploited while being transparent to programs, even if the hierarchies consist of memory blocks with different speeds and sizes. A multi-level memory hierarchy, optimized for common memory access patterns, can be fast as its fastest level for most of the run-time without the cost and latency limit of *addressing* large memory. Besides the cost, addressing latency is another factor to take into account while designing the memory subsystem. Since binary addressing is used, every time memory size is doubled, additional circuitry is required to fetch the relevant word, which increases access latency. For most of the memory hierarchies (with exceptions), the smaller subset is cached at a higher level to increase performance. Since most of the time, the programs access the data close to the previously accessed data, a data block containing each access location is also fetched from the lower/slower levels. In this scheme, a random access pattern would be heavily penalized since it causes each level to fetch from lower levels. On the other hand, a sequential access pattern only causes one block fetch from the lower-levels for all the accesses to the corresponding block. Pre-fetching strategies can even reduce the latency by guessing the next access and fetching it before while the processor is busy processing current data.

2.5 Amdahl's Law

Amdahl's law (Amdahl, 1967) refers to the theoretical speed-up that a computer system can achieve by adding more computation resources to a fixed-size problem. Since most of the programs has an inherently sequential part, we can only speed up the part of the program that can be parallelized. For example, a program that spends 90% of its execution time on a parallel task without any overhead can only be at most $10\times$ faster via parallelism; infinite computational resources can theoretically reduce the execution time of the parallel part to 0, yet the remaining 10% will still incur the same latency, even with infinite resources. We can state Amdahl's law in the following manner:

$$(2.1) \quad S(N) = \frac{1}{(1-p) + (p/N)}$$

where N is number of compute units for parallelization, p is fraction of time of the program's parallel sections, and S is the speed-up that can be theoretically achieved using N compute units. The argument neglects a few architectural considerations,

including memory bottlenecks, I/O, data-transfers, communication latencies, and the increase in problem size, i.e., parallelization overhead, due to the extra parallelization complexities.

Gustafson’s law (Gustafson, 1988) tackles the inadequacies of Amdahl’s law, which assumes a fixed problem size (work) with respect to additional computation resources. On the other hand, Gustafson’s law argues that programmers prefer to set the workload size to fully use the available computing resources and adapt as the resources become better. Hence, if the processors become better, the workload can be higher while keeping the processing time the same. Gustafson’s law can be stated in the following manner:

$$(2.2) \quad S(N) = (1 - p) + Np$$

where S is the theoretical speedup of the execution, N is the number of processors, and p is the time fraction of the execution that can leverage parallelization.

2.6 Memory Wall

Even heavily optimized kernels often perform well below hardware’s peak performance. The main reason for this sub-par performance is the performance difference between memory accesses and arithmetic computations. In addition, the performance disparity between memory and processor grows larger in time. The term “Memory Wall”, coined by Wulf and McKee (1995), refers to the phenomenon that even the low rate of misses (an access to a memory level without success) can dominate the run-time due to the growing latencies between the processor and lower-level memory. The increasing number of processing cores without the same rate of performance jump in DRAM bandwidth (as the time of writing this work) makes the memory wall more prevalent. For example, the memory hierarchy of a typical CPU and the expected latencies for each level are given below;

- Registers: ALU directly works on registers, typically less than a cycle.
- Level 0 (L0): Micro operations cache, holds microinstructions for complex instructions, less than a cycle.
- Level 1 (L1): Separate instruction and data caches. Very fast 1-2 cycles.
- Level 2 (L2): Shared instruction and data cache. 10-14 cycles.

- Level 3 (L3): Shared data cache between the cores in the same module/socket. Allows communication. 20+ cycles.
- Level 4 (L4): Shared data cache between the cores in the same package. Allows communication. 50+ cycles.
- Main memory (RAM): Main random access data storage. 300+ cycles.

The average memory access time equation can be written as:

$$(2.3) \quad t_{avg} = p \times t_c + (1 - p) \times t_m$$

where t_c is the cache access latency, t_m is main memory access latency, and p is cache hit ratio. For simplicity, multiple levels of caches are not considered but can be easily added by replacing t_m with average latency for lower levels. Even if the term $(1 - p)$ is small, which is a must for good performance, making it zero is impossible for most of the problems. Hence, as the difference between t_c and t_m grows, the performance will suffer more. Therefore, we will hit the memory wall regardless. Furthermore, graph algorithms have to follow the network edges (or endpoints of the edges) distributed into irregular and unpredictable memory locations, hence, suffer significantly from the cache misses. Probabilistic graph algorithms especially suffer since many edge traversal operations during sampling do not contribute to the output, i.e., incur a useful arithmetic operation, although they incur more cache misses.

2.7 Single Instruction-Multiple Data

Flynn's taxonomy defines *Single Instruction-Multiple Data* (SIMD) as a class of parallel computers that can operate on data vectors. SIMD architectures allow parallelism at the instruction level. Data vectors hold a fixed number of elements in consecutive locations. SIMD instructions work by applying operators on respective locations in alignment (with some exceptions, i.e., shuffle). A vector addition example is given in Fig. 2.2. The part on the left, Fig. 2.2a, illustrates a simple addition operation, the values (1,5), (2,6), (3,7), and (4,8) are loaded into registers, respectively, then the results (6), (8), \dots are stored into the memory. Eight loads, four addition operations, and four stores are required to complete which are performed as $4 \times (2 \text{ loads} + 1 \text{ addition} + 1 \text{ store})$. The figure on the right, Fig. 2.2b, illustrates the

same operation in SIMD manner; items (1, 2, 3, 4) and (5, 6, 7, 8) gets loaded into SIMD registers, then the results (6, 8, 10, 12) stored into the memory. Two SIMD loads, one SIMD addition, one SIMD store are required to complete the operation.

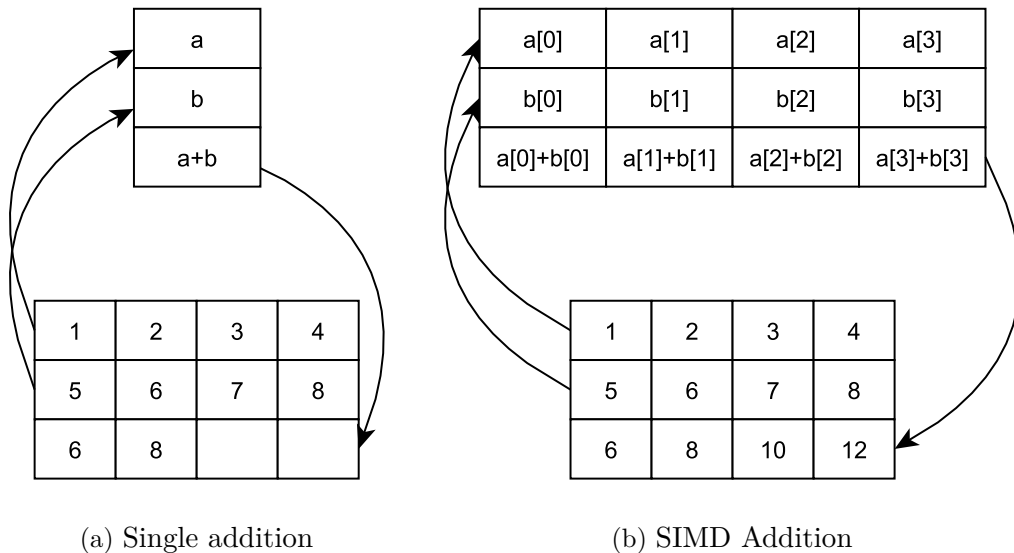


Figure 2.2 Difference between serial addition(a) and SIMD addition(b)

A SIMD architecture allows a single instruction to process a large amount of work in parallel. Even though multi-scalar processors (i.e., almost all modern processors) can process multiple instructions, SIMD has the edge over it. Using vector instructions reduces multiple decode operations of the same instruction to a single decode. Furthermore, it also reduces the number of instructions to be processed. When accessing the memory, vector instructions can access aligned consecutive memory faster, further reducing the overheads.

Even though early SIMD architectures were part of Supercomputers, modern SIMD architectures stem from desktop multi-media accelerators. With the inception of 128-bit MMX vector extensions, many enhancements have been implemented in modern processors. In this work, we utilized two SIMD architectures; Advanced Vector Extensions 2 (AVX2) instruction set available in Intel and AMD CPUs, Compute Unified Device Architecture(CUDA) available in Nvidia GPUs.

Advanced Vector Extensions 2 (AVX2) also known as Haswell New Instructions, are extensions to the standard x86 instruction set architecture for micro-processors and proposed by Intel in March 2008. AVX2 provides new features, new instructions, and a new coding scheme. AVX2 works on 256-bit registers in many packed forms, including all possible addressing schemes; 1x256, 2x128, 4x64, 8x32, 16x16, and 32x8 bits storage arrangements are supported. The instruction set scales up most integer commands to 256 bits and introduces fused multiply-

accumulate (FMA) operations. In addition, AVX2 supports “Gather” instructions that allow reducing 256-bit vectors to 32/64 bitmasks.

In this work, we employed the AVX2 instruction set. We added the corresponding vector instructions manually to the code since, even though compilers translate and optimize most of the loops to vectorized forms, *compare and move-mask* operations were detected not to be recognized by auto-vectorization in our preliminary experiments. For completeness, the intrinsics explicitly used are described in Table 2.2.

Table 2.2 AVX2 intrinsics used in the implementation.

Intrinsic	Definition
<code>_mm256_set1_epi32</code>	Initializes 256-bit vector with scalar integer values. Does not map to any AVX instructions.
<code>_mm256_and_si256</code>	Performs bitwise logical AND operation on 256-bit integer vectors.
<code>_mm256_xor_si256</code>	Performs bitwise logical XOR operation on 256-bit integer vectors.
<code>_mm256_cmpgt_epi32</code>	Compares packed 8x 32-bit integers of two input vectors.
<code>_mm256_movemask_ps</code>	Extracts the first bits of 8x 32-bit elements in a compact 8-bit format
<code>_mm256_blendv_epi8</code>	Blends/selects byte elements of input vectors depending on the bits in a given mask vector.

AVX512 expands AVX2 to 512-bit vector support and proposed by Intel in July 2013. Even though AVX2 was the main target for most of the optimization in this thesis, we have also used AVX512 to test the scalability of our approach. The instruction set consists of multiple extensions that can be implemented independently. This policy is a departure from the historical requirement of implementing the entire instruction block. Only the core extension AVX-512F (AVX-512 Foundation) is required by all AVX-512 implementations.

In addition to vector arithmetic/logical operations, AVX512 supports gather/scatter operations. This way, non-vectorized operations can be done side by side with vectorized operations, removes the need of complex shuffle/mask operations. In our experiments, single-core AVX512 implementation was relatively fast due to the hardware-supported gather instruction. Yet, the approach did not scale to multiple cores due to “*frequency scaling*”. Unfortunately, at the time of this work, AVX512 implementations did suffer from heat and energy limits (Lemire, 2018; Krasnov, 2017), only by instruction weaving (using AVX2 instructions in between AVX512 instructions) can perform comparatively well. However, using too many AVX512 instructions can slow the whole core (Krasnov, 2017) to base clock speeds.

2.8 Compute Unified Device Architecture (CUDA)

Compute Unified Device Architecture, often used as its abbreviation, CUDA, entails running programs on Nvidia GPUs for acceleration. Even though GPUs are used to be associated with computer graphics, currently, they have been extensively used for numerical computations, optimization, machine learning, deep learning, etc., due to their massive parallelism power with thousands of cores and low energy cost per operation. At the time of this thesis, more than ten thousand cores in a single GPU (device) are available. Because of this, they are well-suited to computations that benefit from parallel processing. These being said, CUDA devices and their execution system are different than today's x64 systems, and it is critical to understand those distinctions. The efficient utilization of CUDA devices requires a thorough understanding of the programmatical and architectural differences and how they determine performance.

The most significant execution difference between CUDA and x64 programming is the threading model. We see a smaller number of cores on a CPU (host) that execute a small number of instructions simultaneously with higher instructions per cycle. Whereas on CUDA systems, a single *Streaming Multiprocessor* can execute 2048 active threads concurrently, and there exist up to 82 SMs per device. Currently, it is possible to have 10496 CUDA cores in a single device. Nevertheless, comparing the number of threads between GPUs and CPUs can be misleading: CPU threads are smarter with higher instruction counts per cycle. Compared to GPU threads, they employ better branch/instruction prediction, with deeper cache hierarchy and faster random access to the memory. However, CPU threads have very slow context switches. On the other hand, threads on GPUs are extremely lightweight. Thousands of threads (in groups of warps/wavefronts of 32 threads) are queued for work in proper CUDA applications.

If the GPU would have to wait for one warp of threads to complete, it immediately starts working on another. Since all threads use different registers, the GPU can switch between contexts without saving the local state, i.e., with zero overhead. The threads in a warp are executed together; if the warp threads take multiple branches, GPU has to switch back and forth between branches, significantly reducing the performance. For a better GPU utilization, one should cluster the branches if possible. For example, a kernel keeps branching on the last bit of sorted integers, reordering the list with respect to its last bit would improve performance significantly; in the former, we can only expect half utilization since a warp should execute both paths.

On the other hand, in the latter, the warp will only execute single branch in most cases.

Running programs on the GPUs efficiently requires considering if the problem needs processing many values in a similar fashion and how much communication is required (transferring data to devices can be costly). For example, few small 4x4 matrix multiplications for a video game logic or inner product (i.e., the dot product) of two vectors might be faster in CPUs since few vector instructions can do it in few cycles, whereas transferring the data to GPU and back has a high overhead compared to the former. This implies that, we should keep the data on the GPU memory for the best performance as long as needed, and continuous GPU-CPU communication must be restricted.

When the threads in a single warp, being executed together, working on adjacent/coalesced memory locations, the performance is higher. Memory coalescing is important since warps are practically SIMD units executing the same instruction on wider vectors. Hence, coalesced load/store operations are consuming fewer cycles and lower overhead. In addition, for a slightly better performance, each item should be aligned to multiples of the warp size (32). Especially in algorithms with random access patterns, such as graph algorithms, the computation might need a careful restructuring to achieve all these. In this work, the algorithms are designed so that the sequential accesses follow random accesses to compensate the short-comings of GPUs (to an extent, CPUs with AVX2 also benefits).

The GPU has its own physically distinct but restricted global memory, and data should be transferred from the main memory to GPUs' memory back and forth on a regular basis. For small communications, it is possible for CPU and GPU to access each other's memory, but for maximum throughput, (possibly proactive) data transfers are required. In addition, the GPUs have registers and local for storage (given in order of increasing latency).

Table 2.3 GPU Memory Hierarchy

Memory	Location	Cached	Scope
Registers	On-chip	-	Thread
Shared	On-chip	-	Block
Local	Off-chip	Yes	Thread
Global	Off-chip	Yes	Device

CUDA programs have a task hierarchy of grid, blocks, and threads. A kernel runs in a 1D, 2D or 3D grid that consists of blocks. Similarly, each block can also be 1D, 2D, and 3D and contains a number of threads. For the best performance, one should do the composition of the tasks with respect to the computation pattern and

communication structure of the algorithm.

A significant difference on programming CPUs and GPUs is the transparency of the memory hierarchy. In CUDA, the scope of the data items is more prevalent. Registers are only accessible from the thread they are used, so context switches are not as expensive. However for the context-switches in a CPU, the registers are required to be stored in the stack. The communication among the warp threads can be done using ballot or shuffle instructions. In addition, the threads in a block can communicate on predefined shared memory. The global memory, which is the largest memory block, is accessible to all threads on the device, and there exist device-wise atomic instructions for global synchronization.

3. FUSED SAMPLING

Probabilistic graph algorithms often employ a sampling step which processes probabilities and extracts a sample. The performance of sampling is important since the transformation cost, i.e., the cost of extraction, creation, storage and processing of the samples do not overwhelm the overall execution. This is why fusing the sampling step with actual computation can improve performance drastically. Also, deterministic sampling strategies which keep the simulation quality of the process can further enhance performance.

3.1 Hash-Based Sampling

Generating random values fast and fair is a complex problem. In this work, we focus on random numbers good enough but not truly random. The use of good enough pseudo-random numbers allows us to compute them fast and devise random-number generation (RNG) schemes that we can exploit for performance without hurting statistical probabilities significantly. Most common random generation method, implemented as `rand()` function by most C compilers, is Linear Congruential Generator (LNG) (Thomson, 1958). LNG method starts from a seed value X_0 , then generates a sequence of random numbers using the recurrence $X_{n+1} = aX_n + b \bmod m$ where a , b , m are fixed, large (possibly prime/co-prime) integers. Since the recurrence requires reading and updating the previously generated values, its simple parallel use creates race conditions, false sharing, and pressure on cache/memory. The same weakness applies to some other recurrent random number generation schemes as well, such as the Mersenne Twister (Matsumoto & Nishimura, 1998), which is another RNG that is implemented by C++ standard.

For parallel/distributed systems, *random tree* and *LeapFrog* methods are preferable

RNGs in most scenarios. Both methods are variants of the LNG method. The random tree method uses a binary divergent recurrence $L_{k+1} = a_L L_k + b_L \text{ mod } m$ and $R_{k+1} = a_R R_k + b_R \text{ mod } m$ where a, b are different for all divergent paths. Left generator L generates many L_n values that are used as the base for right generators R^n . Then, many R generators can be used to generate pseudo-random numbers independently. For a fixed number of generators, the LeapFrog method can be utilized to generate non-overlapping pseudo-random sequences. The method is useful for scenarios requiring one generator for each task in a domain decomposition. The random sequence is defined as follows; $L_{k+1} = aL_k \text{ mod } m$ and $R_{k+1} = a^n R_k \text{ mod } m$. Similarly to random tree, the actual random numbers are generated by the R generators. Instead of many generators generating distinct sequences, *LeapFrog* uses many generators to generate a common sequence from different places. Each generator skips n elements, so the sequences generated by different generators are non-overlapping. Even though the *LeapFrog* method has superior properties that are orthogonal to parallel/distributed algorithms, it is not specialized to support graph sampling.

For sampling on graphs, using the edge information, i.e., the endpoints' IDs, for generating random values allows us to specialize the random number generator for graph sampling. Even though vertex ids are distinct, they are often correlated to vertex ID on the other end of the edge. Also, bits of vertex IDs are highly biased, assuming they are not given in random. These properties of the vertex IDs make them a terrible input for pseudo-random number generation. However, since they are distinct for each edge, using a good hash function provides uncorrelated, unbiased numbers that are used for building blocks of our pseudo-random number generation scheme.

In this thesis, we propose a hash-based method pseudo-random number generator for graph sampling. The proposed method utilizes a hash function $h(u, v)$ that takes the source and target vertex IDs as inputs and returns the hash value generated for the edge e_{uv} . Then, the hash is XOR'ed with another random value associated with the specific sample to get random for the edge in that sample. Finally, the resulting value is divided by the hash functions limit to generate the random value.

3.2 Fused Sampling

It is possible to restructure the algorithms so that whether an edge or a vertex will be processed is decided on the fly without computing samples beforehand. The probabilities are realized while performing actual computations in the fused sampling method. Lazy evaluation of chances opens possible performance exploits. The most important one is that fused sampling removes the need for extra copying of the sample/graph.

3.2.1 Hash-based Fused Sampling

Traditionally, the cascade model requires a new sample, i.e., a subgraph, from $G = (V, E)$ to simulate the diffusion process. The probabilistic nature of cascade models requires sampling subgraphs from $G = (V, E)$ to simulate the diffusion process. If performed individually as a preprocessing step, as the literature traditionally does, sampling can be an expensive stage, furthermore, a time-wise dominating one for the overall IM kernel. We identify two main bottlenecks; first, sampling multiple sub-graphs may demand multiple passes on the graph, which can be very large and expensive to stream to the computational cores. Second, if samples are memoized, the memory requirement can be a multiple of the graph size. We eliminate the necessity of the creation and storage of the sample subgraphs in memory. In Fig. 3.1, we briefly illustrate fused-sampling; instead of processing the samples independently as in Fig. 3.1a, fused-sampling processes each edge concurrently for multiple simulations as shown in Fig. 3.1b. This allows us to process each edge only a few times instead of once for every simulation.

When an edge of the original graph is being processed, it is processed for all possible samples. Then, it is decided to be *sampled* or *skipped* depending on the outcome of the hash-based random value for each sample. Given a graph $G = (V, E)$, for an edge $(u, v) \in E$, the hash function used is given below:

$$(3.1) \quad h(u, v) = \text{MURMUR3}(u||v) \bmod 2^{31}$$

where $||$ is the concatenation operator. In our preliminary experiments, we have tried a set of hash algorithms. After a careful analysis, we chose MURMUR3 due to its simplicity and good avalanche behavior with a maximum bias 0.5% (Appleby, 2017). Although the approach mentioned above generates a unique hash value for each edge, and hence a unique sampling probability, different simulations require different probabilities. First, a set of uniformly randomly chosen numbers $X_r \in_R [0, h_{max}]$

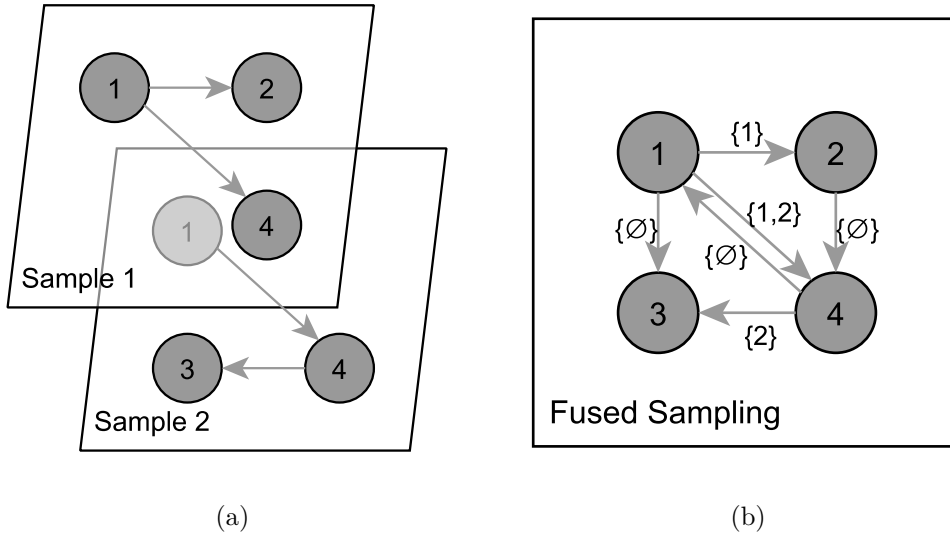


Figure 3.1 (a) Two sampled subgraphs of a toy graph with 4 vertices and 6 edges. (b) The simulations performed are fused with sampling. Each edge is labeled with the corresponding sample/simulation IDs.

associated with each simulation r are generated to enable this for each edge. Then the sampling probability of (u, v) for simulation r , $P(u, v)_r$, is computed: To do this, the hash value, $h(u, v)$, is XOR'ed with X_r and the result is normalized by dividing the value to the upper limit of the hash value h_{max} . Formally,

$$(3.2) \quad P(u, v)_r = \frac{X_r \oplus h(u, v)}{h_{max}}.$$

The edge (u, v) exists in the sample r if and only if $P(u, v)_r$ is smaller than the edge threshold $w_{u, v}$. One of this approach's benefits is that an edge can be sampled using a single XOR and compare-greater-than operation. Moreover, the corresponding control flow branch overhead can be removed using *conditional move* instructions.

Using a strong hash function such as MURMUR3 ensures that all bits independently change if the input is changed. This property allows us to generate good enough pseudo-random values for fair sampling. To evaluate the randomness and fairness of the values generated with the hash-based approach, we generated a large number of samples for various real-life networks and plotted the cumulative distribution (Fig. 3.2) and the bias of the random values $P(u, v)_r$ used (Fig. 3.3). As the former shows, the sampling distribution of the hash-based computation values resembles a uniform random distribution. Furthermore, the latter shows that the bias is insignificant for each network.

Being able to generate the samples on the fly allows us to avoid many memory accesses. The only downside of hash-based fused sampling is that we have to generate

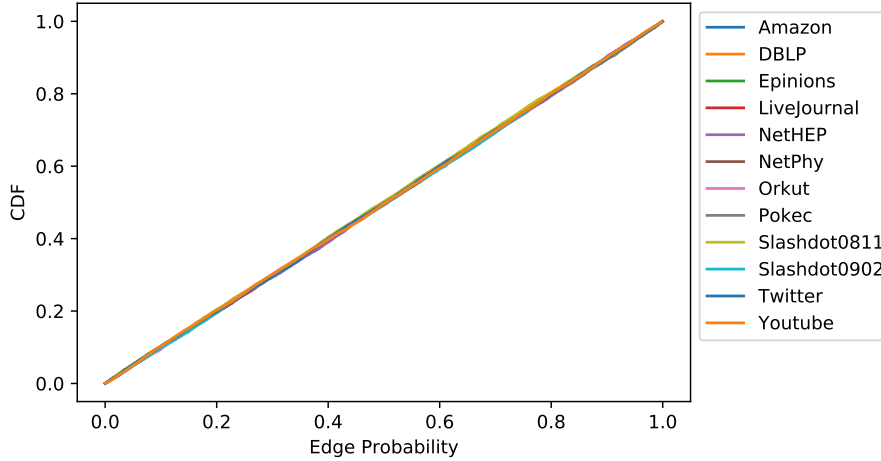


Figure 3.2 Cumulative probability function of hash-based sampling probabilities on various real-life networks.

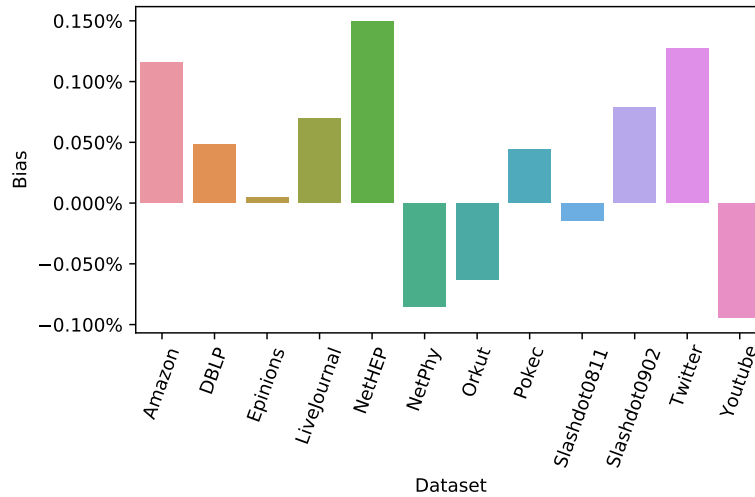


Figure 3.3 Bias distribution of hash-based sampling probabilities on various real-life networks.

all these random values, $P(u, v)_r$, for each edge traversal and each simulation r . The edges' hash values are pre-computed for all the edges in E to reduce computation cost and leverage fused sampling's performance gains. Fortunately, the rest of the operations, i.e., one XOR and one division, are very fast on modern computing hardware. The trade-off between extra memory and computation here maybe not be applicable for different computation architectures and faster/simpler hash functions. In our experiments, we found that CRC32 (Koopman, 2002) is an another good candidate for the hash function. CRC32 can be applied using specialized instructions in 3 cycles latency. Using CRC32 slightly slows down the execution (<10%) yet reduces memory use by nearly half. Unfortunately, CRC32 does not have the randomness properties mentioned about Murmur3.

3.2.2 Direction oblivious hash-based sampling

Undirected diffusion models (Chen, Wang, & Yang, 2009) requires the same probability, and the same random number is generated for forwards and backward edges. State-of-the-art implementations (Chen, Wang, & Yang, 2009) sample edges from E and add them to a set along with reversely oriented edges to make the subgraph, which is constructed from this sampled edge set, undirected. This approach requires a complete traversal of G , followed by a relatively expensive integration of the reversed edges. The algorithms proposed in this work do not explicitly sample. Whenever an edge with a certain orientation is read from memory, it is sampled or skipped depending on the outcome of direction-oblivious sampling that assigns the same sampling probability for both directions, (u, v) and (v, u) .

Unlike the directed variant, we utilize a hash function $h(u, v) = h(v, u)$ to get the same probability for forward and backward directions within the same simulation. The hash function used is

$$(3.3) \quad h(u, v) = \text{MURMUR3}(\min(u, v) \parallel \max(u, v))$$

where \parallel is the concatenation operator. To avoid the cost of hashing during simulations, all possible hash values are pre-computed. Although there exist $\frac{n \times (n-1)}{2}$ possible vertex pairs, we only need the vertex pairs having an edge in between, i.e., only m hash values are pre-computed. We have tried a few other hash algorithms as well; Ideally, we chose MURMUR3 (Appleby, 2017) due to its simplicity and good avalanche behavior with maximum bias 0.5% (Appleby, 2017).

Although the approach, as mentioned earlier, generates a unique hash value for each edge and hence a unique sampling probability, different simulations require different probabilities. To achieve this, we use a random number X_r for each simulation r . To compute the sampling probability of $\{u, v\}$ during r th simulation, $h(u, v)$ is first XOR'ed with a uniformly randomly chosen $X_r \in_R [0, h_{max}]$ and the outcome is divided to the maximum possible hash value h_{max} . Let $\rho(u, v)_r$ denote this sampling probability for $\{u, v\}$ in simulation r . Formally,

$$(3.4) \quad \rho(u, v)_r = \frac{X_r \oplus h(u, v)}{h_{max}}.$$

The edge $\{u, v\}$ is verified to be in the sample if $\rho(u, v)_r$ is smaller than or equal to the threshold $w_{u,v}$. With the proposed approach, sampling an edge reduces to an XOR and compare-greater-than operation. The branching on the latter can be removed to enable SIMD instructions as explained in Section 2.7.

MURMUR3 guarantees a change on the 50% of the bits when a single bit of the input changes. Furthermore, all bits independently change when the input is changed. These properties allow us to generate good pseudo-random values to simulate the process. For practical considerations, we stored all the $\rho(u, v)_r$ s generated for various real-life networks and plotted the Cumulative Distribution Function (CDF) of these values. For a given graph $G = (V, E)$, the CDF of a sampling probability x is computed as $\Pr(x \leq \rho(u, v)_r)$ for all $\{u, v\} \in E$ and $0 \leq r < R$. Figure 3.2 shows the CDFs for 12 real-life networks. The sampling probability distribution with hash-based computation is almost identical to the uniform distribution, which is required to simulate the diffusion process.

In proposed methods, the diffusion is performed on a subgraph which is never constructed; in fact, each diffusion is simulated on G . Thus the overhead of generating and storing a sample and reading it back from memory is avoided. However, for each visit of $\{u, v\}$, since INFUSER-MG does not know if the edge is in the sample or not, $\rho(u, v)_r$ is recomputed. Another immediate benefit of fusing is traversing only the vertices that contribute to influence score and their neighbors. On the other hand, a non-fused implementation would traverse all edges for all simulations. Often, the total influence is a very small fraction of the total number of vertices and hence, fusing is vital to have a scalable IM kernel.

4. BOOSTING INFLUENCE-MAXIMIZATION KERNELS WITH FUSED SAMPLING

Classical Monte-Carlo based IM algorithms first sample a sub-graph and then perform a single simulation. Such an approach is amenable to thread-level, coarse-grain parallelization since the simulations are independent of each other. However, this requires the graph to be read from the memory for every simulation. The state-of-the-art implementations use this *one-sample-per-simulation* approach and build a unique graph for every sample to find the marginal influence scores (Chen, Wang, & Yang, 2009). With coarse-grain parallelization, this makes the IM kernels inefficient in terms of performance since the graphs are sparse (and samples are sparser), memory accesses are irregular, and performing a single simulation per graph traversal increases the already hindering pressure on the memory subsystem and makes the IM process further memory bound. As mentioned before, to make the IM computations faster, heuristics, sketches, and proxy models have been proposed in the literature. Unlike these, INFUSER-MG exploits the properties of the greedy Monte-Carlo algorithm. It is tuned for the undirected graphs and the Independent Cascade model. However, the techniques such as fusing can be adopted by the other models or Monte-Carlo graph algorithms using sampling. INFUSER-MG leverages three techniques to achieve its goals.

- Instead of explicitly constructing a data structure for each subgraph, the proposed approach uses direction-oblivious pseudo-random numbers throughout the edge-based simulation to fuse the sampling with the computation of influence scores.
- To reduce the memory subsystem pressure, INFUSER-MG leverages batched simulations and instruction-level parallelism and when possible, utilizes each edge access for multiple simulations.
- To reduce the number of operations performed, the component IDs for each vertex and sampled subgraph are memoized which can then be used while computing the marginal gains during the CELF stage.

On top of these, multi-core parallelism is applied to further increase the performance by running multiple threads and assigning each batch to a different thread.

To handle parallel edges faster, as a preprocessing step, INFUSER-MG calculates the joint probability of these edges with (4.1).

$$(4.1) \quad \{u, v\} \in G_r \iff \rho(u, v)_r < 1 - (1 - p)^{|E_{u,v}^G|}$$

That is rather than trying each of these edges one by one, INFUSER-MG treats them as a single but more powerful edge.

4.1 Vectorized Monte-Carlo graph traversal

In MIXGREEDY (Algorithm 3), both the NEWGREEDY step and marginal gain computations utilize graph sampling. By leveraging vectorization, a single thread in INFUSER-MG can process a batch of B samples/simulations at once. A high-level visualization of how the samples are batched is given in Figure 4.1. In a perfect, fused, and batched execution, the edges (of the original graph) flow from the memory to the cores and they are consumed by carefully structured SIMD kernels. Once an edge is visited, all \mathcal{R} simulations are taken into account by batches of B simulations. Although fusing and vectorization can incur redundant computations, as the experiments will show, the proposed approach significantly boosts the performance.

4.1.1 A vectorized NEWGREEDY step

For an undirected graph, the NEWGREEDY step of MIXGREEDY needs to identify the reachability sets $R_{G'}(v)$ for all $v \in G'$. Traditional IM implementations work on a single subgraph and initiate many graph traversals until all vertices are visited. The time complexity of this process is linear in terms of the number of vertices and edges. However, its memory access pattern tends to be irregular; many random memory accesses are required which results in low CPU utilization. Instead of graph traversal, e.g., Breadth-First Search, the connected components within a sampled subgraph can be found via *connected-component labeling* (Rosenfeld & Pfaltz, 1966), which starts by assigning unique labels to each vertex. Then at each iteration, the

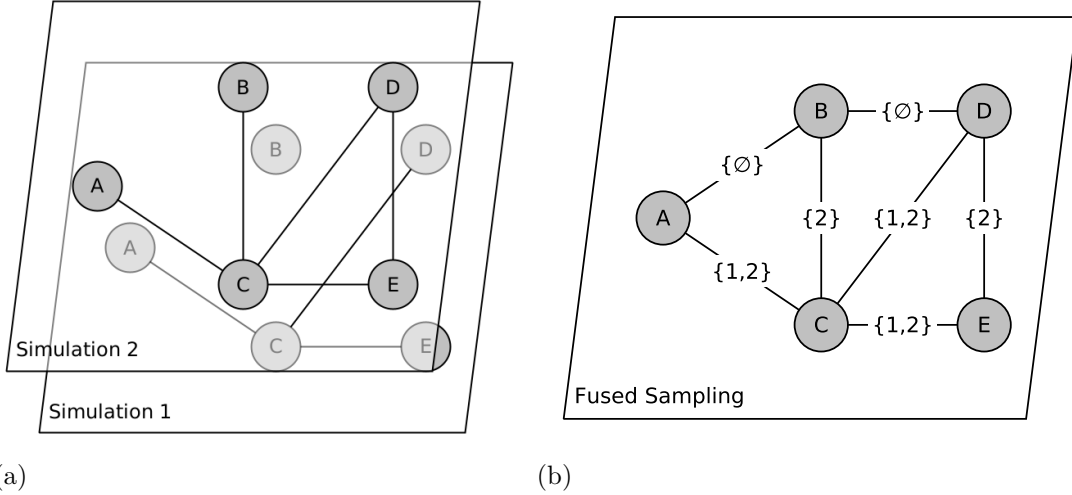


Figure 4.1 (a) Two sampled subgraphs of the toy graph from Figure 2.1a with 5 vertices and 3 and 5 edges, respectively. (b) The simulations are performed in a way to be fused with sampling. Each edge is labeled with the corresponding sample/simulation IDs.

edges are visited and the labels of both endpoints are set to the minimum of the two. This process continues until convergence; i.e., no label is changed within a single iteration. The total amount of work performed by this algorithm is superlinear since each edge is touched at each iteration. To reduce the time complexity, one can mark the (*live*) vertices whose labels are updated in the current step, and only process their edges in the next step. Although this does not guarantee a linear-time algorithm, it significantly reduces the number of edge accesses.

INFUSER-MG runs the above-mentioned, label-propagation-based approach in a fused and batched manner. For all \mathcal{R} samples, the propagation is simulated on the original graph G by taking only the sampled edges into account. Taking the SIMD register sizes into account, e.g., $8 \times 32\text{bit}$ for AVX2, the simulations are processed on batches of $B = 8$ samples which are never constructed. To do that, the existence of the edge in these samples is rechecked every time it is being processed. All the live vertices within a single iteration are processed in parallel by multiple threads. Further parallelization at this stage comes from running B simulations at once in a SIMD fashion. An example run with $\mathcal{R} = B = 2$ simulations is given in Figure 4.2 continuing from Figure 4.1. Although the label-propagation-based approach requires superlinear sequential work, even when live vertices are marked, we prefer the propagation-based approach to the traversal-based approach since propagation is more vectorization friendly, and running B fused simulations does not require any explicit synchronization. On the other hand, doing so with the traversal-based algorithm requires extra bookkeeping to go for the next component

after a component is processed.

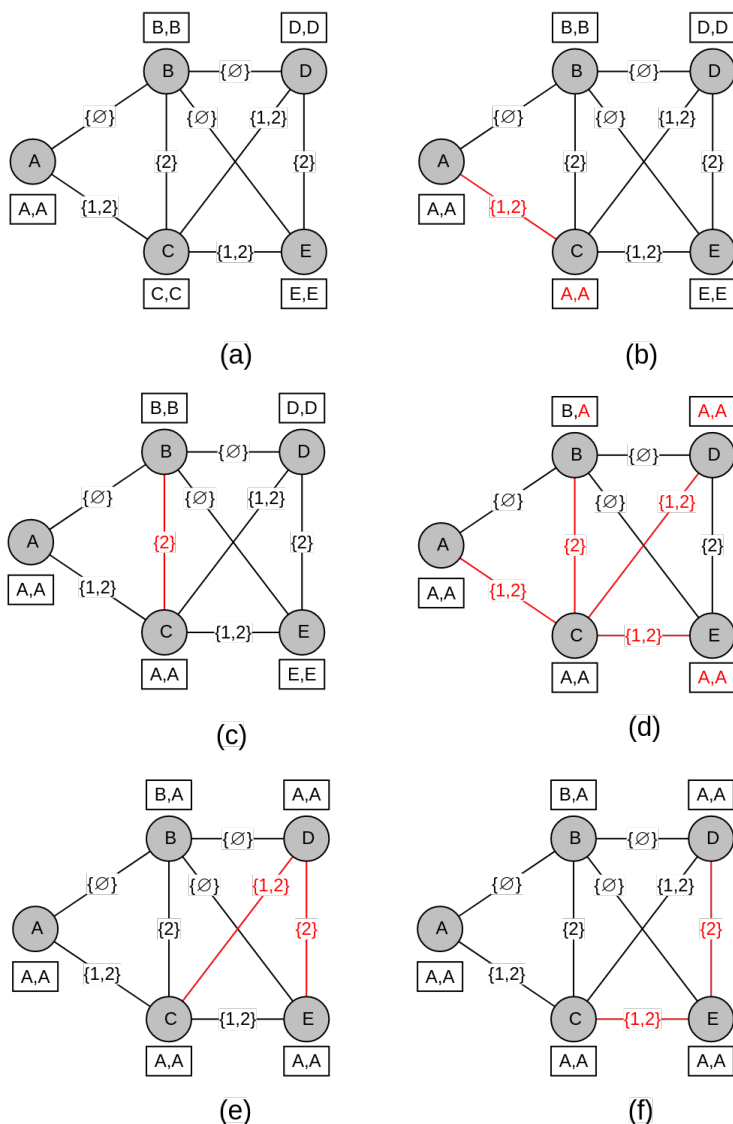


Figure 4.2 (a) The initial state on a toy graph for connected-component labeling; all vertices are labeled with their ids. (b) First, the edges of A are processed; the edge to C is in both samples. C 's labels are updated. (c) B 's edges are processed. The edge to C exists in the second sample. C 's second label is smaller, hence no update is performed. (d) C 's edges are being processed. It has edges to A , B , D , and E in the samples. The labels $\langle A, A \rangle$ are propagated to D and E since the edges are in both samples. B 's second label is updated because only sample 2 contains the corresponding (C, B) edge. (e)(f) D and E edges in the samples. However, there are no updates.

Algorithm 5 describes the fused and vectorized `NEWGREEDYSTEP-VEC`. The algorithm takes two inputs G , the original graph, and \mathcal{R} , the number of simulations. It works along the same lines with the original `NEWGREEDY` with additional operations for connected-component labeling. The labels for each vertex are initially set as the vertex IDs (lines 1– 2). The outer **while** loop checks if there exist any more live vertices. Here, a vertex is said to be *live* if at least one of its \mathcal{R} labels is changed

Algorithm 5 NEWGREEDYSTEP-VEC(G, \mathcal{R})

Input: $G = (V, E)$: the influence graph

\mathcal{R} : number of MC simulations per seed vertex

Output: mg : marginal influence scores

l : connected component labels

```
1: for  $v \in V$  do
2:    $l_v \leftarrow [v, \dots, v]_{\mathcal{R}}$ 
3:  $\mathcal{L} \leftarrow V$ 
4: while  $\mathcal{L}$  is not empty do
5:    $\mathcal{L}' \leftarrow \emptyset$ 
6:   for  $u \in \mathcal{L}$  in parallel do
7:     for  $v \in \Gamma_G(u)$  do
8:        $r \leftarrow 0$ 
9:       while  $r < \mathcal{R}$  do
10:        for  $r' = r \dots r + 7$  do  $\triangleright B = 8$ 
11:          if  $\rho(u, v)_{r'} \geq w_{u, v}$  then
12:             $min_{label} \leftarrow \min(l_u[r'], l_v[r'])$ 
13:            if  $min_l \neq l_v[r']$  then
14:               $l_v[r'] \leftarrow min_l$ 
15:               $\mathcal{L}' \leftarrow \mathcal{L}' \cup \{v\}$ 
16:             $r \leftarrow r + 8$ 
17:    $\mathcal{L} \leftarrow \mathcal{L}'$ 
18: for  $v \in V$  in parallel do
19:    $mg_v \leftarrow 0$ 
20:   for  $r = 1 \dots \mathcal{R}$  do
21:      $mg_v \leftarrow mg_v + |\{u : l_u[r] = l_v[r]\}|$ 
22: return  $mg, l$ 
```

Algorithm 6 VECLABEL $(r, u, v, X_r, l_u, l_v, r)$

Input: r : ID of the first simulation in the current batch

u : source vertex

v : target vertex

X_r : random number for simulations r to $r + 7$

l_u : vector of component labels of u

l_v : vector of component labels of v

Output: l_v : labels of vertex v after traversing edge (u, v)

$live_v$: a boolean which state if v is live

```
1:  $mask \leftarrow \_mm256\_cmpgt\_epi32(l_u[r], l_v[r])$ 
2:  $labels \leftarrow \_mm256\_blendv\_epi8(l_u[r], l_v[r], mask)$ 
3:  $hashes \leftarrow \_mm256\_set1\_epi32(hash(u, v))$ 
4:  $probs \leftarrow \_mm256\_xor\_si256(hashes, X_r)$ 
5:  $w_{vec} \leftarrow \_mm256\_set1\_epi32(\lfloor w_{u,v} \times INT\_MAX \rfloor)$ 
6:  $select \leftarrow \_mm256\_cmpgt\_epi32(w_{vec}, probs)$ 
7:  $l_v[r] \leftarrow \_mm256\_blendv\_epi8(l_v[r], labels, select)$ 
8:  $live_v \leftarrow \_mm256\_movemask\_ps(\_mm256\_and\_si256(select, mask))$ 
9: return  $l_v, live_v$ 
```

during the previous iteration. The first inner **for** at line 6 loops over the live vertices in a multi-threaded fashion. A single thread runs the next **for** loop at line 7 to visit the edges of the live vertex being processed. The operations corresponding to each of the \mathcal{R} simulations are performed for a visited edge (u, v) in batches of 8.

For each $0 \leq r < \mathcal{R}$, where r is a multiple of $B = 8$, the vectorized steps that perform the operations in simulations r to $r + 7$ are given between lines 10–15. These steps are performed as described in Algorithm 6, VECLABEL. The algorithm first compares the labels using element-wise compare intrinsic `_mm256_cmpgt_epi32` which returns all 1's ($2^{32} - 1$) when the first value is larger, and 0 otherwise. Then, pairwise minimum of the labels from the two vectors can be selected by `_mm256_blendv_epi8` that employs the $mask$ entries generated by the previous step. This intrinsic selects the bytes from the first vector if the corresponding $mask$ entry is not zero. Otherwise, it selects the bytes from the second vector. Hence for an edge $(u, v) \in E$, the resulting vector, $labels$, contains the smaller of the endpoints', i.e., u 's and v 's labels, for each simulation. The edge (u, v) may not have been sampled by all simulations. To find the simulations it is sampled, the algorithm generates the sampling probabilities by XORing the corresponding hash, $h(u, v)$, and the random values, X_r . Being computed in the preprocessing step, the hash is promoted to a vector, $hashes$, by the intrinsic `_mm256_set1_epi32`. The XOR operations are performed in a SIMD fashion with the intrinsic `_mm256_xor_si256`. We then promote $w_{u,v}$ to a vector w_{vec} by first multiplying it with `INT_MAX` using the `_mm256_set1_epi32` intrinsic. Then, this vector is element-wise compared

to the vector *probs* by using `_mm256_cmpgt_epi32`. The result of this operation is the *select* vector containing the selection masks for simulations. Blending *labels* (from line 2) with *v*'s current labels based on the *select* entries produces *v*'s final labels for the corresponding simulations r to $r + 7$ by using the intrinsic `_mm256_blendv_epi8`.

After the new labels are computed, we check if any of *v*'s labels are modified to verify whether the process is converged or not. To do this, we first perform bitwise-and operations for the elements in *mask* and *select* by using the intrinsic `_mm256_and_si256`. Then, the first bits of the 32-bit elements are extracted in a compact 8-bit format by using the `_mm256_movemask_epi8` intrinsic. This intrinsic eliminates 8 comparison branches and produces a **boolean** variable $live_v$. As mentioned above, at each iteration, the algorithm only processes the vertices whose labels are changed in the previous iteration. Initially, all the vertices are considered live. Each thread uses these $live_v$ values to keep track of the set \mathcal{L} of live vertices. To do this, we use an array of size n in which the v th entry is marked if v is live. After an iteration is finished, \mathcal{L} is updated. This approach allows us only to process live vertices.

4.2 Finding marginal gains with memoization

During the labeling stage in NEWGREEDYSTEP-VEC, INFUSER-MG computes and stores all the component labels l (obtained by concatenating each l_v for all $v \in V$) that can be considered as a two-dimensional $n \times \mathcal{R}$ array. The first seed vertex is indeed the one having the largest expected (average) component size. Instead of resampling, this information can be utilized during the CELF stage while computing marginal gains and finding the remaining $K - 1$ seed vertices. The marginal gain for a vertex u , i.e., mg_u , can be found by computing the average number of vertices (over all the \mathcal{R} samples) that belong to u 's connected component but do not belong to the components of the seed vertices. This is equal to the expected number of additional vertices that will be influenced by inserting u to the seed set S .

While computing mg_u , for all simulations, one can compare u 's label to all the component labels of the seed vertices in respective simulation. In our implementation, the data structure l is stored as a single large memory block where the \mathcal{R} labels of a single vertex are stored consecutively for a better spatial locality. The component

sizes are also stored in similar two-dimensional $n \times \mathcal{R}$ array where rows correspond to component labels and columns correspond to simulations. Labels that do not map to a component are wasted for fast access while keeping the asymptotic space complexity the same (as l 's space complexity). This process is equivalent to using `RANDCAS` over existing \mathcal{R} samples for finding marginal gains, except, no graph traversal or sampling is performed. Compared to the original approach, the memory accesses are more regular and the cache is better utilized. In our experiments, on the largest network `MixGreedy` can process, we observed up to 30% cache-miss rate for `MIXGREEDY`, whereas our implementations have only a 1.9% cache-miss rate. Furthermore, this operation can be efficiently parallelized as shown in the pseudo-code of `INFUSER-MG`, Algorithm 7 (lines 15–16).

Algorithm 7 `INFUSER-MG`(G, K, \mathcal{R})

Input: G : Graph

K : size of the seed set

\mathcal{R} : number of MC simulations to perform

Output: S : a seed set that maximizes influence

```

1:  $mg, l \leftarrow \text{NEWGREEDYSTEP-VEC}(G, \mathcal{R})$ 
2:  $S \leftarrow \{\emptyset\}$ 
3:  $q \leftarrow \text{PriorityQueue}()$ 
4:  $iter_v \leftarrow 0, \forall v \in V$ 
5: for  $v \in V$  do
6:    $q.enqueue(v, \text{priority}=mg_v)$ 
7:  $R_{G'}(v) \leftarrow 0, \forall v \in V$ 
8: while  $|S| < K$  do
9:    $u \leftarrow q.dequeue()$ 
10:  if  $iter_u = |S|$  then
11:     $R_{G'}(S) \leftarrow R_{G'}(S \cup \{u\})$  ▷ Append  $l_u$  to  $R_{G'}(S)$ 
12:     $S \leftarrow S \cup \{u\}$  ▷ Commit  $u$  into  $S$ 
13:  else
14:     $mg_u \leftarrow 0$ 
15:    for  $r = 1$  to  $\mathcal{R}$  in parallel reduce( $mg_u$ ) do
16:       $mg_u \leftarrow mg_u + |l_u[r] \in \{l \setminus R_{G'_r}(S)\}|$ 
17:     $iter_u \leftarrow |S|$ 
18:     $q.enqueue(u, \text{priority}=mg_u)$ 
19: return  $S$ 

```

4.3 Implementation Details

All the algorithms use the Compressed Sparse Row (CSR) graph data structure. In CSR, an array, $xadj$, holds the starting indices of each vertices neighbors, other vector, adj , holds neighbors of each vertex consecutively. So, to reach neighbors of vertex i , first we visit $xadj[i]$ and $xadj[i + 1]$ to find start and end positions in data, then scan starting from $adj[xadj[i]]$ until $adj[xadj[i + 1]]$ position. All sets including the live vertex sets \mathcal{L} and \mathcal{L}' are implemented as one-hot vectors. One-hot vectors are chosen since our method performs $\mathcal{O}(m)$ insert-if-not-exist operations, and iterates over a live set once per iteration.

4.4 Experimental Results

The experiments are performed on a server equipped with two AVX2-capable 8-core Intel Xeon E5-2620v4 sockets running on 2.10GHz and 192GB memory. Hence, there exist 16 cores in total. The OS running on the server is Ubuntu 16.04.2 LTS with Linux 4.4.0-66 generic kernel. The algorithms are implemented in C++ and compiled with GCC 8.2.0 with `-Ofast` as the optimization flag. To further test the performance of the proposed approach with wider SIMD vector units, we employ another architecture with AVX512-capable Intel Xeon Gold 6140 CPU running at 2.30GHz and 256 GB memory. On this architecture, GCC 9.2.0 is used to compile the code. Multi-threaded CPU parallelization is obtained with OpenMP pragmas. During a parallel execution, we employ dynamic scheduling with a batch size of 8192. We have manually utilized AVX instructions available on the CPUs by using compiler intrinsics to implement the algorithms.

For better readability, in all the tables, the values showing a better performance are given in bold.

4.4.1 Network datasets used in the experiments

The experiments are performed on twelve graphs (six undirected, six directed) that have been frequently used for Influence Maximization. For directed datasets, the reverse edges are added to obtain undirected variants. The datasets are Amazon co-purchase network (Leskovec & Krevl, 2014), DBLP co-laboration network (Leskovec

Table 4.1 Properties of networks used in the experiments

	Dataset	No. of Vertices	No. of Edges	Avg. Weight	Avg. Degree
Undirected	Amazon	262,113	1,234,878	1.00	4.71
	DBLP	317,081	1,049,867	1.00	3.31
	NetHEP	15,235	58,892	1.83	3.87
	NetPhy	37,151	231,508	1.28	6.23
	Orkut	3,072,441	117,185,083	1.00	38.14
	Youtube	1,134,891	2,987,625	1.00	2.63
Directed	Epinions	75,880	508,838	1.00	6.71
	LiveJournal	4,847,571	68,993,773	1.00	14.23
	Pokec	1,632,803	30,622,564	1.00	18.75
	Slashdot0811	77,360	905,468	1.00	11.70
	Slashdot0902	82,168	948,464	1.00	11.54
	Twitter	81,306	2,420,766	1.37	29.77

& Krevl, 2014), **Epinions** consumer review trust network, **LiveJournal** (Leskovec & Krevl, 2014), **NetHEP** citation network (Chen, Wang, & Yang, 2009), **NetPhy** citation network (Chen, Wang, & Yang, 2009), **Orkut** (Leskovec & Krevl, 2014), **Pokec** Slovakian poker game site friend network (Leskovec & Krevl, 2014), **Slashdot** friend-foe networks (08-11, 09-11) (Leskovec & Krevl, 2014), **Twitter** list co-occurrence network (Leskovec & Krevl, 2014), and **Youtube** friendship network (Leskovec & Krevl, 2014). The properties of these datasets are given in Table 4.1.

For a thorough experimental evaluation, four influence settings are simulated; for each network, we use

1. constant edge weights $p = 0.01$ (as in (Kempe, Kleinberg, & Tardos, 2003) and (Chen, Wang, & Yang, 2009)),
2. constant edge weights $p = 0.1$ (as in (Kempe, Kleinberg, & Tardos, 2003)),
3. uniformly distributed weights from the interval $[0, 0.1]$,
4. normally distributed weights with mean 0.05 and std. deviation 0.025 so that 95% of the weights lie in $[0, 0.1]$.

The last two settings are used to evaluate the performance in case of non-uniform edge weights.

4.4.2 Metrics used to evaluate the performance

Following the literature, we employ three metrics to evaluate an algorithm; (i) the influence score, i.e., the expected number of vertices that are influenced (ii) the

execution time, (iii) maximum memory size. There is an interplay among these metrics; it is trivial to devise an ultra-fast IM algorithm with a bad influence score. Similarly, using more memory can make an algorithm avoid computations. We present these metrics for each algorithm on all graphs.

When the algorithms run on the same machine, the reported execution times and memory usages of different algorithms are comparable. However, the reported influence scores can be misleading since the algorithms may be using different approaches to estimate the influence score. To find the expected number of vertices, we requested and used the original implementation from Chen, Wang, and Yang (2009) as an oracle with minor modifications; i.e., without logging and using heap memory instead of stack memory to handle large-scale graphs. The random values in the oracle are generated by C++’s Mersenne Twister 32-bit pseudo-random generator `mt19937`, with a state size of 19937 bits. For a precise evaluation, we have used $\mathcal{R} = 20000$ samples within this oracle.

4.4.3 Algorithms evaluated in the experiments

The algorithms that are evaluated can be classified into three groups. The first class contains MIXGREEDY, obtained from Chen et al. (Chen, Wang, & Yang, 2009), which is also used as the oracle to compute the influence scores. The second class contains two variants from the current state-of-the-art, Minutoli et al.’s IMM (Minutoli et al., 2019). IMM is a fast algorithm robustly producing high-quality seed sets which can influence a large number of vertices. In the original paper, the variant with $\epsilon = 0.13$, a user-defined hyper-parameter controlling the approximation boundaries, is suggested. We use this variant along with a much faster one with $\epsilon = 0.5$, which is also experimented in (Minutoli et al., 2019).

The third class of algorithms contains two INFUSER-MG variants. To show the speedup breakdown, we consider each variant as a separate algorithm. The first variant is FUSED SAMPLING which only integrates the sampling step by generating probabilities on the fly without any algorithmic improvements or edge traversal savings. This variant performs the simulations one-by-one as in MIXGREEDY. The second variant is the proposed approach INFUSER-MG employing vectorization and memoization. Both of these variants employ CELF and use the queue-based vertex processing as the base algorithm MIXGREEDY.

In this section, we first compare the INFUSER-MG variants with MIXGREEDY to

Table 4.2 Execution times (in secs), memory use (in GBs), and influence scores (higher is better) of the algorithms on the networks with $K = 50$ seeds and constant edge weights with $p = 0.01$.

Dataset	Execution time in seconds.				Memory use in Gigabytes.			Influence scores.		
	MIX	FUSED	INFUSER	INFUSER	MIX	FUSED	INFUSER	MIX	FUSED	INFUSER
	GREEDY ($\tau = 1$)	SAMPLING ($\tau = 1$)	MG ($\tau = 16$)	($K = 1$) ($\tau = 16$)	GREEDY ($\tau = 1$)	SAMPLING ($\tau = 1$)	MG ($\tau = 16$)	GREEDY ($\tau = 1$)	SAMPLING ($\tau = 1$)	MG ($\tau = 16$)
Amazon	141.31	48.84	2.09	2.09	0.76	0.18	4.05	158.28	158.63	158.63
DBLP	-	305.38	7.02	7.00	-	0.25	6.56	-	245.43	245.43
Epinions	-	157069.53	1.91	1.53	-	0.05	1.18	-	3051.39	3051.39
LiveJ.	-	-	265.84	218.28	-	-	75.38	-	-	260364.56
NetHEP	259.05	12.60	0.08	0.07	2.27	0.01	0.24	132.38	136.45	136.45
NetPhy	1725.15	247.21	0.36	0.34	8.56	0.03	0.58	312.56	332.52	332.52
Orkut	-	-	654.52	586.55	-	-	50.45	-	-	650237.06
Pokec	-	-	227.24	196.85	-	-	26.02	-	-	104196.34
Sdot0811	-	211783.43	2.69	2.00	-	0.07	1.21	-	5197.88	5197.88
Sdot0902	-	233822.30	3.11	2.17	-	0.08	1.29	-	5432.14	5432.14
Twitter	-	-	3.07	2.11	-	-	1.32	-	-	12441.56
Youtube	-	-	26.18	20.85	-	-	17.83	-	-	9139.01

present the speedups over the baseline with fusing and vectorization. We then compare INFUSER-MG, using $\mathcal{R} = 2048$ samples, with the state-of-the-art to better position the proposed approach in the literature. Last, we evaluate the multi-threading performance of INFUSER-MG with $\tau \in \{1, 2, 4, 8, 16\}$ threads. In all experiments, we use a time-limit of 302,400 seconds (3.5 days).

4.4.4 Comparing INFUSER-MG with MIXGREEDY

Table 4.2 shows the execution times (columns 2–5), memory usages (columns 6–8), and influence scores (columns 9–11) of the baseline algorithm and INFUSER-MG variants. MIXGREEDY runs with a single thread and finishes only in three graphs Amazon, NetHEP, and NetPhy in 141.3, 259.1 and 1725.2 seconds, respectively. In fact, with a 302,400 seconds (3.5 days) timeout, these are the only three (out of 12) real-life graphs (with 1.2M, 58.9K, and 231.5K edges) that can be processed by MIXGREEDY. For the others, the original algorithm cannot find a seed set of $K = 50$ vertices within the time limit. However, INFUSER-MG with $\tau = 16$ threads completes all the 12 graphs around 1200 seconds in total, where the maximum run-time is 654.5 seconds for the Orkut network having 3.1M vertices and 117.2M edges. The shortest execution time of INFUSER-MG on a graph that cannot be completed by MIXGREEDY is 1.5 seconds. Only by looking at the sequential execution times of FUSED SAMPLING on three graphs, we can conclude that $3 \times - 21 \times$ of this speedup comes from fusing.

The fifth column of Table 4.2 presents the execution times of INFUSER-MG to

find the first seed vertex which is simply Algorithm 7 where the **while** loop is executed only once, which is equivalent to the setting with $K = 1$. Comparing these values with the ones in the previous column, we can argue that the benefits of the memoization are more for large K values such as 500 or 1000, since most of the time is spent on the NEWGREEDYSTEP-VEC. For instance, for large graphs, adding the next 49 seeds only takes 10%–20% of the overall execution time. The actual value depends on the number of the CELF stage; for **Amazon**, to add the remaining seed vertices, INFUSER-MG needs only 79 vertex visits. This is why the cost of the CELF stage is negligible.

Although it is extremely useful, memoization is also the reason of high memory usage. The values for **NetHEP** and **NetPhy** are relatively lower compared to the baseline. However, these two graphs have only 15K and 37K vertices, much lower than the other graphs. In fact, FUSED SAMPLING can be a more efficient implementation of MIXGREEDY memory-wise. Comparing the memory use of FUSED SAMPLING with that of INFUSER-MG reveals the overhead of memoization more clearly. However, even with this overhead, the proposed approach stays practical and extremely efficient on a single server.

Overall, INFUSER-MG is a practical algorithm, and unlike MIXGREEDY, it can be used on *undirected* graphs that have been considered *too large* in the literature. On the comparable instances, it runs in 2.1, 0.1, 0.4 seconds where MIXGREEDY takes 141.3, 259.1, and 1725.2 seconds, respectively. Furthermore, as the last three columns of Table 4.2 show, the influence scores of the proposed approach are comparable with those of MIXGREEDY.

4.4.5 Comparing INFUSER-MG with State-of-the-Art

To better position INFUSER-MG within the literature, we compare the performance, memory usage, and influence score with a fast, state-of-the-art approximation algorithm IMM (Minutoli et al., 2019) which can produce high-quality seed sets that influences a large number of vertices for both directed and undirected graphs. We also run IMM by setting the undirected graph parameter. The methods based on Reverse Influence Sampling (such as IMM) control the approximation factor directly. Whereas greedy-based methods, the approximation factor is guided indirectly through the number of Monte-Carlo simulations \mathcal{R} (Sadeh, Cohen, & Kaplan, 2020). Due to this reason, we performed our experiments against two IMM settings; first tuned for speed ($\epsilon = 0.5$) and second tuned for quality ($\epsilon = 0.13$).

Table 4.3 Execution times (in secs) of the algorithms with $K = 50$ seeds and $\tau = 16$ threads in different simulation settings.

Dataset	$p = 0.01$		$p = 0.1$		$p \in N(0.05, 0.025)$		$p \in [0, 0.01]$		
	IMM ($\epsilon = 0.13$)	INFUSER MG ($\epsilon = 0.5$)	IMM ($\epsilon = 0.13$)	INFUSER MG ($\epsilon = 0.5$)	IMM ($\epsilon = 0.13$)	INFUSER MG ($\epsilon = 0.5$)	IMM ($\epsilon = 0.13$)	INFUSER MG ($\epsilon = 0.5$)	
Amazon	62.67	4.95	24.80	2.72	8.64	0.84	8.15	1.29	3.56
DBLP	55.92	4.02	168.68	15.34	46.90	4.97	56.34	5.02	12.66
Epinions	72.39	7.55	86.10	7.82	92.28	9.58	91.68	9.08	1.10
LiveJournal	9078.34	860.38	-	1527.58	-	1678.81	-	1732.58	214.98
NetHEP	2.80	0.29	6.31	0.65	4.33	0.43	4.41	0.42	0.19
NetPhy	3.55	0.39	22.57	2.06	18.07	1.64	16.55	1.69	0.69
Slashdot0811	135.54	12.33	146.09	14.48	166.84	16.08	160.03	17.18	1.57
Slashdot0902	107.83	10.63	129.15	13.29	151.31	13.59	145.54	14.74	1.56
Orkut	24300.59	2279.10	-	1987.11	-	2541.79	-	2642.77	225.02
Pokec	2646.98	247.36	-	611.36	8060.71	796.88	8477.91	735.35	96.11
Twitter	298.97	26.70	261.94	23.70	321.48	30.10	310.16	30.44	1.91
Youtube	201.65	19.42	740.35	78.51	643.09	61.08	649.30	61.80	25.18

Table 4.4 Memory use (in GBs) of the algorithms on the networks with $K = 50$ seeds in different simulation settings.

Dataset	$p = 0.01$			$p = 0.1$			$p \in N(0.05, 0.025)$			$p \in [0, 0.01]$		
	IMM ($\epsilon = 0.13$)	IMM ($\epsilon = 0.5$)	INFUSER MG	IMM ($\epsilon = 0.13$)	IMM ($\epsilon = 0.5$)	INFUSER MG	IMM ($\epsilon = 0.13$)	IMM ($\epsilon = 0.5$)	INFUSER MG	IMM ($\epsilon = 0.13$)	IMM ($\epsilon = 0.5$)	INFUSER MG
Amazon	5.46	0.55	4.05	1.76	0.24	4.05	0.82	0.16	4.06	0.82	0.16	4.06
DBLP	5.12	0.51	6.56	10.34	1.04	6.56	2.14	0.28	6.57	2.32	0.28	6.57
Epinions	0.78	0.10	1.18	3.88	0.39	1.18	2.53	0.26	1.19	2.52	0.27	1.19
LiveJ.	71.14	9.27	75.38	-	67.97	75.38	-	47.35	75.38	-	47.22	75.38
NetHEP	0.26	0.03	0.24	0.36	0.04	0.24	0.16	0.02	0.24	0.15	0.02	0.24
NetPhy	0.30	0.05	0.58	1.18	0.13	0.58	0.61	0.07	0.58	0.61	0.07	0.58
Sdot0811	1.17	0.15	1.21	6.32	0.65	1.21	4.30	0.45	1.22	4.28	0.45	1.22
Sdot0902	1.22	0.16	1.29	6.67	0.69	1.29	4.53	0.47	1.30	4.50	0.47	1.30
Orkut	172.53	20.11	50.45	-	71.97	50.45	-	62.93	50.45	-	62.34	50.45
Pokec	26.61	3.55	26.02	-	27.13	26.02	185.55	21.27	26.22	185.54	21.02	26.22
Twitter	2.43	0.31	1.32	10.66	1.11	1.32	8.20	0.85	1.34	8.18	0.85	1.34
Youtube	2.68	0.48	17.83	41.29	4.17	17.83	21.07	2.24	17.85	20.88	2.24	17.85

Tables 4.3 and 4.4 show the execution times (in secs.) and memory use (in GBs), respectively, of INFUSER-MG and two IMM variants, utilizing 16 threads, for 12 graphs and 4 simulation settings given in Section 4.4.1. The experiments show that INFUSER-MG is $2.3\times$ – $173.8\times$ faster than state-of-the-art while always being (marginally) superior in terms of influence scores, and using a comparable amount of memory. As expected, the memory usage of IMM is increasing with smaller ϵ values. In addition, it also increases when the edge weights are larger, i.e., when the samples are denser. For instance, with $p = 0.01$, IMM($\epsilon = 0.5$) uses 20GBs for Orkut. However, when $p = 0.1$, the memory usage increases to 72GBs. Furthermore, IMM($\epsilon = 0.13$) cannot run on LiveJournal, Orkut, and Pokec networks due to insufficient memory. On the other hand, INFUSER-MG’s memory usage does not change with different values since it never explicitly creates and stores the samples thanks to fusing. Last, as shown in Table 4.5, the influence scores of the proposed approach and IMM($\epsilon = 0.13$) are comparable. Figure 4.3 shows the speedup values of INFUSER-MG with respect to IMM($\epsilon = 0.13$).

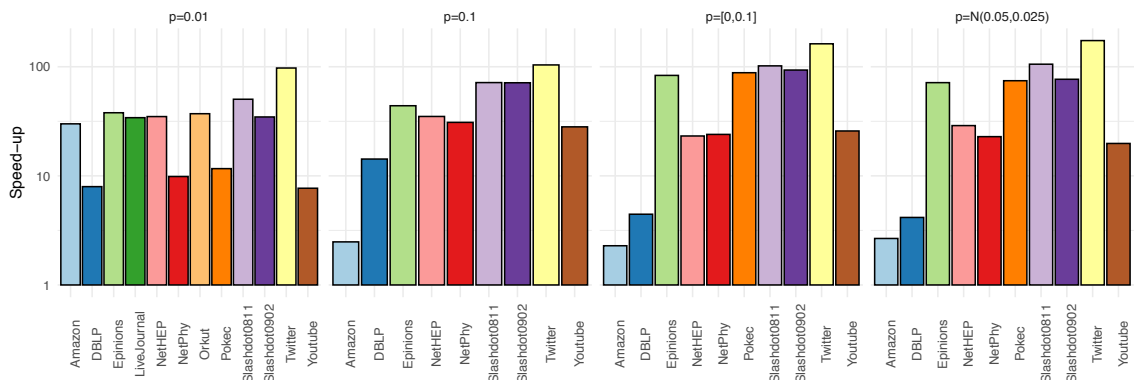


Figure 4.3 Speedup obtained by INFUSER-MG($\mathcal{R} = 2048, \tau = 16$) over IMM($\epsilon = 0.13, \tau = 16$).

4.4.6 Scalability with multi-threaded parallelism

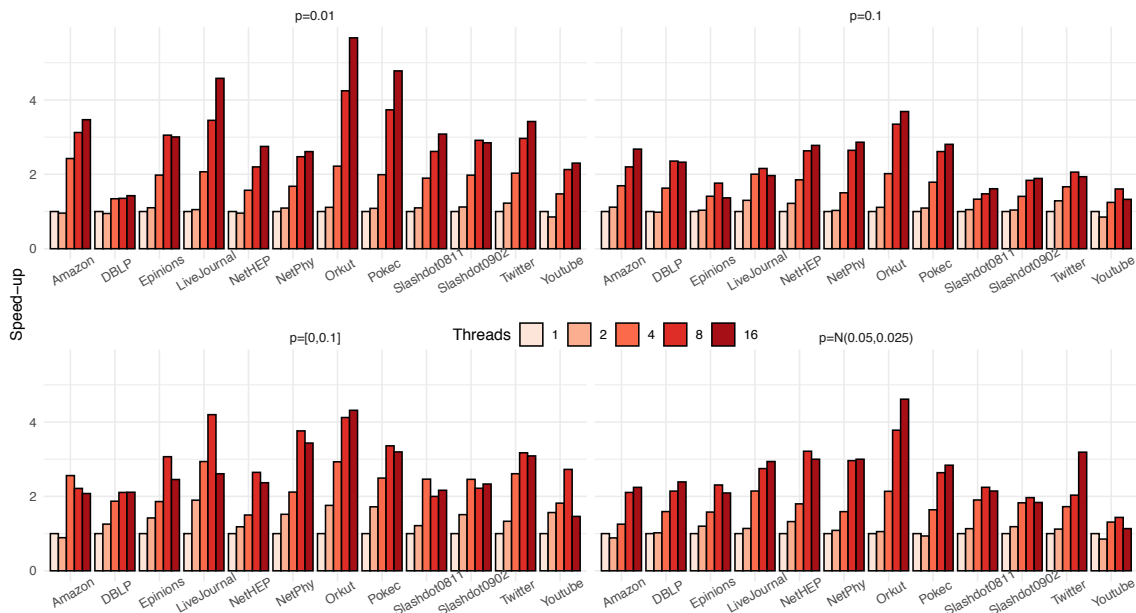


Figure 4.4 INFUSER-MG speedup with multiple threads.

Figure 4.4 shows the speedup values obtained via `OpenMP` parallelization. Since most of the time is spent by `NEWGREEDYSTEP-VEC`, the parallelization efficiency at line 6 of Algorithm 5 has a significant impact on the performance. In our implementation, the parallel processing of live (as source) vertices is necessary to reduce the number of visited edges. However, since a (target) vertex can be a target for multiple sources, the update operation at line 14 of this *push-based* approach is a potential source of race conditions. For denser samples, e.g., for $p = 0.1$, this happens more frequently. Hence, larger influence probabilities may increase (1) the false sharing probability and (2) the number of iterations due to vectorized updates. We argue that these are the reasons for $3 \times - 5 \times$ speedup with $\tau = 16$ threads. We observed significant increase in cache misses, up-to $2.7 \times$, with $\tau = 16$ threads. Even on relatively small networks, like the Amazon network, we observed the cache-miss rate to increase from 1.9% to 5.0%. In addition, we believe that the performance can further be improved by adopting optimizations on storage (Shun & Blelloch, 2013; Lim, Kang, & Faloutsos, 2014), parallel processing (Shun & Blelloch, 2013), and vertex reordering (Arai et al., 2016; Wei et al., 2016).

4.4.6.1 Performance with Wider AVX Registers

Table 4.6 Execution times (in secs) on the AVX512-capable architecture, networks with $K = 50$ seeds, using $\mathcal{R} = 2048$ for INFUSER-MG and constant edge weights with $p \in \{0.01, 0.1\}$.

τ	Dataset	p	NoAVX	AVX2	AVX512	IMM($\epsilon = 0.5$)
1	LiveJournal	0.01	3,176.61	685.80	564.93	1,918.22
		0.10	530.20	267.86	247.69	8,406.32
	Orkut	0.01	12,188.35	1,808.84	1,427.10	5,898.02
		0.10	1,234.59	473.87	387.04	11,661.69
	Pokec	0.01	3,178.43	501.16	388.53	529.84
		0.10	346.16	137.68	116.63	3,460.62
8	LiveJournal	0.01	519.80	195.76	175.91	1,398.29
		0.10	169.67	139.62	131.90	2,819.03
	Orkut	0.01	1,726.03	324.59	281.84	3,167.05
		0.10	234.99	122.40	114.12	3,030.86
	Pokec	0.01	477.02	116.24	113.19	330.65
		0.10	91.83	51.42	61.01	1,022.52
16	LiveJournal	0.01	333.60	157.25	157.06	742.78
		0.10	165.31	143.36	142.04	1,548.61
	Orkut	0.01	878.21	211.75	192.23	2,008.35
		0.10	166.70	105.08	106.80	1,811.96
	Pokec	0.01	263.12	78.79	83.04	278.60
		0.10	76.37	53.52	56.58	655.06

In Table 4.6, we show the performance of the proposed algorithm on an AVX512-capable architecture and on large-scale graphs with more than 10^6 edges. Even though frequency scaling is reducing the performance of AVX512 in higher thread counts, the proposed method is shown to leverage wider SIMD registers. While running with a single thread, the architecture has a 5% clock difference between normal and the AVX mode. Utilizing AVX512 instructions with 16 threads drops the clock frequency up-to 45%.

5. FAST AND ERROR-ADAPTIVE INFLUENCE

MAXIMIZATION BASED ON COUNT-DISTINCT SKETCHES

Most IM algorithms have the same few steps to find the best seed vertex set; sampling, building the influence oracle, verifying the impact of new candidates, and removing the latent seed set’s residual reachability set. Following the idea proposed in (Göktürk & Kaya, 2021), HYPERFUSER fuses the sampling step with other steps to avoid reading the graph multiple times.

HYPERFUSER first performs a diffusion process; the process starts with per-vertex sketches that are initialized with the hash value of the corresponding vertex (i.e., every vertex reaches to itself). Then, for all the (sampled) edges (u, v) , i.e., the ones that contribute to the diffusion for this simulation, the sketch of the source vertex u is merged into that of the target vertex v until all sketch registers for all the vertices converge. The merge operation utilized in this process is slightly different from the conventional one and retrofitted to mimic the IC diffusion.

Throughout the process, each register is used only for a single sample/simulation. For an edge (u, v) in simulation j where v is a live vertex, an update, i.e., a merge operation, on u ’s register is performed on the corresponding register $M_u[j]$, i.e., $M_u[j] = \max(M_u[j], M_v[j])$. That is, at each iteration, vertices (outgoing) neighbors’ reachability sets are added to their sketches. This recursive formulation of the influence iteratively relays the reachability information among the vertices, allowing us to estimate the marginal influence for all vertices very fast.

After estimating the reachability set cardinalities, HYPERFUSER picks the vertex v with the largest cardinality by evaluating the sketches. Then it finds the (actual) reachability set of the latent seed set, which is the union of the reachability sets of v and the vertices in the seed set, by performing Monte-Carlo simulations. The vertices in this reachability set are removed from the live set L . Hence, in later iterations, these vertices will not contribute to the marginal gain. Finally, the algorithm checks if rebuilding is necessary for the sketches based on the difference between the sketch estimate and Monte-Carlo estimate.

5.1 Count-Distinct Sketches

The *distinct element count* problem focuses on finding the number of distinct elements in a stream where the elements are coming from a universal set \mathcal{U} . Finding the number of vertices to be influenced of a candidate seed vertex u , i.e., the cardinality of u 's *reachability set*, is a similar problem. For each sample subgraph, the number of visited vertices is found while traversing the subgraphs starting from u . Note that an exact computation of set cardinality requires memory proportional to the cardinality, which is $\mathcal{O}(|\mathcal{U}|)$.

The reachability set of a vertex is the union of all its connected vertices (via outgoing edges). Many IM kernels exploit this property to some degree. The methods based on *reverse reachability* (Borgs et al., 2014) utilize this property directly to merge the reachability sets of connected vertices to estimate the number of vertices influenced. *MixGreedy* (Chen, Wang, & Yang, 2009) goes one step further; it utilizes the fact that for an undirected graph, all vertices in a connected component have the same reachability set. Therefore, all the reachability sets within a single sample subgraph can be found via a single graph traversal.

For directed graphs, storing reachability sets for all vertices and merging these sets are infeasible for nontrivial graphs. If one-hot vectors are used to store the reachability sets for constant insertion time, $\mathcal{O}(n^2\mathcal{R})$ bits of memory is required where each merge operation has $\mathcal{O}(n)$ time complexity. If disjoint sets are used for storing reachability sets; $\mathcal{O}(n\sigma\mathcal{R})$ memory is required to store all reachability sets, and each merge operation has $\mathcal{O}(\text{Ack}(\sigma))$ complexity where Ack is the Ackermann function (Ackermann, 1928).

Count-Distinct Sketches can be leveraged to estimate reachability sets' cardinality efficiently; for instance, the Flajolet–Martin (FM) sketch (Flajolet & Martin, 1985) can do this with a constant number, \mathcal{J} , of registers. Furthermore, the union of two sketches can be computed in constant time. The FM sketch stores that how rare the elements are in a stream. The rarity of the elements is estimated by counting the maximum number of leading zeros in the stream elements' hash values. Initially, each register is initialized with zero. The items are hashed one by one, and the length of the longest all-zero prefix is stored in the register. With a single register, the cardinality estimation can be done by computing the power 2^ℓ where ℓ is the value in the register.

In practice, multiple registers and hash values, $M[j]$ and h_j , are commonly used to reduce variance. For a sketch with multiple registers, the impact of adding an item

$x \in \mathcal{U}$ is shown in (5.1):

$$(5.1) \quad M[j] = \max(M[j], clz(h_j(x))), \quad 1 \leq j \leq \mathcal{J}$$

where $clz(y)$ returns the number of leading zeros in y and \mathcal{J} is the number of sketch registers. With multiple registers, the average of the register values can be used to estimate the cardinality, and the result is divided to a correction factor $\phi \approx 0.77351$ to fix the error due to hash collisions. That is the estimated cardinality e is computed as

$$(5.2) \quad e = 2^{\bar{M}} / \phi$$

where $\bar{M} = \text{avg}_j\{M[j]\}$ is the mean of the register values.

In this work, we utilize a variant of Flajolet–Martin sketch; since multiple Monte-Carlo simulations are performed to calculate the estimated influence, we use one register per simulation and take the average length of the longest leading zeros. Two given FM sketches M_u and M_v can be merged, i.e., their union M_{uv} can be computed by taking the pairwise maximums of their registers. Formally;

$$(5.3) \quad M_{uv}[j] = \max(M_u[j], M_v[j]), \quad 1 \leq j \leq \mathcal{J}.$$

In our implementation, the merge operations are performed if and only if there is a sampled edge between the vertices.

5.2 Estimating Reachability Set Cardinality

A greedy solution to the influence maximization problem requires finding a vertex that maximizes the marginal influence gain at each step until the seed set size reaches K . For an exact computation, one must find all candidate vertices’ reachability sets within all the samples. Such a task involves many graph traversals and is expensive even with various algorithmic optimizations and a scalable parallelized implementation, e.g., see (Göktürk & Kaya, 2021). The influence estimation problem is quite similar to the Count-Distinct problem applied to all sample subgraphs, as explained above. Hence, in this work, we pursue the idea of using Count-Distinct sketches to estimate marginal influence scores. In this work, we propose an efficient and effective IM kernel, HYPERFUSER, that utilizes Flajolet–Martin sketches described in

Section 5.1 to estimate the averages of distinct elements in the sampled subgraphs. Algorithm 8 shows the steps taken by the kernel.

Algorithm 8 HYPERFUSER(G, K, \mathcal{J})

Input: $G = (V, E)$: the influence graph

K : number of seed vertices

\mathcal{J} : number of Monte-Carlo simulations

Output: S : a seed set that maximizes the influence on G

```

1:  $S \leftarrow \emptyset$ 
2: for  $v \in V$  do in parallel
3:   for  $j \in \{1, \dots, \mathcal{J}\}$  do
4:      $M_v[j] \leftarrow \text{clz}(\text{hash}_j(v))$ 
5:  $M \leftarrow \text{SIMULATE}(G, M, \mathcal{J}, \emptyset)$ 
6:  $M_{S'} \leftarrow \text{zeros}(\mathcal{J})$ 
7:  $\varsigma \leftarrow 0$ 
8: for  $k = 1 \dots K$  do
9:    $s \leftarrow \underset{v \in V}{\text{argmax}}\{\text{ESTIMATE}(\text{MERGE}(M_{S'}, M_v))\}$ 
10:   $S \leftarrow S \cup \{s\}$ 
11:   $e \leftarrow \text{ESTIMATE}(\text{MERGE}(M_{S'}, M_s))$ 
12:   $R_G(S) \leftarrow$  reachability set of  $S$  (for all simulations)
13:   $\sigma \leftarrow$  Monte-Carlo-based (actual) influence of  $S$ 
14:   $\delta = \sigma - \varsigma$ 
15:   $\text{err}_l = |(e - \delta)/\delta|$ 
16:   $\text{err}_g = |(e - \delta)/\sigma|$ 
17:  if  $\text{err}_l < \epsilon_l \vee \text{err}_g < \epsilon_g$  then
18:     $M_{S'} \leftarrow \text{MERGE}(M_{S'}, M_s)$ 
19:  else
20:    for  $v \in V$  do in parallel
21:      for  $j \in \{1, \dots, \mathcal{J}\}$  do
22:         $M_v[j] \leftarrow \text{clz}(\text{hash}_j(v))$ 
23:       $M \leftarrow \text{SIMULATE}(G, M, \mathcal{J}, R_G(S))$ 
24:       $M_{S'} \leftarrow \text{zeros}(\mathcal{J})$ 
25:       $\varsigma \leftarrow \sigma$ 
26: return  $S$ 

```

Algorithm 8 first initializes the reachability sets of all vertices by adding the vertices themselves. That is for all vertices u , its j th register is set to $M_u[j] = \text{clz}(h_j(u))$ meaning $R_{G_j}(u) = \{u\}$ where G_j is the j th sampled graph. Then, we perform the diffusion process on the sketch registers whose pseudocode is given in Algorithm 9. The diffusion starts by adding all the vertices to the *live vertex set* L . Then at each step, the incoming edges of the live vertices are processed. For a vertex u , its sketch, M_u , is updated by merging the sketches M_v of all live outgoing neighbors vertices $v \in L \cap \Gamma_G^+(u)$. For each such vertex v and simulation j , the operation $M_u[j] = \max(M_u[j], M_v[j])$ is performed. This approach can be seen as a bottom-up, i.e., reversed, diffusion process where at each iteration, the cardinality information

is pulled from vertices neighbors. If any of u 's sketch registers changes during this operation, it is added to the live vertex set L' of the next iteration. Once the incoming edges of all live vertices are processed, the iteration ends. Figure 5.1 shows how HYPERFUSER performs two simulations at the same time using sketch registers.

Algorithm 9 SIMULATE(G, M, \mathcal{J}, R_S)

Input: $G = (V, E)$: the influence graph

M : sketch vectors of vertices

\mathcal{J} : number of MC simulations

R_S : reachability set of the seed set

Output: M : updated Sketch vectors

```

1:  $L \leftarrow V$ 
2:  $L' \leftarrow \emptyset$ 
3: while  $|L|/|V| > \epsilon_c$  do
4:   for  $u \in \Gamma(L)$  do in parallel
5:     for  $e_{u,v} \in A(u)$  do
6:       for  $j \in (0, \mathcal{J}]$  do
7:         if  $P(u,v)_j < w_{u,v} \wedge u \notin R_S[j]$  then
8:            $M_u[j] \leftarrow \max(M_u[j], M_v[j])$ 
9:         if  $M_u$  changed then
10:           $L' \leftarrow L' \cup u$ 
11:    $L \leftarrow L'$ 
12:    $L' \leftarrow \emptyset$ 
13: return  $M$ 

```

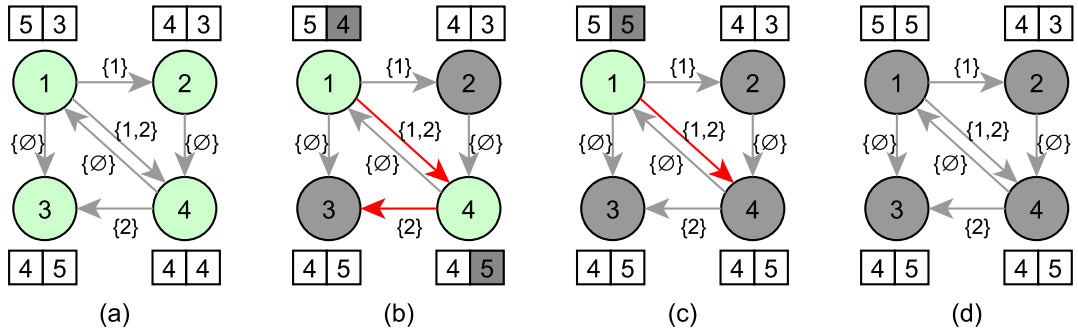


Figure 5.1 (a) The initial state on the toy graph for HYPERFUSER; all vertices are set as live (green), and their registers are initialized with the length of the zero prefix of their hashes. (b) For the $(1,4)$ edge which is live for both simulations, 1's registers are set to the maximum of both 1's and 4's registers. The $(4,3)$ edge is live only in the second simulation. Hence, the second register of 4 is updated to 5. For the second iteration, vertices 1 and 4 are live (green) since their registers have changed. (c) For the live $(1,4)$ edge, 1's second is updated and 1 is set as live again. (d) All the registers converged. As no live vertices exist, the process stops.

The traditional Greedy algorithm (Kempe, Kleinberg, & Tardos, 2003) processes the

simulations one-by-one and computes the vertices’ reachability sets. On the other hand, HYPERFUSER efficiently performs multiple simulations at once in a single-step iteration. Since each iteration relays one level of cardinality information, this step requires at most d iterations where d is the diameter of G . When processed individually as the Greedy algorithm does, the j th simulation over the sampled subgraph G_j would require only at most d_j iterations, which is the diameter of G_j , and probably much smaller than d . Although HYPERFUSER seems to perform much more iterations, d is a loose upper bound for HYPERFUSER. A better one is $\max\{d_j : 1 \leq j \leq \mathcal{J}\}$ where d_j is the diameter of G_j . To further reduce the overhead of concurrent simulations and avoid bottleneck simulations due to remaining perimeter vertices, we employ an early-exit threshold ϵ_c over the remaining live vertices ratio, which is expected to be very small when only one or two simulations remain. That is if $|L'| \leq |V| \times \epsilon_c$ the diffusion process in Algorithm 9 stops. Otherwise, L is set to L' , L' is cleared, and the next iteration starts. We used $\epsilon_c = 0.02$ to make HYPERFUSER faster while keeping its quality almost the same.

After the diffusion process, the following steps are repeated until K vertices are added to the seed set S . First, for each $v \in V$, the cardinality of the reachability set, $R_G(S \cup v)$, is estimated by merging $M_{S'}$ and v ’s sketch registers where $M_{S'}$ is the set of sketch registers for the seed set S used to estimate the number of already influenced vertices by S .¹ Before the kernel, these registers are initialized with zeros. Second, a vertex s with the maximum cardinality estimation is selected and added to S . Third, the actual simulations are performed to compute the reachability set of S . Having an actual $R_G(S)$ allows us to calculate the estimation errors and find the blocked vertices for all simulations, which is vital since these blocked vertices can be skipped during the next diffusion steps. Besides, we leverage the actual influence to have an *error-adaptive kernel*, i.e., to compute the actual sketch error and rebuild the sketches when the accumulated error reaches a critical level which can deteriorate the quality for the following seed vertex selections.

5.3 Error-adaptive sketch rebuilding

¹In fact, the definition is exact only if sketch rebuilding is disabled. As it will be described in the following subsection, when HYPERFUSER’s error-adaptive mechanism is enabled, $M_{S'}$ is periodically rebuilt to estimate the cardinality of reachability sets over the remaining, unblocked vertices. This is why S' is used instead of S .

Sketches are fast. However, each sketch operation, including update and merge, can decrease their estimation quality below a desired threshold. Our preliminary experiments revealed that sketches are highly competent at finding the first few seed vertices for influence maximization. Unfortunately, after a few seed vertices, the sketch registers $M_{S'}$, which are updated at line 18 of Algorithm 8 via merging with new seed vertex s 's reachability set, become large. The saturation of $M_{S'}$ registers is important since HYPERFUSER uses them to select the best seed candidate at line 9. When they lose their sensitivity for seed selection, a significant drop in the quality is observed.

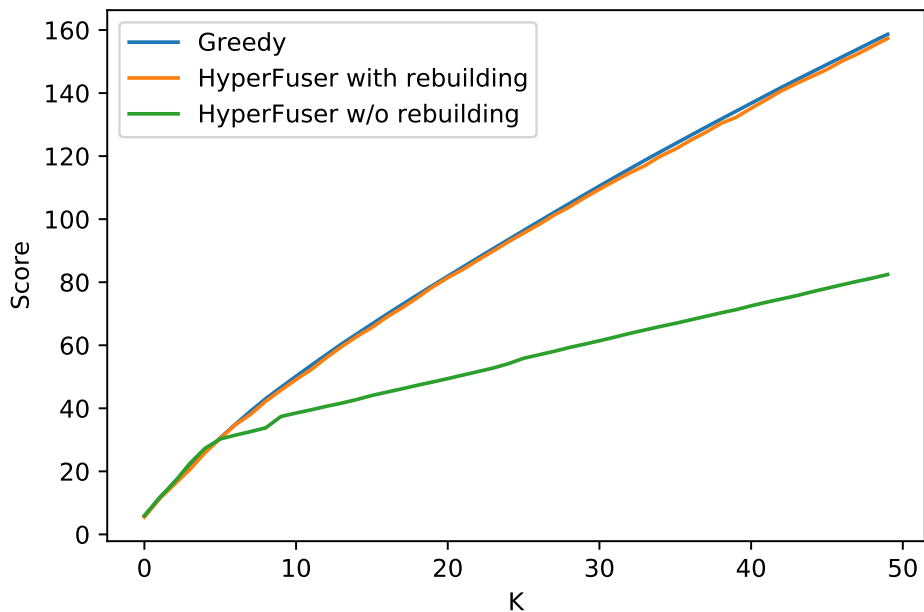
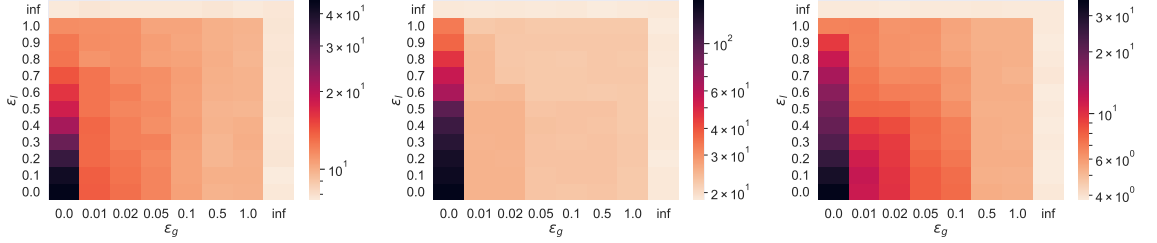
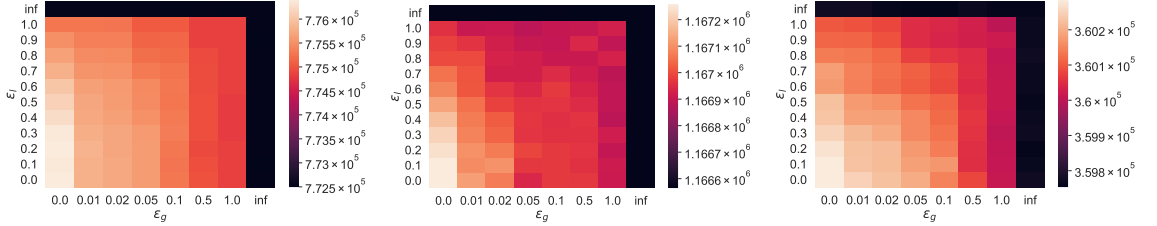


Figure 5.2 Effect of register saturation on Amazon dataset ($w = 0.01$) using HYPERFUSER ($\mathcal{J} = 256$) without rebuilding against Greedy ($\mathcal{R} = 20000$) method (Kempe, Kleinberg, & Tardos, 2003).

Figure 5.2 shows the effect of register saturation by comparing two HYPERFUSER variants; the first one rebuilds a new sketch to choose each seed vertex s , i.e., the **else** part in lines 20–25 of Algorithm 8 is executed for every iteration of the **for** loop at line 8. This sketch is built on the residual graph $G \setminus R_G(S)$, which remains after the current seed set’s reachability is removed. The second variant builds a sketch only once at the beginning and employs it through the IM kernel, i.e., the **else** part is never executed. The figure shows that the latter’s seed selection quality is comparable to that of the former for the first few seed vertices. However, a significant reduction in the quality is observed for the later vertices. Furthermore, the former approach’s quality is on par with the expensive Greedy algorithm’s quality, which computes actual reachability sets. This shows that sketch-based estimation can



(a) LiveJournal execution time in seconds (b) Orkut execution time in seconds (c) Pokec execution time in seconds



(d) LiveJournal influence score (e) Orkut influence score (f) Pokec influence score

Figure 5.3 Effect of ϵ parameters on HYPERFUSER ($\mathcal{J} = 256$, $\epsilon_c = 0.02$) performance, using $\tau = 18$ threads. Lighter shades are better.

perform as well as the accurate but expensive approach. Note that rebuilding also allows HYPERFUSER to work on a smaller problem for the following seed vertex selection since we remove the already influenced vertices from sample subgraphs and work on the remaining subgraphs.

Although its quality is on par with the traditional algorithm, the variant which rebuilds a sketch for all the seed vertex selections can be expensive. Here, we leverage an error-adaptive approach by rebuilding them when a significant cardinality estimation error is observed. The estimation error is calculated as follows; we store the influence score after each sketch rebuild in ς (line 25 of Algorithm 8). Let σ be the real influence for the seed set S including the selected vertex. We first compute the marginal influence gain $\delta = \sigma - \varsigma$, which is the additional influence obtained since the last sketch rebuilt. Note that e , computed at line 11 is the sketch estimate for this value. HYPERFUSER computes the local estimation error $err_l = |(e - \delta)/\delta|$ and the global error threshold $err_g = |(e - \delta)/\sigma|$. The sketches are assumed to be fresh if the local estimation error err_l is smaller than a local threshold ϵ_l or the global error threshold $err_g = |(e - \delta)/\sigma|$ is smaller than a global threshold ϵ_g .

The use of two different, local and global, thresholds allows the algorithm to rebuild the sketches after significant local errors and skip this expensive process if the estimation error is insignificant compared to the total influence. As explained above, when the rebuilding is skipped, HYPERFUSER only updates $M_{G'}$ by merging it with

the candidate vertex’s sketch. Hence, the selected threshold values, ϵ_l and ϵ_g , have a significant impact on the performance. Setting $\epsilon_l = \epsilon_g = 0$ means that the algorithm always rebuilds. Conversely, setting $\epsilon_l = \epsilon_g = \infty$ will make HYPERFUSER fast since sketches are built only once. However, the influence scores will suffer, which is already shown by Fig. 5.2. To evaluate the interplay and find the thresholds that yield a nice tradeoff, we conducted a grid search in which HYPERFUSER’s execution time and influence quality are measured for different parameters. The results of this preliminary experiment are shown in Figure 5.3. We found that the parameters $\epsilon_l = 0.3$ and $\epsilon_g = 0.01$ perform well on many datasets, both in terms of speed and quality.

5.4 Implementation Details

To efficiently process real-life graphs, HYPERFUSER uses the Compressed Sparse Row (CSR) graph data structure. In CSR, an array, $xadj$, holds the starting indices of the adjacency lists for each vertex, while another array, adj , holds the actual adjacency lists (i.e., the outgoing neighbors) one after another. Hence, the adjacency list of vertex i is located in adj at locations $adj[index[i], \dots, index[i + 1] - 1]$.

The traditional two-step (sample-then-diffuse) computation model stores the (graph) data in a loosely coupled fashion. While designing HYPERFUSER, we fine-tuned it to be vectorization friendly, including its data layout and computation patterns. These design choices allow us to perform multiple operations, i.e., the same operations but on different data, at once. For instance, we keep all the memory registers of a single vertex from different simulations adjacent, and this allows the efficient use of vectorized computation hardware while performing lines 6–8 of Algorithm 9. Also, random number generation, fused sampling, and sketch merging are vectorizable operations when the data are stored in a coupled way as in HYPERFUSER.

5.5 Experimental Results

We performed the experiments on a server with an 18-core Intel Xeon Gold 6140, running at 2.3Ghz, and 250GB memory. The Operating System on the server is

Ubuntu 16.04 LTS with 5.4.0-48 kernel. The algorithms are implemented using C++20, and compiled with GCC 9.2.0 with "-Ofast" and "-march=native" optimization flags. Multi-thread parallelization was achieved with OpenMP pragmas. AVX2 instructions are utilized by handcrafted code with vector intrinsics.

Table 5.1 Properties of networks used in the experiments

	Dataset	No. of Vertices	No. of Edges	Avg. Weight	Avg. Degree
Undirected	Amazon	262,113	1,234,878	1.00	4.71
	DBLP	317,081	1,049,867	1.00	3.31
	NetHEP	15,235	58,892	1.83	3.87
	NetPhy	37,151	231,508	1.28	6.23
	Orkut	3,072,441	117,185,083	1.00	38.14
	Youtube	1,134,891	2,987,625	1.00	2.63
Directed	Epinions	75,880	508,838	1.00	6.71
	LiveJournal	4,847,571	68,993,773	1.00	14.23
	Pokec	1,632,803	30,622,564	1.00	18.75
	Slashdot0811	77,360	905,468	1.00	11.70
	Slashdot0902	82,168	948,464	1.00	11.54
	Twitter	81,306	2,420,766	1.37	29.77

5.5.1 Experiment Settings

We performed the experiments on twelve graphs (six undirected, six directed). For comparability, graphs that have been frequently used within the Influence Maximization literature are selected. These graphs are **Amazon** co-purchase network (Leskovec & Krevl, 2014), **DBLP** co-laboration network (Leskovec & Krevl, 2014), **Epinions** consumer review trust network, **LiveJournal** (Leskovec & Krevl, 2014), **NetHEP** citation network (Chen, Wang, & Yang, 2009), **NetPhy** citation network (Chen, Wang, & Yang, 2009), **Orkut** (Leskovec & Krevl, 2014), **Pokec** Slovakian poker game site friend network (Leskovec & Krevl, 2014), **Slashdot** friend-foe networks (08-11, 09-11) (Leskovec & Krevl, 2014), **Twitter** list co-occurrence network (Leskovec & Krevl, 2014), and **Youtube** friendship network (Leskovec & Krevl, 2014). The properties of these graphs are given in Table 5.1.

Three diffusion settings are simulated for a comprehensive experimental evaluation; for each network, we use

1. constant edge weights $w = 0.005$,
2. constant edge weights $w = 0.01$ (as in (Kempe, Kleinberg, & Tardos, 2003) and (Chen, Wang, & Yang, 2009)),

3. constant edge weights $w = 0.1$ (as in (Kempe, Kleinberg, & Tardos, 2003)),

We selected $w = 0.005$ as a benchmark-setting to challenge HYPERFUSER. Due to its diffusion algorithm’s nature, HYPERFUSER traverses vertices even if they are blocked, which happens faster when the graph is sparser. Also, for each live vertex, HYPERFUSER processes the sample edges for all simulations. The other two settings are selected to emulate the experiments of Kempe, Kleinberg, and Tardos (2003) and Chen, Wang, and Yang (2009).

5.5.2 Performance Metrics

The algorithms are evaluated based on (1) execution time, (2) influence score, and (2) maximum memory used. For Influence Maximization, there is a trade-off among these performance metrics; in one extreme, it is trivial to select random vertices as the seed set. In another, one can compute the reachability sets of every possible seed set of size K and choose the best one. In all our experiments, the execution times are the wall times reported by the programs. All the methods we benchmarked exclude the time spent on reading files and preprocessing. We only left out the time to spend on reading files for HYPERFUSER. We allowed all methods to utilize all the CPU cores in all benchmarks, except TIM+, a single-threaded algorithm. The memory use reported in this paper is the *maximum resident set sizes* (RSS), which are measured using GNU `time` command.

Since the algorithms may use different methods to measure the influence, the reported influence scores may not be suitable for comparison purposes with high precision. Due to this reason, we implemented an oracle with a straightforward, sample-then-diffuse algorithm without any optimization mentioned. For sampling, the random values are generated by the 32-bit Mersenne Twister pseudo-random generator `mt19937` from C++ standard library. The same independent oracle obtains all influence scores in this paper.

5.5.3 Algorithms evaluated in the experiments

We evaluated our method against three other state-of-the-art influence maximization algorithms, TIM+, SKIM, and IMM. The first algorithm focuses on the influence score, whereas the second is a sketch-based algorithm that takes the execution time

into account. The third one is an approximation algorithm with a parameter to control the influence quality.

- The Two-phased Influence Maximization (TIM+) runs in two phases: *Parameter Estimation* which estimates the maximum expected influence and a parameter θ and *Node Selection* which randomly samples θ reverse reachability sets from G and then derives a size- K vertex-set S that covers a large number of these sets (Tang, Xiao, & Shi, 2014). The algorithm has a parameter ϵ which allows a trade-off between the seed set quality and execution time. In our experiments, we set $\epsilon = 0.3$ to have a high-quality influence maximization baseline. We also experimented with $\epsilon = 1.0$ as suggested, which gives around $7\times$ speedup on average but a reduction on the influence score up to 6%.
- The Sketch-based Influence Maximization (SKIM) uses a combined bottom- k min-hash reachability sketch (Cohen & Kaplan, 2007) to estimate the influence scores of the seed sets (Cohen et al., 2014). As suggested by the authors, in this work, we employ SKIM with $k = 64$ and $\ell = 64$ sampled subgraphs. The implementation (from the authors) is partially parallelized and leverages multicore processors.
- Minutoli et al.’s IMM is a high-performance, parallel algorithm that efficiently produces accurate seed sets (Minutoli et al., 2019). It is an approximation method that improves the Reverse Influence Sampling (RIS) (Borgs et al., 2014) algorithm by eliminating the need for the threshold to be used. We have used $\epsilon = 0.5$ as suggested in the original paper, where ϵ is a user-defined parameter to control the approximation boundaries.

5.5.4 Comparing HYPERFUSER with State-of-art

To compare the run time, memory use, and quality of HYPERFUSER with those of the state-of-the-art, we perform experiments using the following parameters controlling the quality of the seed sets: TIM+ ($\epsilon = 0.3$), IMM ($\epsilon = 0.5$), SKIM ($l = 64, k = 64$). In fact, one of the drawbacks of HYPERFUSER is that it does not have a direct control over the approximation factor, whereas TIM+ and IMM have one. Still, HYPERFUSER can control the quality indirectly by tuning the number of Monte-Carlo simulations \mathcal{J} which also increases the number of sketches used per vertex. In the experiments, we set $\mathcal{J} = 256$. In addition, as explained in the previous section, we use a global error threshold $\epsilon_g = 0.01$, a local error threshold as $\epsilon_l = 0.3$, and the early-exit ratio as $\epsilon_c = 0.02$.

Table 5.2 HYPERFUSER execution times (in secs), influence scores, and memory use (in GBs) on the networks with $K = 50$ seeds using $\tau = 18$ threads and constant edge weights $w = 0.005$. Influence scores are given relative to HYPERFUSER. The runs that did not finish due to high memory use shown as "-". The values in the last 4 rows are normalized w.r.t. to those of HYPERFUSER.

Method Dataset	Time				Influence Score				Memory			
	HYPER FUSER	TIM+	IMM	SKIM	HYPER FUSER	TIM+	IMM	SKIM	HYPER FUSER	TIM+	IMM	SKIM
Amazon	1.30	124.38	5.58	63.73	96.9	1.041×	1.000×	0.562×	0.17	21.62	0.90	6.78
DBLP	1.61	177.99	5.71	28.24	106.6	1.068×	1.027×	1.036×	0.27	31.24	0.95	3.12
Epinions	1.11	12.16	0.50	8.29	635.3	1.026×	1.001×	0.939×	0.06	0.78	0.07	1.04
LiveJ	13.25	4172.69	118.82	19.35	37174.1	1.010×	0.995×	0.957×	3.97	27.43	2.78	2.00
NetHEP	0.31	2.22	0.31	1.96	80.2	1.065×	0.993×	0.871×	0.01	0.80	0.04	0.33
NetPhy	0.39	7.40	0.40	1.00	124.5	1.042×	0.999×	0.777×	0.03	2.01	0.08	0.16
Orkut	30.22	-	780.22	41.82	158842.6	-	0.997×	1.001×	5.19	-	7.81	1.77
Pokec	11.05	149.34	5.04	18.40	1095.1	1.032×	1.027×	0.925×	1.57	10.16	1.40	1.74
Sdot0811	1.18	6.93	0.40	1.00	576.4	1.015×	0.983×	0.942×	0.06	0.76	0.09	0.13
Sdot0902	1.14	6.96	0.40	1.17	610.5	1.022×	0.998×	0.953×	0.06	0.71	0.08	0.14
Twitter	1.10	171.17	5.40	1.60	3458.7	1.006×	0.990×	0.942×	0.09	2.02	0.12	0.15
Youtube	1.95	46.61	2.42	13.24	1820.8	1.025×	1.013×	1.000×	0.73	3.96	0.48	1.39
Arit. mean		69.39×	4.37×	8.13×		1.032×	1.002×	0.909×		42.60×	2.05×	9.76×
Geo. mean	all	28.19×	1.69×	3.47×	all	1.032×	1.002×	0.899×	all	22.75×	1.65×	3.58×
Max. perf	1.00×	314.92×	25.82×	49.02×	1.000×	1.068×	1.027×	1.036×	1.00×	127.18×	5.29×	39.88×
Min. perf		5.88×	0.34×	0.85×		1.006×	0.983×	0.562×		5.43×	0.66×	0.34×

Table 5.3 HYPERFUSER execution times (in secs), influence scores, and memory use (in GBs) on the networks with $K = 50$ seeds using $\tau = 18$ threads and constant edge weights $w = 0.01$. Influence scores are given relative to HYPERFUSER. The runs that did not finish due to high memory use shown as "-". The values in the last 4 rows are normalized w.r.t. to those of HYPERFUSER.

Method Dataset	Time				Score				Memory			
	HYPER FUSER	TIM+	IMM	SKIM	HYPER FUSER	TIM+	IMM	SKIM	HYPER FUSER	TIM+	IMM	SKIM
Amazon	0.96	107.94	3.28	59.32	152.7	1.037×	1.024×	0.390×	0.17	18.11	0.55	6.40
DBLP	0.73	73.37	2.85	18.71	233.5	1.043×	1.005×	0.997×	0.27	11.92	0.52	2.05
Epinions	0.82	112.08	3.78	5.07	2480.1	1.006×	0.983×	0.984×	0.06	1.95	0.10	0.68
LiveJ	16.72	-	386.37	16.23	155375.8	-	0.996×	0.993×	3.97	-	6.64	1.45
NetHEP	0.26	1.84	0.23	1.89	129.1	1.036×	0.997×	0.826×	0.01	0.60	0.03	0.31
NetPhy	0.24	3.33	0.23	0.86	320.5	1.010×	0.985×	0.732×	0.03	0.67	0.05	0.12
Orkut	42.37	-	1870.35	114.82	650157.1	-	1.000×	1.000×	5.19	-	20.13	3.43
Pokec	11.65	4148.03	88.89	7.25	44685.8	1.004×	0.996×	0.988×	1.57	39.98	2.09	0.78
Sdot0811	0.84	102.96	3.70	0.87	2882.1	1.003×	0.984×	0.976×	0.06	2.01	0.10	0.08
Sdot0902	0.90	129.31	4.19	0.88	3061.5	1.008×	0.992×	0.980×	0.06	2.42	0.11	0.08
Twitter	0.90	390.36	10.96	1.22	9628.6	1.004×	0.992×	0.978×	0.09	4.91	0.23	0.09
Youtube	2.18	534.41	14.86	16.97	9042.7	1.009×	0.994×	1.006×	0.73	7.10	0.48	1.73
Arit. mean		167.17×	9.73×	9.99×		1.016×	0.996×	0.904×		42.91×	2.09×	8.26×
Geo. mean	all	100.14×	5.45×	3.58×	all	1.016×	0.996×	0.879×	all	36.26×	1.91×	2.82×
Max. perf	1.00×	433.73×	44.14×	61.79×	1.000×	1.043×	1.024×	1.006×	1.00×	106.53×	3.88×	37.65×
Min. perf		7.08×	0.89×	0.62×		1.003×	0.983×	0.390×		9.73×	0.66×	0.37×

Table 5.4 HYPERFUSER execution times (in secs), influence scores, and memory use (in GBs) on the networks with $K = 50$ seeds using $\tau = 18$ threads and constant edge weights $w = 0.1$. Influence scores are given relative to HYPERFUSER. The runs that did not finish due to high memory use shown as "-". The values in the last 4 rows are normalized w.r.t. to those of HYPERFUSER.

Method Dataset	Time				Score				Memory			
	HYPER FUSER	TIM+	IMM	SKIM	HYPER FUSER	TIM+	IMM	SKIM	HYPER FUSER	TIM+	IMM	SKIM
Amazon	0.69	133.22	1.98	23.62	11797.0	1.006×	0.990×	0.815×	0.17	5.49	0.23	2.59
DBLP	0.54	1368.83	14.54	6.30	48549.9	1.001×	0.995×	1.001×	0.27	35.19	1.06	0.65
Epinions	0.26	439.33	5.46	9.42	18409.9	1.000×	0.998×	0.997×	0.06	12.17	0.39	1.18
LiveJ	10.10	-	1071.30	65.73	2134726.0	-	1.000×	1.000×	3.97	-	65.49	1.40
NetHEP	0.10	14.18	0.33	0.61	2461.7	1.002×	0.975×	0.899×	0.01	1.02	0.04	0.10
NetPhy	0.16	107.52	1.53	0.34	8339.5	1.007×	0.994×	0.975×	0.03	3.84	0.13	0.03
Orkut	16.55	-	1964.83	446.92	2692366.5	-	1.000×	1.000×	5.19	-	71.94	9.68
Pokec	4.90	-	514.79	31.80	1034859.8	-	1.000×	1.000×	1.57	-	26.46	0.98
Sdot0811	0.21	677.49	7.34	2.44	25871.8	1.000×	1.000×	0.999×	0.06	19.10	0.59	0.25
Sdot0902	0.23	695.12	7.99	2.35	27519.5	1.000×	1.000×	0.999×	0.06	18.45	0.66	0.24
Twitter	0.33	1897.50	16.09	1.62	55327.3	1.000×	0.998×	0.998×	0.09	34.56	1.04	0.05
Youtube	1.12	7158.56	60.59	30.57	171392.9	1.000×	0.999×	1.001×	0.73	139.12	4.19	2.88
Arit. mean		2624.61×	47.17×	15.37×		1.002×	0.996×	0.974×		199.54×	8.79×	5.32×
Geo. mean	all	1449.41×	27.87×	11.06×	all	1.002×	0.996×	0.972×	all	163.77×	7.15×	2.63×
Max. perf	1.00×	6391.57×	118.72×	36.23×	1.000×	1.007×	1.000×	1.001×	1.00×	384.00×	16.85×	19.67×
Min. perf		141.80×	2.87×	2.13×		1.000×	0.975×	0.815×		32.29×	1.35×	0.35×

We present the results in Tables 5.2, 5.3 and 5.4 for edge weights $w = 0.005$, 0.01 , and 0.1 , respectively. The top part of each table shows the results for the networks, and the bottom four rows are the arithmetic mean, geometric mean, maximum and minimum, respectively, of the scores after they are normalized w.r.t. those of HYPERFUSER’s scores. In all tables, for the execution time (2–5) and memory (10–13) columns, lower values are better. For the influence scores, i.e., for columns 6–9, higher values are better.

The tables show that for small and relatively sparser graphs such as NetHep, NetPhy, DBLP and Amazon, TIM+, the high-quality baseline, has 0.1%–7% more influence score compared to the proposed approach. Except DBLP, the other sketch-based algorithm, SKIM also performs bad on these graphs. For NetPhy, NetHEP, and Amazon, its influence scores are 10%–44% worse than those of HYPERFUSER. For the rest of the graphs, TIM+ is only up to 3%, 0.9% and 0.1% better in terms of influence for the edge weights $w = 0.005$, 0.01 , and 0.1 , respectively, while being 69×, 167×, and 2624× slower on average over all the graphs. It is clear that HYPERFUSER’s influence performance is getting closer to that of TIM+ when w increases. Indeed, when w is small, e.g., 0.005 , it may have a hard time while catching potential influence paths; the probability an edge being captured is $1 - (1 - 0.005)^{256} = 0.72$. Using $\epsilon = 0.3$, TIM+ does not suffer from sparsity, but as the tables show, SKIM can suffer more. With respect to the execution-time performance, HYPERFUSER is superior to other methods; for instance, when $w = 0.01$, it is 167×, 10×, and 10× faster on average compared to TIM+, IMM, and SKIM, respectively. Although

IMM and SKIM look similar for $w = 0.01$ in terms of relative average execution time performance, for large graphs, SKIM is faster than IMM. When $w = 0.01$, the maximum execution times for 4148, 1870, and 114 seconds for TIM+, IMM, and SKIM, respectively. For the proposed approach and with the same w , the maximum time spent is only 42 seconds.

HYPERFUSER’s memory consumption is less compared to those of others. Furthermore, it stays the same for all experimental settings with different w values. This is partly due to fused sampling; the memory consumption is linearly dependent only on the number of vertices in G . Both sketch-registers and *visited* information are stored per vertex. Hence, HYPERFUSER’s memory consumption stays constant for any simulation parameters or any number of edges. That is given \mathcal{J} , HYPERFUSER’s memory use is predictable for any graph. On the other hand, the other methods’ memory consumptions tend to increase with w , and their behaviours change with different parameters and graphs.

Overall, the performance characteristics of the proposed algorithm are different from its state-of-the-art competitors. HYPERFUSER’s performance is highly affected by G ’s diameter. For instance, for *Pokec* with $w = 0.01$, the average diameter of the samples is 43, which makes HYPERFUSER to lose its edge against its fastest competitor. On the other hand, with $w = 0.1$, the average diameter is only around 17, and HYPERFUSER is six times faster than its nearest competitor. Indeed, its execution time decreases as the samples and the influence graph G get denser. On the other hand, the other methods tend to get slower under these changes.

Figure 5.4 shows the speedups of HYPERFUSER over IMM for all the graphs and all w values. As described above, the relatively sparser setting $w = 0.005$ is especially challenging due to the high diameter of the influence graph and low vector unit utilization. Even with this w value, HYPERFUSER is only slower by a few seconds and only when the influence is small. For larger graphs with larger influences HYPERFUSER is much faster than IMM. As explained before, for larger w , HYPERFUSER’s execution-time performance is usually better, and its influence quality is on par with that of IMM.

Figure 5.5 compares HYPERFUSER’s execution-time performance with that of SKIM. As the figure shows, the proposed approach performs much better, both in terms of quality and speed in almost all settings. For the notorious *Pokec* dataset, HYPERFUSER performs better than SKIM, except for $w = 0.01$. The diameter of G does not affect SKIM’s performance as much as HYPERFUSER. SKIM is faster in this setting, but it has worse influence quality. In some settings such as *Amazon* and $w = 0.01$, SKIM performs very poorly; only 39% of the influence is achieved with

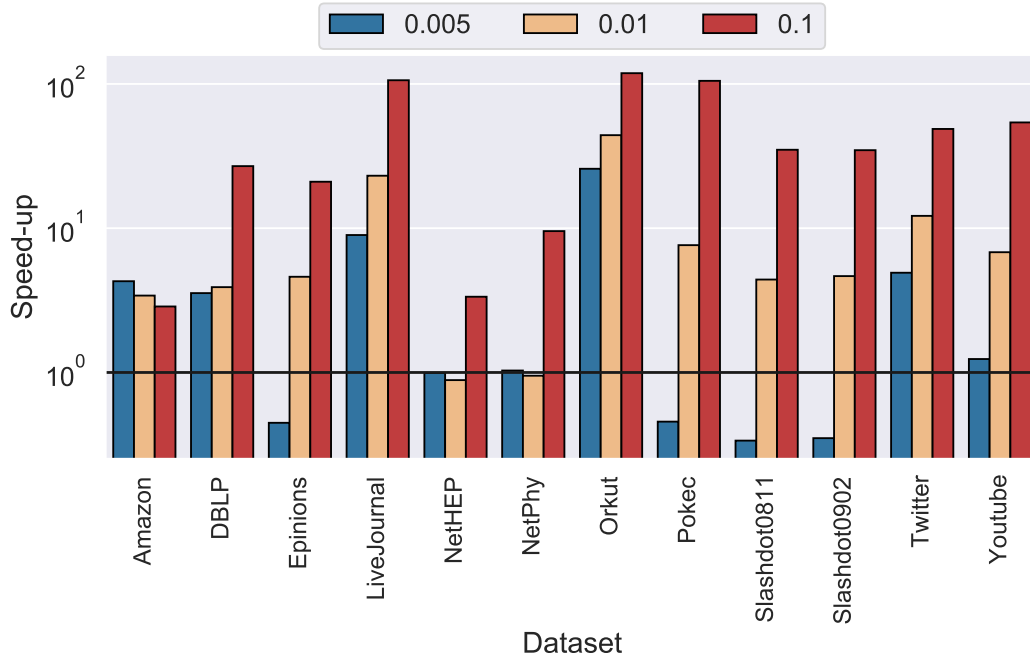


Figure 5.4 Speedups obtained by HYPERFUSER ($\mathcal{J} = 256$) over IMM ($\epsilon=0.5$) using $\tau = 18$ threads.

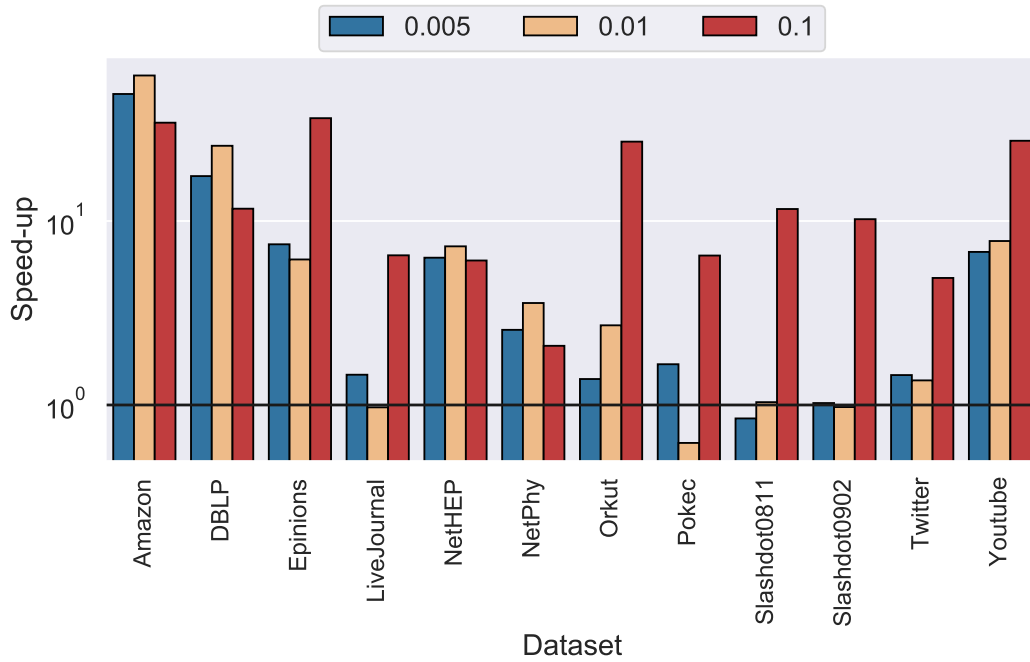


Figure 5.5 Speedups obtained by HYPERFUSER ($\mathcal{J} = 256$) over SKIM ($r=64, l=64$) using $\tau = 18$ threads.

respect to HYPERFUSER. In addition, under the same setting, SKIM spends 59.3 seconds whereas HYPERFUSER finishes in less than one second.

5.5.5 Scalability with multi-threaded parallelism

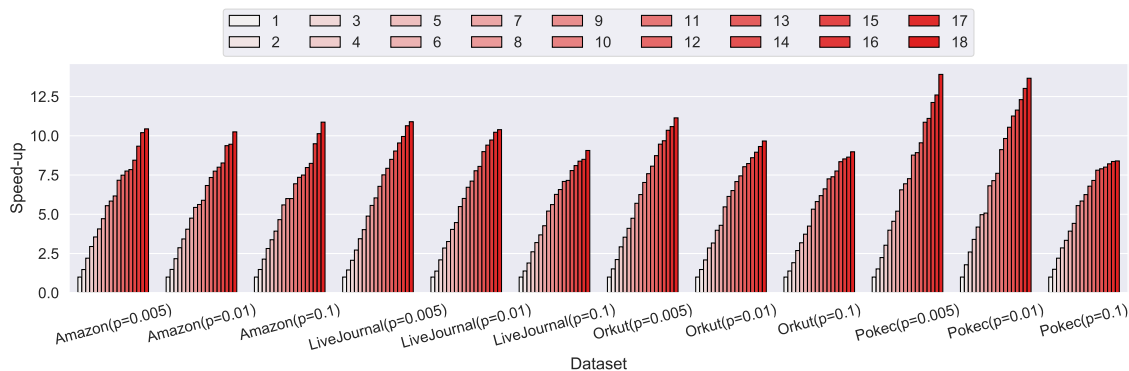


Figure 5.6 Scaling of HYPERFUSER with multiple threads on some of largest datasets in the benchmarks.

In our implementation, we used a *pull-based* approach in which the vertices (processed at line 4 of Algorithm 9) pull the influence, i.e., reachability set cardinality estimations, from their outgoing neighbors. A classical *push-based* approach, in which the vertices relay their influence to their outgoing neighbors could also be leveraged. However, the push-based approach makes a (target) vertex register potentially updated at the same time in different computation units. Specifically, the update operation (corresponding to the one at line 8 of Algorithm 9) of the *pull-based* approach will be the cause of race conditions. One can easily argue that since we are already using sketches, and not computing exact cardinalities, such race conditions are acceptable and they will not reduce the quality of influence estimations. However, the performance may suffer due to false sharing. Figure 5.6 shows HYPERFUSER’s speedup values obtained via a simple `OpenMP` parallelization at line 4 of Algorithm 9. Even though the pull-based diffusion shows a nice parallel performance, it is possible to implement HYPERFUSER using other approaches such as the *queue-based* approach which may improve performance by only processing live vertices. The pull-based diffusion method is chosen due to its simplicity and scalability to many threads.

6. A MULTI-GPU APPROACH TO FUSED-SAMPLING AND INFLUENCE MAXIMIZATION

Even though both INFUSER-MG and HYPERFUSER are built for scalability, parallelism, and independent execution from the ground up, IM on GPUs brings many additional optimization challenges and opportunities. The differences in between the architectures can be exploited with the use floating-point values and how the edges are sampled. The parameters for the CUDA kernel launches are denoted as «`blocks`, `threads`» through the chapter. In the algorithms, block and thread IDs are represented with “`blockIdx.x`” and “`threadIdx.x`”. The size of the (1D) grid in terms of blocks is shown with “`gridDim.x`”. The maximum number of blocks per kernel is predefined and shown as B . As mentioned before, the CUDA runtime allows multiple dimensions for blocks and threads. Even though we have shown the first dimension with “`.x`” to be compatible with CUDA API, only a single dimension is used for all algorithms in this chapter.

6.1 Count-distinct Sketches for GPUs

The count-distinct sketch used in HYPERFUSER is designed for simplicity and optimized for AVX2/AVX512 capable processors. Due to differences between AVX accelerators and CUDA-capable devices, we explored a new sketch for GPUs. In fact, using 8-bit storage for a single register is space efficient; 7 of the bits can be used to store the number of leading zeros, and the remaining bit can hold the information on whether the vertex has been visited in the respective sample. On the other hand, even though some GPUs (the server-grade ones) are good at processing 8-bit elements, their floating-point performance is unparalleled. A float can store much more information, even when holding leading zeros. To exploit this, we have

devised a method that holds multiple $clz \circ h$ in a single float, instead of storing all registers separately. We reduced a small number of consecutive registers to a single register using harmonic mean. That is instead of 256 8-bit registers, we are using 64 floating-point registers, each is holding a harmonic mean of 4 registers of the former scheme. Formally,

$$(6.1) \quad M[j] = H(clz(h_{4j}), clz(h_{4j+1}), clz(h_{4j+2}), clz(h_{4j+3})), \quad 1 \leq j \leq \mathcal{J}$$

where H is the harmonic mean function, clz is count leading zeros function, and h_j refers to j th hash function.

Aggregated floating-point sketches allow us to remove the extreme values before they affect the rest without any post-processing done. A similar approach has been applied in HYPERLOGLOG++ (Heule, Nunkesser, & Hall, 2013); HYPERLOGLOG++ removes 30% of the largest values to improve the sketch accuracy.

In our experiments, we have observed that floating-point sketches proposed for SUPERFUSER are on par with those of HYPERFUSER in much smaller register counts J . Instead of using $J = 256$ registers in integer sketches, $J = 64$ floating-point sketches can be used to reduce the total number of operations while consuming the same amount of storage. For CPU implementations, integer sketches are still preferable since 32 8-bit vector operations can be performed together in SIMD fashion with AVX2, and 64 of them with AVX512. Whereas on the GPU, 32 registers can be concurrently processed by a single warp. Furthermore, as our preliminary experiments show, the floating-point sketches use less number of registers to estimate within the same accuracy. Especially on consumer-grade RTX cards, floating-point operations are much faster, so floating-point sketches are strongly preferred. We also encoded the *visited* information on the vertices inside the sketch registers. This allows us to find the sample IDs for each vertex u such that u is active only on these samples without requiring any other memory usage. The algorithm for initializing sketches is given in Algorithm 10.

6.2 Influence Cascade on GPU

Given the current seed set, almost all of the IM algorithms must find the set of influenced vertices for all samples. This allows the algorithm to first compute the current

Algorithm 10 FILL-SKETCHES(G, M, \mathcal{J}, s)

Input: $G = (V, E)$: the influence graph
 M : sketch vectors of vertices
 \mathcal{J} : number of registers per vertex
 τ : partition id

Output: M : updated sketch vectors

```
1:  $i \leftarrow \text{blockIdx.x}$ 
2:  $j \leftarrow \text{threadIdx.x}$ 
3:  $\text{stride} \leftarrow \text{gridDim.x}$ 
4: while  $i < |V|$  do
5:   if  $M_i[j] \neq -1$  then
6:      $\text{acc} \leftarrow 0$ 
7:     for  $l \in [\tau\mathcal{J} + 4j, \tau\mathcal{J} + 4j + 4)$  do
8:        $\text{acc} \leftarrow \text{acc} + 1/\text{clz}(\text{hash}_l(i))$ 
9:     if  $\text{acc} \neq 0$  then
10:       $M_i[j] \leftarrow 4/\text{acc}$ 
11:     $i \leftarrow i + \text{stride}$ 
```

influence score and be ready to efficiently compute the marginal gain for the next seed candidate. This process, which simply requires multiple sample graph traversals in our algorithms, will be called as Influence Cascade. Independently traversing each sample graph using a queue for each worker is a trivial approach for performing the influence cascade on GPU in parallel. However, this approach requires $\mathcal{O}(|V|)$ storage per worker which makes it infeasible on GPUs which have a limited memory. Even when a small number of workers are used, this approach incurs path divergence within each warp. Divergent warp problem can be mitigated by processing a vertex per warp while processing the edges associated with that vertex with the warp threads (Shahrouz, Salehkaleybar, & Hashemi, 2021). However, this adds an additional limit on the parallelism. Another solution to the divergence problem is combining the frontiers, i.e., taking their union, for each sample, and processing the vertices that appear in a frontier of at least one sample in parallel. One needs to repeat this process while keeping track of any updates until the convergence, the state with no update, is reached which implies that all traversal levels have been processed for all samples. Similar approaches have been applied for different graph kernels such as centrality computation by Sariyüce et al. (2015). In HYPERFUSER and SUPERFUSER, we have used the latter except that we integrate the fused sampling to the influence cascade process for all samples and for each edge. Instead of using multiple queues, i.e., one queue per sample, we used a single, unified queue to store the distinct vertices to be traversed. For simplicity (in implementation), this queue is split into two segments; the first segment stores the vertices that will be processed in the current kernel launch, and the second accumulates the vertices to

be processed in next iteration. At each iteration, we started a block for each vertex in the first segment (within an upper limit on the number of concurrent blocks). Then each thread in the block processes the same edge for all samples. Finally, the vertices visited by the threads are accumulated and added to the latter segment of the queue to be processed in the next iteration. After the kernel finishes, we swap the first and latter segments and repeat until no more elements in the queue.

Algorithm 11 SIMULATE-SKETCHES(G, M, \mathcal{J})

Input: $G = (V, E)$: the influence graph

M : sketch vectors of vertices

\mathcal{J} : number of registers per vertex

Output: M : updated Sketch vectors

```

1:  $i \leftarrow \text{blockIdx.x}$ 
2:  $j \leftarrow \text{threadIdx.x}$ 
3:  $\text{stride} \leftarrow \text{gridDim.x}$ 
4: while  $i < |V|$  do
5:   if  $M_i[j] = -1$  then
6:     continue
7:   for  $e_{i,v} \in A(i)$  do
8:     if  $P(u,v)_j < w_{u,v} \wedge u \notin R_S[j]$  then
9:        $M_u[j] \leftarrow \max(M_u[j], M_v[j])$ 
10:   $i \leftarrow i + \text{stride}$ 

```

Processing the same edge by all threads in a block allows the (shared or global) memory to broadcast the data item as well as the algorithm to exploit shared memory for fast processing. Furthermore, divergent paths in the warps will be eliminated. With this approach, a single block reads and updates consecutive memory locations. The only performance disrupting memory access is the single atomic operation done for adding a new vertex to the queue. To further reduce the memory pressure, as mentioned above, we used the sketch registers to hold visited vertices. Since the visited vertices (by the current seed set) in a sample cannot have an impact on the influence, we set the sketch register of that sample to a hard-coded value, -1, to both remove that register for cardinality estimation and label the vertex visited in the associated sample. This kernel is described in Algorithm 12. With the proposed approach, we were able to process thousands of vertices in parallel without memory bottlenecks. The run-time analysis shows > 80% utilization of the streaming multiprocessors and the memory sub-system.

6.3 Fused-Sampling Aware Sample-Space Split

Algorithm 12 CASCADE-KERNEL($G, M, \mathcal{J}, Q, Q_{next}$)

Input: $G = (V, E)$: the influence graph M : sketch vectors of vertices \mathcal{J} : number of registers per vertex Q : current queue Q_{next} : current queue **Output:** M : updated Sketch vectors

```
1:  $i \leftarrow blockIdx.x$ 
2:  $j \leftarrow threadIdx.x$ 
3:  $stride \leftarrow gridDim.x$ 
4:  $rnd \leftarrow rands[j]$ 
5: while  $i < size(Q)$  do
6:    $u \leftarrow Q[i]$ 
7:   if  $M_v[j] \neq 1$  then
8:     continue
9:   for  $e_{u,v} \in A(u)$  do
10:     $flag \leftarrow 0$ 
11:    if  $P(u,v)_j < w_{u,v} \wedge u \notin R_S[j]$  then
12:       $M_u[j] \leftarrow -1$ 
13:       $flag \leftarrow 1$ 
14:    if  $j == 0$  and  $ballot(flag) == 1$  then
15:       $Q_{next} \leftarrow Q_{next} \cup \{v\}$  ▷ Atomic
16:     $i \leftarrow i + stride$ 
return  $M$ 
```

Deterministic generation of random values opens many opportunities for performance. Since random values are generated using a random seed vector X , we can manipulate, i.e., permute, the vector entries for a better performance without losing randomness. Putting similar X_r together reduces the divergence in code path, and hence, the number of “wasted” compute resources in SIMD lanes that do not contribute to the final result. On AVX2/AVX512, while the fused-sampling step processes edges, a batch of samples are processed regardless if some of the samples do not include the edge. After this step, only the relevant results are updated using the SIMD shuffle/mask instructions. On the GPU, similarly, a warp diverges on relevant samples, and the rest of the threads wait for joining to the next batch.

As a simple yet effective approach, we opted to sort random seed values in the vector X so that similar bit-flips are clustered with respect to their significance. For one end of the sorted vector, we expect the lower bits of the edge hash is flipped to generate random value, so the values for the same edge, but consecutive samples, will be close to each other. On the other end, regardless of the remaining ones, we expect the higher bits to be flipped. Still, for consecutive locations the higher bit positions that are flipped are expected to be the same. This approach improves the fill rate of our SIMD lanes (on GPU this corresponds to a warp) during fused

Table 6.1 Average size and maximum achievable speed-up using fused-sample aware split method. Selected datasets are split into two and four batches using sampling probability $p = 0.01$.

Dataset	2 Devices, Batch Size=128				4 Devices, Batch Size =64			
	Average Size		Max Speed-up		Average Size		Max Speed-up	
	Naive	Sorted	Naive	Sorted	Naive	Sorted	Naive	Sorted
Amazon	83%	50%	2.41	3.85	59%	26%	6.67	14.81
DBLP	75%	49%	2.63	3.92	49%	25%	8.00	14.81
Epinions	78%	49%	2.56	3.92	53%	25%	7.41	14.81
LiveJournal	81%	50%	2.44	3.85	57%	25%	6.90	14.81
NetHEP	83%	50%	2.41	3.85	61%	26%	6.35	14.29
NetPhy	79%	50%	2.50	3.85	56%	25%	7.02	14.81
Orkut	75%	49%	2.63	3.92	49%	25%	8.00	14.81
Pokec	79%	49%	2.53	3.92	54%	25%	7.27	14.81
Slashdot0811	82%	50%	2.44	3.85	58%	25%	6.78	14.81
Slashdot0902	81%	50%	2.44	3.85	57%	25%	6.78	14.81
Twitter	80%	50%	2.47	3.85	57%	25%	6.90	14.81
Youtube	75%	49%	2.63	3.92	49%	25%	8.00	14.81
Geo. Mean	79%	50%	2.50	3.88	55%	25%	7.15	14.77

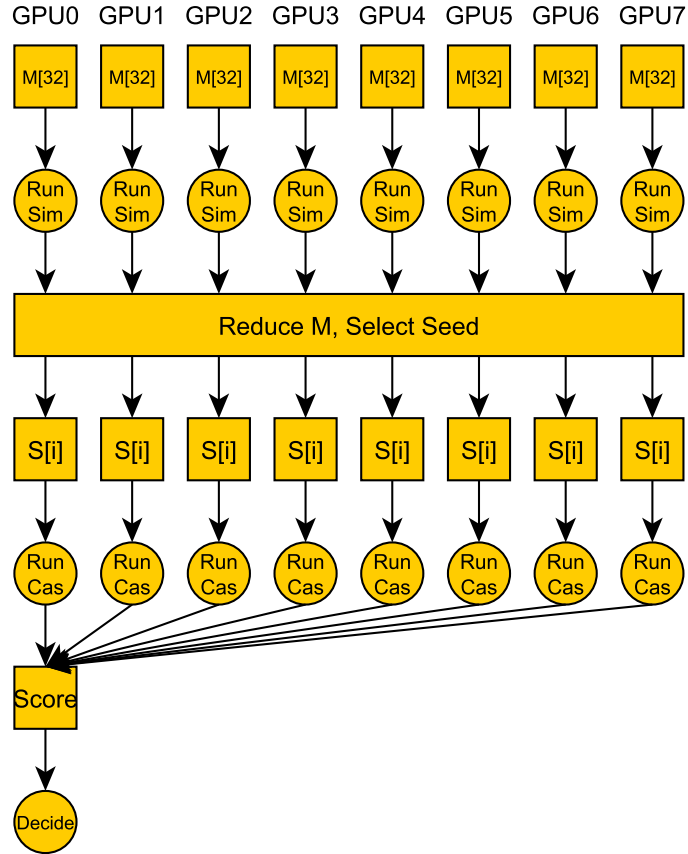
sampling, and allows us to provide an early exit for empty masks. On average 30% speed-up can be achieved using sorted random seeds without any significant changes to the implementation. In fact, all the impacts in this paragraph are achieved with a single line. On AVX512, the instructions that can gather 512-bit vectors to 32/64-bit integers make the early-exit strategies more effective than AVX2 counter-parts.

Building on the idea in the previous paragraph, we propose a *Fused-Sampling Aware Sample-Space Split* method. The proposed method, while distributing the samples to multiple tasks, takes the advantage of the pseudo-random number generator to reduce the total number of edges traversed. For large edge probabilities w , previously proposed methods, INFUSER-MG and HYPERFUSER, converges very quickly. For small edge probabilities, since only a small fraction of the edges can be active in a limited number of samples, the total number of edges to be processed is very small compared to the whole graph. In addition, sorting random seeds allows us to cluster sampled and not sampled edges to sample batches so that each batch has fewer edges compared to the unsorted case. This method starts with a predefined number of batches μ . First, the random values are split into μ batches of equal size. For each batch, we sample the edges for all possible samples with the random X values in that batch. If the edge is sampled with any of the random values, we copy that edge to the task. Otherwise, the edge is skipped. Table 6.1 shows the effect of splitting 256 samples to two and four batches using both naive and the proposed approach.

6.4 The SUPERFUSER algorithm

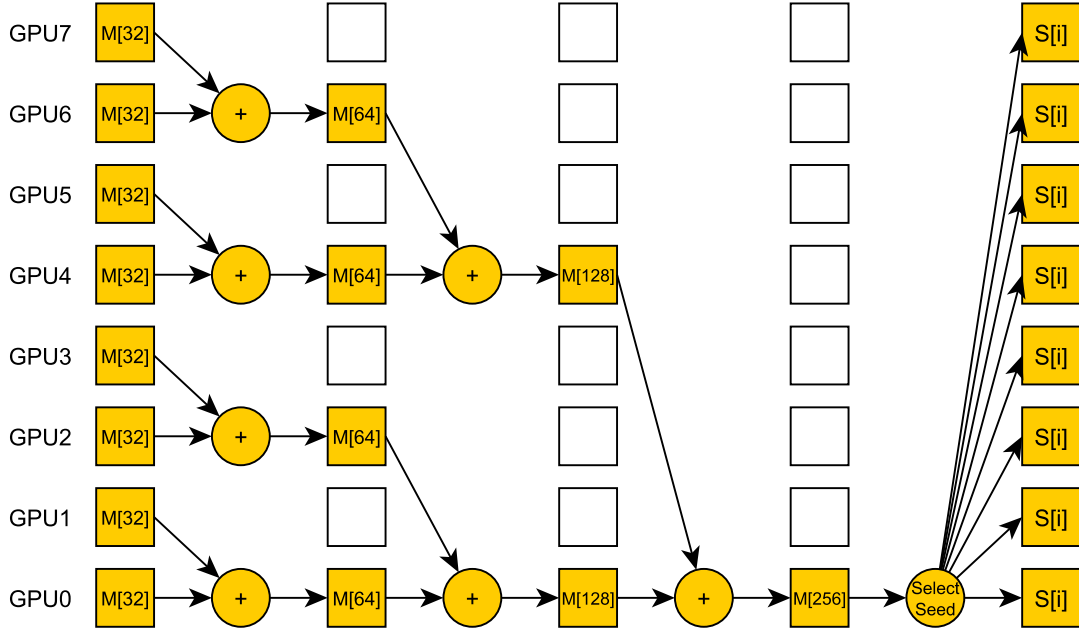
Building on the mentioned methods, we propose SUPERFUSER algorithm, a multi-GPU approach to Fused-Sampling and Influence Maximization. SUPERFUSER consists of several distributed steps to efficiently compute the reachability sets and select the seed vertices. First, all the devices start from their own random state which is an equisized partition of X . Then, the edges are distributed to each device with respect to the fused-sampling aware sample-space split method. Each partition creates its own sketches for all vertices using its own random X values and initializes sketches with the vertices. Each device runs simulations on the sketches using the edges assigned to the partition. After, each device takes the element-wise sum of registers to find cardinality estimates of samples in its own partition. Then, a binary partition reduction is performed to find the seed vertex. Then, each device performs a random cascade to calculate influence, and label the visited vertices. Finally, the first device reads scores from other devices and decides whether to rebuilt sketches in the next iteration. Figure 6.1 describes how SUPERFUSER distributes data and schedules the communication among multiple GPUs.

Figure 6.1 An example run of a seed selection in SUPERFUSER; The example scheme uses 8 GPUs and $\mathcal{J} = 256$ registers. $M[j]$ represents a sketch with j registers. Timeline goes from top to bottom.



In the seed selection step, we employ a binary reduction scheme to improve the performance. Especially when the number of devices is high, i.e., 4-8, most of the time is spent on the communication among the GPUs rather than taking the sketch-wise sum of registers and selecting vertex with the largest cardinality. Due to this reason, SUPERFUSER first performs partial sketch sums in local devices. Then, the devices with IDs divisible by $z = 2$ reduce their sketch sums with the device $(ID + y)$ for $y = 1$ (the result is stored on the device with smaller ID). After, we iterate the same process with IDs divisible by doubling z and y until all sketch-sums are accumulated in the device with $ID = 0$. Then, device 0 selects the vertex with the largest sketch sum as a seed vertex, and the vertex ID is broadcasted to all devices. Figure 6.2 illustrates an example reduction scheme with 8 devices and $\mathcal{J} = 256$ registers per sketch, where each device is responsible for $\mathcal{J} = 32$ sketches.

Figure 6.2 The reduction scheme of SUPERFUSER to find a seed vertex using sketch registers. The scheme uses 8 GPUs and $\mathcal{J} = 256$ registers. $M[j]$ represents a sketch with j registers. The symbol $+$ represents the element-wise sum of registers. Yellow boxes show active devices. The timeline goes from left to right.



Even though the algorithm is parallel in most of the execution, it needs to synchronize all devices in several places. For instance, the seed selection requires all devices to be synchronized before the reduction, as well as before the broadcasting the seed vertex. Furthermore, after running a random influence cascade, a barrier is required to estimate the scores correctly. In total, $\log_2(\#devices) + 1$ barriers are required for seed selection. These being said, since the algorithm is probabilistic and prone to small errors, we can relax some of these barriers if the error rate is lower than the rebuilding threshold. That is for these points, the exact influence score is not important. Algorithm 13 describes SUPERFUSER in detail including its synchronization points.

Algorithm 13 SUPERFUSER($\check{G}_i, \mathcal{J}, K, i, \mu$)

Input: $\check{G}_i = (V, E)$: i th partition of the graph \mathcal{J} : number of Sketch registers K : Size of the seed set id : Device id μ : Number of devices**Output:** S : Seed set

```
1:  $S \leftarrow \emptyset$ 
2:  $score \leftarrow 0, oldscore \leftarrow 0$ 
3:  $M \leftarrow \text{ZEROS}(|V \in \check{G}_i|, \mathcal{J})$ 
4:  $M \leftarrow \text{FILL-SKETCHES}\langle B, J \rangle(M, \mathcal{J}, i)$ 
5:  $M \leftarrow \text{SIMULATE-GPU}\langle B, J \rangle(\check{G}_i, M, \mathcal{J})$ 
6: while  $|S| < K$  do
7:   BARRIER ▷ Synchronize all devices
8:    $sums \leftarrow \text{SKETCHWISE-SUM}(M)$ 
9:    $sums \leftarrow \text{REDUCE}(sums, i, \mu)$ 
10:   $s \leftarrow \text{ArgMax}(sums)$ 
11:  BROADCAST  $s$  TO ALL
12:  BARRIER
13:  if  $i=0$  then
14:     $S \leftarrow S \cup \{s\}$ 
15:   $Q \leftarrow \{s\}, Q_{next} \leftarrow \emptyset$ 
16:  while  $Q \neq \emptyset$  do
17:     $blocks \leftarrow \min(\text{size}(Q), B)$ 
18:     $\text{CASCADE-KERNEL}\langle \text{BLOCKS}, J \rangle(\check{G}_i, M, \mathcal{J}, Q, Q_{next})$ 
19:     $Q \leftarrow Q_{next}, Q_{next} \leftarrow \emptyset$ 
20:   $localscore \leftarrow \text{Count}(M, -1)$ 
21:  BARRIER
22:   $oldscore \leftarrow score$ 
23:   $score \leftarrow \text{GATHER}(localscore, +) / \mu$ 
24:  if  $|(score - oldscore)| / score > e_{global}$  then
25:     $M \leftarrow \text{FILL-SKETCHES}\langle B, J \rangle(M, \mathcal{J}, i)$ 
26:     $M \leftarrow \text{SIMULATE-GPU}\langle B, J \rangle(\check{G}_i, M, \mathcal{J})$ 
27: return  $S$ 
```

6.5 Experimental Results

The experiments are performed on a server with two 64-core AMD EPYC 7742 processors, 1TB memory, and eight Nvidia A100s. In total, 128-core/256-threads were available for CPU computing, and $8 \times 6912 = 55296$ CUDA cores with $8 \times 80 = 640$ GB VRAM were available for GPU computing. On the software side, CentOS 9.2 with GCC 9.2 and CUDA 11 is used to compile and run the experimented software. All the software is compiled using authors' build instructions. Our software uses opti-

Table 6.2 Properties of networks used in the experiments

	Dataset	No. of Vertices	No. of Edges	Avg. Degree
Directed	LiveJournal	4,847,571	68,993,773	14.23
	Pokec	1,632,803	30,622,564	18.75
	Sinaweibo	58,655,849	261,321,071	4.46
Undirected	Friendster	65,608,366	1,806,067,135	27.53
	Orkut	3,072,441	117,185,083	38.14
	Youtube	1,134,891	2,987,625	2.63

mization flags including; "-use-fast-math", "-march=native", and code generation is set for Nvidia Ampere architecture.

In evaluation, we used three influence settings to be comprehensive. For all networks, we evaluated our performance with

1. constant edge weights $p = 0.005$ (as in (Gokturk & Kaya, 2021))
2. constant edge weights $p = 0.01$ (as in (Kempe, Kleinberg, & Tardos, 2003) and (Chen, Wang, & Yang, 2009)),
3. constant edge weights $p = 0.1$ (as in (Kempe, Kleinberg, & Tardos, 2003))

As before, the first setting we used, $p = 0.005$ is selected especially for challenging our method. The setting causes very sparse influence networks. Although our opponents' runtimes increase with influence score (hence, they perform well with sparse networks), SUPERFUSER's runtime increases with the depth of influence graph (which is higher for sparse networks). This setting allows us to show our performance in worst-case scenarios. The other two settings are selected as they appear in the literature.

Experiments are performed on six large social networks; LiveJournal, Pokec, and Sinaweibo are directed networks, whereas Friendster, Orkut, and Youtube networks are undirected. All networks mentioned are taken from SNAP network database (Leskovec & Krevl, 2014). For all experiments, we set seed set size as $K = 50$ as in (Chen, Wang, & Yang, 2009).

We evaluated the algorithms on their execution times and influence scores. In optimization problems like IM, faster computation is not the sole objective. The methods under test should also provide high-quality results for being practical. Even though SUPERFUSER can compute high-quality seed sets on largest datasets without rebuilding, and be order of magnitude faster, we still employ the rebuilding strategies so that the same parameters provide good results on all of the datasets while still being faster than the competition. The memory use is another important factor, yet,

for this set of experiments, we opt not to report the memory statistics of the methods. SUPERFUSER has the same memory use characteristics as HYPERFUSER, only J items per vertex are stored for sketches, and only 9 bytes per vertex are used for traversal tasks. On the other hand, our main baseline, GIM consumes much larger memory and, in some cases, limits its memory use with hard-coded limits. In such a case, we believe it would not be fair to compare memory use.

SUPERFUSER performs Monte-Carlo simulations to report its own estimated influence scores. On the other hand, GIM refers to its own Reverse-Inverse-Reachability sets and relies on another oracle for reliable influence estimations. Even though the reliability of the hash-based, fused-sampling method is shown by the experiments on INFUSER-MG & HYPERFUSER, the number of Monte-Carlo simulations performed are fairly few in our settings. On those experiments, to verify the validity of the results, we have used an independent oracle which do not have any optimizations and use a large number of samples employing standard RNGs. However, the datasets for SUPERFUSER experiments are too big for this independently implemented the oracle software. For that reason, in this chapter, we use a fused-sampling oracle implemented with different random seeds and a higher sample count ($\mathcal{R} = 512$) to calculate the influence scores.

6.5.1 Algorithms evaluated in the experiments

Evaluations are done on two algorithms; GIM (Shahrouz, Salehkaleybar, & Hashemi, 2021) and SUPERFUSER. GIM is selected for its best performance and ability to process large datasets. GIM is an efficient parallel implementation of IMM algorithm on GPU. GIM uses two sets as its queue to eliminate race-conditions. Optimizes memory access using a local memory stack for its second queue. Unfortunately, CURIPPLES (Minutoli et al., 2020) took too much time for the experiments; we were not able to complete them in a feasible time.

SUPERFUSER experiments are conducted in both single GPU and multi-GPU settings. We run our experiments using $\mathcal{J} = 64$ for fast computation, and $\mathcal{J} = 256$ to show how SUPERFUSER scales. Multi-GPU code is run on $\{1, 2, 4, 8\}$ GPUs for a comprehensive scaling analysis. For each GPU, we assigned multiples of 32 sketches/samples to utilize the warps efficiently. The single GPU implementation is single-threaded and devoid of barriers and excess device synchronizations. On the other hand, the multi-GPU implementation still uses barriers and synchronization steps even if a single GPU is used. In addition, we run our experiments using

both HYPERFUSER sketches (Gokturk & Kaya, 2021), and improved floating-point sketch designed for GPUs in Algorithm 10. The former is shown with ‘Int’ to indicate it uses integers to represent sketches, and the latter is shown with ‘Float’ since it uses floating-point numbers.

Table 6.3 Single GPU SUPERFUSER performance compared to state-of-art. The values in the last three rows are normalized w.r.t. to those of SUPERFUSER ($\mathcal{J} = 64, \text{FLOAT}$). The Friendster dataset is excluded due to missing table values (i.e., too large to handle via a single GPU).

p	Method Dataset	Score			Time		
		GIM ($\epsilon = 0.3$)	SUPERFUSER ($\mathcal{J} = 256, \text{Int}$)	SUPERFUSER ($\mathcal{J} = 64, \text{Float}$)	GIM ($\epsilon = 0.3$)	SUPERFUSER ($\mathcal{J} = 256, \text{Int}$)	SUPERFUSER ($\mathcal{J} = 64, \text{Float}$)
0.005	LiveJournal	34520	36970	36022	8.35	3.59	1.65
	Orkut	147272	155931	155469	3.40	10.67	2.74
	Pocec	1121	1098	1099	9.45	12.37	3.99
	Sinaweibo	759328	947125	946996	61.01	68.89	15.46
	Youtube	1866	1773	1792	5.13	3.02	1.61
0.01	LiveJournal	112103	152571	152938	5.60	4.69	1.40
	Orkut	514970	629224	649256	2.30	12.25	3.81
	Pocec	44298	44522	44508	7.26	1.30	1.15
	Sinaweibo	1778890	2213637	2213615	44.06	50.05	16.31
	Youtube	8599	9083	9034	4.23	2.06	1.22
0.1	LiveJournal	1803360	1862170	2133416	2.19	3.86	1.53
	Orkut	2395522	2650180	2559172	0.84	5.53	1.72
	Pocec	1035354	1035410	1035428	0.87	1.40	0.53
	Sinaweibo	11203638	14451355	14804055	18.43	77.24	12.21
	Youtube	170919	171089	171052	1.05	0.38	0.21
Geo. Mean		0.90×	0.99×	1.00×	2.28×	2.73×	1.00×
Avg. Performance		0.91×	0.99×	1.00×	2.86×	2.96×	1.00×
Max. Performance		1.04×	1.04×	1.00×	6.31×	6.33×	1.00×

6.5.2 Comparing SUPERFUSER with GIM

In our experiments, we have observed that we can obtain high-quality results with SUPERFUSER using $\mathcal{J} = 64$ floating-point registers which are comparable to using $\mathcal{J} = 256$ integers registers in terms of solution quality. Also, we have found that using $\epsilon = 0.3$ is a sweet-spot for IMM algorithm, which is also suggested in Minutoli et al., 2019. GIM being an IMM implementation, we use $\epsilon = 0.3$ as well.

Table 6.3 shows the comparative performance of the algorithms. On the left side of the table, we show the expected influence score. On the right side, we show the execution time spend on finding the seed set. At the bottom of the table, we offer a statistical comparison with SUPERFUSER ($\mathcal{J} = 64, \text{Float}$) which is both faster in average and outputs higher quality results than the other methods. So, we report the comparative statistics with respect to the SUPERFUSER ($\mathcal{J} = 64, \text{Float}$) setting.

Single GPU SUPERFUSER ($\mathcal{J} = 64, \text{Float}$) on average 2.86 times faster than GIM

method and 2.73 times faster than SUPERFUSER ($\mathcal{J} = 256, \text{Int}$). If we look at maximum speed-up, we see that SUPERFUSER ($\mathcal{J} = 64, \text{Float}$) can be up-to 6.31 times faster than the competition. If we compare the quality of the results, we see that SUPERFUSER performs better than GIM in almost all settings. On average, GIM can find seed sets that have influence 91% of the SUPERFUSER methods. SUPERFUSER in both settings give very similar results; on the average, there exist only 1% difference in scores.

6.5.3 Comparing Multi-GPU SUPERFUSER with GIM

Table 6.4 shows how SUPERFUSER scales to multiple devices. Unfortunately, GIM only utilizes a single GPU, while SUPERFUSER supports multi-GPU execution. For the best performance, SUPERFUSER needs to assign at least $\mathcal{J} = 32$ samples for each GPU to utilize the warps efficiently. Using two GPUs, SUPERFUSER is 5.68 times faster on average than GIM, and up to 12.01 times speed-ups are observed. Compared to single device execution, SUPERFUSER ($\mathcal{J} = 64$) is 2.06 times faster, with a maximum of 5.17 speed-up. Using 8 GPUs, SUPERFUSER ($\mathcal{J} = 256$) achieves 5.88 times speed-up on average, with a maximum of 6.19 speed-up.

Table 6.4 Multi-GPU SUPERFUSER execution time compared to GIM. The values in the last 3 rows are normalized w.r.t. to those of SUPERFUSER ($\mathcal{J} = 64, \text{FLOAT}, 2 \text{ GPUS}$). The missing values are shown with "-".

p	Method Devices Dataset	GIM	SUPERFUSER		SUPERFUSER			
		(e=0.3) 1	($\mathcal{J} = 64, \text{Float}$) 1 2		($\mathcal{J} = 256, \text{Float}$) 1 2 4 8			
0.005	Friendster	-	-	11.54	-	35.18	18.27	15.40
	LiveJournal	8.35	1.65	0.82	2.93	1.61	1.30	1.83
	Orkut	3.40	2.74	0.53	6.69	2.01	0.86	0.62
	Pokec	9.45	3.99	1.60	9.12	3.27	1.77	1.52
	Sinaweibo	61.01	15.46	5.08	51.05	15.36	8.09	6.66
	Youtube	5.13	1.61	0.72	2.58	1.31	0.84	0.75
0.01	Friendster	-	-	14.84	-	79.35	32.33	17.02
	LiveJournal	5.60	1.40	0.69	4.36	1.61	0.95	0.90
	Orkut	2.30	3.81	0.89	12.10	3.62	1.36	0.83
	Pokec	7.26	1.15	1.28	1.21	0.52	0.32	0.94
	Sinaweibo	44.06	16.31	5.94	49.06	17.11	9.08	7.77
	Youtube	4.23	1.22	0.66	2.72	1.49	0.74	0.73
0.1	Friendster	-	-	14.97	-	34.12	19.03	16.10
	LiveJournal	2.19	1.53	0.82	3.54	1.68	0.92	0.73
	Orkut	0.84	1.72	1.04	4.95	1.99	1.31	0.68
	Pokec	0.87	0.53	0.36	1.51	0.72	0.40	0.31
	Sinaweibo	18.43	12.21	7.70	106.44	28.90	12.08	7.69
	Youtube	1.05	0.21	0.21	0.44	0.28	0.21	0.21
Geo. Mean		4.69×	2.06×	1.00×	5.25×	2.29×	1.24×	1.07×
Avg. Performance		5.68×	2.29×	1.00×	6.53×	2.57×	1.33×	1.11×
Max. Performance		12.01×	5.17×	1.00×	13.82×	5.35×	2.18×	2.23×

7. CONCLUSION AND FUTURE WORK

In this thesis, we proposed memory access accumulation methods for probabilistic graph algorithms. The proposed methods are applied to the Influence Maximization problem and possible implementation with a variety of optimizations and experimental performance results are given.

We proposed *fused-sampling*, a technique to construct probabilistic graph algorithms to run on multiple samples while accumulating memory accesses. Combining with the hash-based random number generation, we have also opened many performance optimization opportunities.

Using fused-sampling and hash-based RNG, we build INFUSER-MG, an Influence Maximization algorithm for undirected graphs that is $2.3\times$ – $173.8\times$ faster than state-of-the-art while being superior in terms of influence scores, and using a comparable amount of memory. Also, INFUSER-MG is more than $3000\times$ faster than the greedy approach, which is considered as one of the gold standard algorithms in the literature, in our experiments.

For directed (and hence all) graphs, we proposed a lightning-fast algorithm HYPERFUSER which utilizes count-distinct sketches, similar to the Flajolet-Martin algorithm, to estimate large reachability sets in many samples. To further exploit our performance gains, while computing the seed sets, we leveraged an error-adaptive rebuilding strategy. The experiments showed us that HYPERFUSER generate high-quality seed sets while being up to $119\times$ faster than a state-of-the-art RIS-based algorithm and up to $62\times$ faster than another sketch-based approach and while producing 3%–12% better influence scores.

We then moved our approach to the multi-GPU setting; we proposed SUPERFUSER, which uses Fused-Sampling Aware Sample Space Split to partition edges to multiple GPUs efficiently. We achieved $6.8\times$ speed-up on average using 8 GPUs against a single GPU, and we were able to process a few of the largest networks available. In the future, we would like to investigate how the proposed methods can be applied

to other probabilistic graph algorithms.

BIBLIOGRAPHY

- Ackermann, W. (1928). “Zum Hilbertschen Aufbau der reellen Zahlen”. In: *Math. Ann.* 99, pp. 118–133. DOI: 10.1007/BF01459088.
- Amdahl, Gene M (1967). “Validity of the single processor approach to achieving large scale computing capabilities”. In: *Proceedings of the April 18-20, 1967, spring joint computer conference*, pp. 483–485.
- Appleby, Austin (2017). *MurmurHash3: Information and Performance*. URL: <https://github.com/aappleby/smhasher/wiki/MurmurHash3> (visited on 05/23/2020).
- Arai, Junya et al. (2016). “Rabbit order: Just-in-time parallel reordering for fast graph analysis”. In: *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, pp. 22–31.
- Borgs, Christian et al. (2014). “Maximizing social influence in nearly optimal time”. In: *Proceedings of the 25th annual ACM-SIAM symposium on Discrete Alg.* SIAM, pp. 946–957.
- Chen, Wei, Chi Wang, and Yajun Wang (2010). “Scalable influence maximization for prevalent viral marketing in large-scale social networks”. In: *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, pp. 1029–1038.
- Chen, Wei, Yajun Wang, and Siyu Yang (2009). “Efficient influence maximization in social networks”. In: *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, pp. 199–208.
- Chen, Wei, Yifei Yuan, and Li Zhang (2010). “Scalable influence maximization in social networks under the linear threshold model”. In: *2010 IEEE international conference on data mining*. IEEE, pp. 88–97.
- Cheng, Suqi et al. (2014). “IMRank: influence maximization via finding self-consistent ranking”. In: *Proc. of the 37th International ACM SIGIR Conf. on Research & Development in Information Retrieval*. ACM, pp. 475–484.
- Cohen, Edith (2015). “All-distances sketches, revisited: HIP estimators for massive graphs analysis”. In: *IEEE Transactions on Knowledge and Data Engineering* 27.9, pp. 2320–2334.

- Cohen, Edith and Haim Kaplan (2007). “Summarizing Data Using Bottom-k Sketches”. In: *Proc. of the 26th Symp. on Principles of Distributed Comp.* PODC ’07. Portland, OR, USA: ACM, pp. 225–234.
- Cohen, Edith et al. (2014). “Sketch-based influence maximization and computation: Scaling up with guarantees”. In: *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management*, pp. 629–638.
- Feige, Uriel (1998). “A threshold of $\ln n$ for approximating set cover”. In: *Journal of the ACM (JACM)* 45.4, pp. 634–652.
- Flajolet, Philippe and G Nigel Martin (1985). “Probabilistic counting algorithms for data base applications”. In: *Journal of computer and system sciences* 31.2, pp. 182–209.
- Flajolet, Philippe et al. (2007). “Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm”. In:
- Galhotra, Sainyam, Akhil Arora, and Shourya Roy (2016). “Holistic influence maximization: Combining scalability and efficiency with opinion-aware models”. In: *Proceedings of the 2016 International Conference on Management of Data*. ACM, pp. 743–758.
- Gokturk, Gokhan and Kamer Kaya (May 2021). “Fast and Error-Adaptive Influence Maximization based on Count-Distinct Sketches”. In:
- Göktürk, G. and K. Kaya (2021). “Boosting Parallel Influence-Maximization Kernels for Undirected Networks With Fusing and Vectorization”. In: *IEEE Transactions on Parallel and Distributed Systems* 32.5, pp. 1001–1013. DOI: 10.1109/TPDS.2020.3038376.
- Goyal, Amit, Wei Lu, and Laks V.S. Lakshmanan (2011a). “CELF++: Optimizing the Greedy Algorithm for Influence Maximization in Social Networks”. In: *Proceedings of the 20th International Conference Companion on World Wide Web*. WWW ’11. Hyderabad, India: Association for Computing Machinery, pp. 47–48. ISBN: 9781450306379. DOI: 10.1145/1963192.1963217. URL: <https://doi.org/10.1145/1963192.1963217>.
- Goyal, Amit, Wei Lu, and Laks VS Lakshmanan (2011b). “Simpath: An efficient algorithm for influence maximization under the linear threshold model”. In: *2011 IEEE 11th international conference on data mining*. IEEE, pp. 211–220.
- Gustafson, John L (1988). “Reevaluating Amdahl’s law”. In: *Communications of the ACM* 31.5, pp. 532–533.
- Heule, Stefan, Marc Nunkesser, and Alex Hall (2013). “HyperLogLog in Practice: Algorithmic Engineering of a State of The Art Cardinality Estimation Algorithm”. In: *Proceedings of the EDBT 2013 Conference*. Genoa, Italy.

- Jung, Kyomin, Wooram Heo, and Wei Chen (2012). “Irie: Scalable and robust influence maximization in social networks”. In: *2012 IEEE 12th International Conference on Data Mining*. IEEE, pp. 918–923.
- Kempe, David, Jon Kleinberg, and Éva Tardos (2003). “Maximizing the spread of influence through a social network”. In: *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, pp. 137–146.
- Kim, Jinha, Seung-Keol Kim, and Hwanjo Yu (2013a). “Scalable and parallelizable processing of influence maximization for large-scale social networks?” In: *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE, pp. 266–277.
- Kim, Jinha, Seung-Keol Kim, and Hwanjo Yu (2013b). “Scalable and parallelizable processing of influence maximization for large-scale social networks?” In: *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE, pp. 266–277.
- Kimura, Masahiro, Kazumi Saito, and Ryohei Nakano (2007). “Extracting influential nodes for information diffusion on a social network”. In: *AAAI*. Vol. 7, pp. 1371–1376.
- Koopman, Philip (2002). “32-bit cyclic redundancy codes for internet applications”. In: *Proceedings International Conference on Dependable Systems and Networks*. IEEE, pp. 459–468.
- Krasnov, Vlad (2017). *On the dangers of Intel’s frequency scaling*. URL: <https://blog.cloudflare.com/on-the-dangers-of-intels-frequency-scaling/> (visited on 10/10/2020).
- Kumar, Rohit and Toon Calders (2017). “Information propagation in interaction networks”. In: *Advances in Database Technology-EDBT 2017*, pp. 270–281.
- Lemire, Daniel (2018). *AVX-512: when and how to use these new instructions*. URL: <https://lemire.me/blog/2018/09/07/avx-512-when-and-how-to-use-these-new-instructions/> (visited on 10/10/2020).
- Leskovec, Jure, Lada A Adamic, and Bernardo A Huberman (2007). “The dynamics of viral marketing”. In: *ACM Transactions on the Web (TWEB)* 1.1, p. 5.
- Leskovec, Jure and Andrej Krevl (June 2014). *SNAP Datasets: Stanford Large Network Dataset Collection*. <http://snap.stanford.edu/data>.
- Leskovec, Jure et al. (2009). “Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters”. In: *Internet Mathematics* 6.1, pp. 29–123.
- Lim, Yongsub, U Kang, and Christos Faloutsos (2014). “Slashburn: Graph compression and mining beyond caveman communities”. In: *IEEE Transactions on Knowledge and Data Engineering* 26.12, pp. 3077–3089.

- Liu, Qi et al. (2014). “Influence maximization over large-scale social networks: A bounded linear approach”. In: *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management*. ACM, pp. 171–180.
- Liu, Xiaodong et al. (2013). “IMGPU: GPU-accelerated influence maximization in large-scale social networks”. In: *IEEE Transactions on Parallel and distributed Systems* 25.1, pp. 136–145.
- Lü, Linyuan et al. (2012). “Recommender systems”. In: *Physics reports* 519.1, pp. 1–49.
- Matsumoto, Makoto and Takuji Nishimura (1998). “Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator”. In: *ACM Transactions on Modeling and Computer Simulation (TOMACS)* 8.1, pp. 3–30.
- Minutoli, Marco et al. (2019). “Fast and Scalable Implementations of Influence Maximization Algorithms”. In: *2019 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, pp. 1–12.
- Minutoli, Marco et al. (2020). “CuRipples: Influence Maximization on Multi-GPU Systems”. In: *Proceedings of the 34th ACM International Conference on Supercomputing*. ICS '20. Barcelona, Spain: Association for Computing Machinery. ISBN: 9781450379830. DOI: 10.1145/3392717.3392750. URL: <https://doi.org/10.1145/3392717.3392750>.
- Moreno, Yamir, Maziar Nekovee, and Amalio F Pacheco (2004). “Dynamics of rumor spreading in complex networks”. In: *Physical Review E* 69.6, p. 066130.
- Narayanam, Ramasuri and Yadati Narahari (2010). “A shapley value-based approach to discover influential nodes in social networks”. In: *IEEE Trans. on Automation Science and Engineering* 8.1, pp. 130–147.
- Nemhauser, George L, Laurence A Wolsey, and Marshall L Fisher (1978). “An analysis of approximations for maximizing submodular set functions—I”. In: *Mathematical programming* 14.1, pp. 265–294.
- Ohsaka, Naoto et al. (2014). “Fast and Accurate Influence Maximization on Large Networks with Pruned Monte-Carlo Simulations”. In: *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence*. AAAI'14. Québec City, Québec, Canada: AAAI Press, pp. 138–144.
- Peng, Zhen et al. (2018). “Graphphi: efficient parallel graph processing on emerging throughput-oriented architectures”. In: *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, pp. 1–14.
- Rosenfeld, Azriel and John L Pfaltz (1966). “Sequential operations in digital picture processing”. In: *Journal of the ACM (JACM)* 13.4, pp. 471–494.

- Sadeh, Gal, Edith Cohen, and Haim Kaplan (2020). “Sample Complexity Bounds for Influence Maximization”. In: *11th Innovations in Theoretical Computer Science Conference (ITCS 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.
- Sariyüce, Ahmet Erdem et al. (2014). “Hardware/Software Vectorization for Closeness Centrality on Multi-/Many-Core Architectures”. In: *2014 IEEE International Parallel & Distributed Processing Symposium Workshops, Phoenix, AZ, USA, May 19-23, 2014*, pp. 1386–1395. DOI: 10.1109/IPDPSW.2014.156.
- Sariyüce, Ahmet Erdem et al. (2015). “Regularizing graph centrality computations”. In: *J. Parallel Distributed Comput.* 76, pp. 106–119. DOI: 10.1016/j.jpdc.2014.07.006. URL: <https://doi.org/10.1016/j.jpdc.2014.07.006>.
- Shahrouz, Soheil, Saber Salehkaleybar, and Matin Hashemi (Mar. 2021). “gIM: GPU Accelerated RIS-based Influence Maximization Algorithm”. In: *IEEE Transactions on Parallel and Distributed Systems* PP, pp. 1–1. DOI: 10.1109/TPDS.2021.3066215.
- Shun, Julian and Guy E Blelloch (2013). “Ligra: a lightweight graph processing framework for shared memory”. In: *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pp. 135–146.
- Tang, Youze, Xiaokui Xiao, and Yanchen Shi (2014). “Influence Maximization: Near-Optimal Time Complexity Meets Practical Efficiency”. In: *CoRR* abs/1404.0900. arXiv: 1404.0900. URL: <http://arxiv.org/abs/1404.0900>.
- Thomson, W. E. (Jan. 1958). “A Modified Congruence Method of Generating Pseudo-random Numbers”. In: *The Computer Journal* 1.2, pp. 83–83. ISSN: 0010-4620. DOI: 10.1093/comjnl/1.2.83. eprint: <https://academic.oup.com/comjnl/article-pdf/1/2/83/1175362/010083.pdf>. URL: <https://doi.org/10.1093/comjnl/1.2.83>.
- Trusov, Michael, Randolph E Bucklin, and Koen Pauwels (2009). “Effects of word-of-mouth versus traditional marketing: findings from an internet social networking site”. In: *Journal of marketing* 73.5, pp. 90–102.
- Wei, Hao et al. (2016). “Speedup graph processing by graph ordering”. In: *Proceedings of the 2016 International Conference on Management of Data*, pp. 1813–1828.
- Wulf, Wm A and Sally A McKee (1995). “Hitting the memory wall: Implications of the obvious”. In: *ACM SIGARCH computer architecture news* 23.1, pp. 20–24.
- Zeng, Daniel et al. (2010). “Social media analytics and intelligence”. In: *IEEE Intelligent Sys.* 25.6, pp. 13–16.