GPU-based Parallel Algorithm for Generating Massive Scale-free Networks Using the Preferential Attachment Model

Maksudul Alam and Kalyan S. Perumalla
Oak Ridge National Laboratory
Oak Ridge, Tennessee, USA
{*alamm, perumallaks*}*@ornl.gov*

***Abstract*—A novel parallel algorithm is presented for generating random scale-free networks using the preferential-attachment model. The algorithm, named cuPPA, is custom-designed for single instruction multiple data (SIMD) style of parallel processing supported by modern processors such as graphical processing units (GPUs). To the best of our knowledge, our algorithm is the first to exploit GPUs, and also the fastest implementation available today, to generate scale-free networks using the preferential attachment model. A detailed performance study is presented to understand the scalability and runtime characteristics of the cuPPA algorithm. In one of the best cases, when executed on an NVidia GeForce 1080 GPU, cuPPA generates a scale-free network of two billion edges in less than 3 seconds.**

***Keywords*-GPU; Preferential–Attachment; Random Networks; Scale–Free Networks;**

I. Introduction

Networks are prevalent in many complex systems, such as circuits, chemical compounds, protein structures, biological networks, social networks, the Web, and XML documents. Recently, there has been substantial interest in the study of a variety of random networks to serve as mathematical models of complex systems. Various network theories, metrics, topology, and mathematical models have been proposed to understand the underlying properties and relationships of these systems. Among the proposed network models, the first and the most studied model is the Erdős-Rényi model [1]. However, the Erdős-Rényi model does not exhibit the characteristics observed in many real-world complex systems [2]. Barabási and Albert [2] discovered a class of inhomogeneous networks, called scale-free networks, characterized by a power-law degree distribution $P(k) \propto k^{-\gamma}$, where k represents the degree of a vertex and γ is a constant. While high degree vertices are improbable in Erdős-Rényi networks, they do occur with statistically significant probability in scale-free networks. Furthermore, the work of Albert et al. [3] suggests these high degree vertices appear to play an important role in the behavior of scale-free systems, particularly with respect to their resilience [4]. For example, the Barabasi-Albert model can be used for evaluating the North American electric grid with high reliability [4]. Additional related work is discussed later in Section V.

As these complex systems of today grow larger, the ability to generate progressively large random networks becomes all the more important. It is well known that the structure of larger networks is fundamentally different from that of small networks, and many patterns, such as communities, emerge only in massive networks [5]. Although various random network models have been used and studied over the last several decades, even efficient sequential algorithms for generating such networks were nonexistent until recently. The efficient sequential algorithms are able to generate networks with millions of edges in a reasonable amount of time, however, generating networks with billions of edges can take prohibitively large amount of time. This motivates the need for efficient parallel algorithms for generating such networks. Naïve parallelization of the sequential algorithms for generating random networks may not work due to the dependencies among the edges and the possibility of creating duplicate (parallel) edges.

Graphics processors (GPUs) are a cost-effective, energy-efficient, and widely available parallel processing platform. GPUs are highly parallel, multi-threaded, many-core processors that have greatly expanded beyond graphics operations and are now widely used for general purpose computing. The use of GPUs is prevalent in many areas such as scientific computation, complex simulations, big data analytics, machine learning, and data mining. However, there is a lack of GPU-based graph/network generators, especially for scale-free networks such as those based on the preferential attachment model. In this paper, we present **cuPPA**, a novel GPU based algorithm for generating networks conforming to the preferential attachment model. With cuPPA, one can generate a network with two billion edges using a modern NVidia GPU in less than three seconds. To the best of our knowledge, this is the first GPU-based algorithm to generate

networks using the exact preferential attachment model.

The rest of the report is organized as follows. In the following Section II, background material is provided in terms of preliminary information, notations, an outline of the network generation problem, and two leading sequential algorithms. In Section III, our parallel **cuPPA** algorithm for the GPU is presented. The experimental study and performance results using **cuPPA** are described in Section IV. We present a review of related works in Section V. Finally, Section VI concludes with a summary and an outline of future directions.

II. Background

A. Preliminaries and Notations

In the rest of this report, we use the following notations. We denote a network $G(V, E)$, where V and E are the sets of vertices and edges, respectively, with $m = |E|$ edges and $n = |V|$ vertices labeled as $0, 1, 2, \ldots, n-1$. For any $(u, v) \in E$, we say u and v are *neighbors* of each other. The set of all neighbors of $v \in V$ is denoted by $N(v)$, i.e., $N(v) = \{u \in V | (u, v) \in E\}$. The degree of v is $d_v = |N(v)|$. If u and v are neighbors, sometimes we say that u is *connected* to v and vice versa.

We develop parallel algorithms using the CUDA (Compute Unified Device Architecture) framework on the GPU. A GPU contains multiple streaming multiprocessors (SMs). An SM is a group of core processors. Each core processor executes only one thread at a time. All core processors can execute their corresponding threads simultaneously. If some threads perform operations that have to wait for data fetches with high latencies, those are put into the waiting state and other pending threads are executed. Therefore, GPUs increase throughput by keeping the processors busy. All thread management, including the creation and scheduling of threads, is performed entirely in hardware with virtually zero overhead and requires negligible time for launching work on the GPU. For these advantages, modern supercomputers, such as Titan, the largest supercomputer in the USA, are build using GPUs in addition to conventional central processing units (CPUs).

We use **K**, **M**, and **B** to denote thousand, million, and billion, respectively; e.g., 2 **B** stands for two billion.

B. Preferential Attachment–Based Models

The preferential attachment model is a model for generating randomly evolved scale-free networks using a preferential attachment mechanism. In a preferential attachment mechanism, a new vertex is added to the network and connected to some existing vertices that are chosen preferentially based on some properties of the vertices. In the most common method, preference is given to vertices with larger degrees: the higher the degree of a vertex, the higher is the probability of choosing it. In this report, we study only the degree-based preferential attachment, and in the rest of the report, by preferential attachment (PA) we mean degree-based preferential attachment.

Before presenting our parallel algorithms for generating PA networks, we briefly discuss the sequential algorithms for the same. Many preferential attachment based models have been proposed in the literature. Two of the most prominent models are the Barabási–Albert model [2] and the copy model [6] as discussed below.

C. Sequential Algorithm: Barabási-Albert Model

One way to generate a random PA network is to use the Barabási-Albert (BA) model. Many real-world networks have two important characteristics: (*i*) they are evolving in nature and (*ii*) the network tends to be scale-free [2]. In the BA model, a new vertex is connected to an existing vertex that is chosen with probability directly proportional to the current degree of the existing vertex.

The BA model works as follows. Starting with a small clique of $\hat{d}$ vertices, in every time step, a new vertex t is added to the network and connected to $d \leq \hat{d}$ randomly chosen existing vertices: $F_\ell(t)$ for $1 \leq \ell \leq d$ with $F_\ell(t) < t$; that is, $F_\ell(t)$ denotes the ℓ-th vertex which t is connected. Thus, each phase adds d new edges $(t, F_1(t)), (t, F_2(t)), \ldots, (t, F_d(t))$ to the network, which exhibits the evolving nature of the model. Let $\mathbb{F}(t) = \{F_1(t), F_2(t), \ldots, F_d(t)\}$ be the set of outgoing vertices from t. Each of the d end points in the set $\mathbb{F}(t)$ are randomly selected based on the degrees of the vertices in the current network. In particular, the probability $P_i(t)$ that a outgoing edge from vertex t is connected to vertex $i < t$ is given by $P_i(t) = \frac{d_i}{\sum_j d_j}$, where d_j represents the degree of vertex j.

The networks generated by the BA model are called the BA networks, which bear the aforementioned two characteristics of a real-world network. BA networks have power-law degree distribution. A degree distribution is called power-law if the probability that a vertex has degree d is given by $\Pr[d] \propto d^{-\gamma}$, where $\gamma \geq 1$ is a positive constant. Barabási and Albert showed that the preferential attachment method of selecting vertices results in a power-law degree distribution [2].

A naïve implementation of network generation based on the BA model takes $\Omega(n^2)$ time where n is the number of vertices. Batagelj and Brandes give an efficient algorithm with a running time of $\mathcal{O}(m)$ where m is the number of edges [7]. This algorithm maintains a list of vertices such that each vertex i appears in this list exactly d_i times. The list can easily be updated dynamically by simply appending u and v to the list whenever a new edge (u, v) is added to the network. Now, to find $F(t)$, a vertex is chosen from the list uniformly at random. Since each vertex i occurs exactly d_i times in the list, we have the probability $\Pr[F(t) = i] = \frac{d_i}{\sum_j d_j}$.

D. Sequential Algorithm: Copy Model

As it turns out, the BA model does not easily lend itself to an efficient parallelization [8]. Another algorithm called the *copy model* [6, 9] preserves preferential attachment and power-law degree distribution. The copy model works as follows. Similar to the BA model, it starts with a small clique of $\hat{d}$ vertices and in every time step, a new vertex t is added to the network to create $d \leq \hat{d}$ connections to existing vertices $F_\ell(t)$ for $1 \leq \ell \leq d$ with $F_\ell(t) < t$. For each connection $(t, F_\ell(t))$ from vertex t the following steps are executed:

Step 1: First a random vertex $k \in [0, t-1]$ is chosen with uniform probability.

Step 2: Then $F_\ell(t)$ is determined as follows:

$$F_\ell(t) = k \text{ with prob. } p \qquad \text{(Direct edge)} \quad (1)$$
$$= F_l(k) \text{ with prob. } (1-p) \qquad \text{(Copy edge)} \quad (2)$$

where l is a random outgoing connection from vertex k. We also denote $\mathbb{F}(t) = \{F_1(t), F_2(t), \ldots, F_d(t)\}$ to be the set of outgoing vertices from vertex t.

It can be easily shown that a connection from vertex t to vertex i is made with probability $\Pr[i \in \mathbb{F}(t)] = \frac{d_i}{\sum_j d_j}$ when $p = \frac{1}{2}$. Thus, when $p = \frac{1}{2}$, this algorithm follows the Barabási-Albert model as shown in Theorem 1 [8, 10].

Theorem 1. *The Barabási-Albert model is a special case of the copy model when $p = \frac{1}{2}$.*

Proof: A vertex i can be selected in $\mathbb{F}(t)$ in two mutually exclusive ways: i) i is chosen in the first step and assigned to an outgoing edge of t in the second step (Equation 1); this event occurs with probability $\frac{1}{t} \cdot p$; or ii) a neighbor of i, $v \in \{u | i \in \mathbb{F}(u)\}$, is chosen in the first step, and the outgoing edge to i is selected (out of d outgoing edges from v) in the second step (Equation 2); this event occurs with probability $\frac{d_i - d}{t} \cdot (1-p) \cdot \frac{1}{d}$ where d_i is the total degree of vertex i. Thus, we have the following equation.

$$\begin{aligned} \Pr[i \in \mathbb{F}(t)] &= \frac{p}{t} + \frac{d_i - d}{dt} \cdot (1-p) \\ &= \frac{dp + (d_i - d)(1-p)}{dt} \\ &= \frac{dp + (d_i - d)(1-p)}{\frac{1}{2}\sum_j d_j} \end{aligned} \quad (3)$$

When $p = \frac{1}{2}$, $\Pr[i \in \mathbb{F}(t)] = \frac{d_i}{\sum_j d_j}$. ∎

Thus, the copy model is more general than the BA model. It has been previously shown [6] that the copy model produces networks with degree distribution that follows a power-law $d^{-\gamma}$, where the value of the exponent γ depends on the choice of p. Further, it is easy to see the running time of the copy model is $\mathcal{O}(m)$. Copy model has been used to develop efficient parallel algorithms for generating preferential attachment networks in distributed and shared-memory machines [8, 11]. In our work presented in this report, we adopt the copy model as a starting point to design and develop our GPU-based parallel algorithm.

III. GPU–BASED PARALLEL ALGORITHM: CUPPA

The PA model imposes a critical dependency that every new vertex needs to have the state of the previous network to compute its edges. This poses a major challenge in parallelizing preferential attachment algorithms. In phase v, to determine $F(v)$, it requires that F_i is known for each $i < v$. As a result, any algorithm for preferential attachment apparently seems to be highly sequential in nature: phase v cannot be executed until all previous phases are completed.

In [8], a distributed–memory based algorithm was proposed that exploits the copy model to relieve this sequentiality and run in parallel. We reexamined that exploitation and designed **cuPPA**, an efficient parallel algorithm for generating preferential attachment based networks on the GPU as described next. Let T be the number of threads in the GPU. The set of vertices V is partitioned into T disjoint subsets of vertices $V_0, V_1, \ldots, V_{T-1}$; that is, $V_i \subset V$, such that for any i and j, $V_i \cap V_j = \emptyset$ and $\bigcup_i V_i = V$. The graph is stored entirely in the GPU memory. Thread $\mathcal{T}_i$ is responsible for computing and updating $F(v)$ for all $v \in V_i$. The algorithm starts with an initial network, which is a clique of the first d vertices labeled $0, 1, 2, \ldots, d-1$. For each vertex v, the algorithm computes d edges $(t, F_1(v)), (t, F_2(v)), \ldots, (t, F_d(v))$ and ensure that such edges are distinct without any parallel edges. We denote the set of vertices $\{F_1(v), F_2(v), \ldots, F_d(v)\}$ by $\mathbb{F}(v)$. The algorithm works in two phases. In the first phase of the algorithm (called *Execute Copy Model*), we execute the copy model for all vertices in parallel (using all threads). This phase creates all the direct edges and some of the "copy" edges (Equation 2). However, many copy edges might not be fully processed due to the dependencies. The incomplete copy edges are put in a waiting queue called $\mathcal{Q}$. In the second phase of the algorithm (called *Resolve Implete Edges*), we resolve the incomplete edges from the waiting queue $\mathcal{Q}$ and finalize the copy edges. The pseudocode of cuPPA is given in Algorithm 1.

In the first phase (Line 3–21) the algorithm executes the copy model for all of its vertices. The edges that could not be completed are stored in a queue $\mathcal{Q}$ to be processed later. We call the queue a waiting queue. Each of the other vertices from d to $n-1$ generates d new edges. There are fundamentally two important issues that need to be handled: i) how we select $F_\ell(v)$ for vertex v where $1 \leq \ell \leq d$, and ii) how we avoid duplicate edge creation. Multiple edges for a vertex v are created by repeating the same procedure d times (Line 4), and duplicate edges are avoided by simply checking if such an edge already exists – such a check is done whenever a new edge is created.

For the ℓ-th edge of a vertex v, another vertex u is chosen from $[1, v-1]$ uniformly at random (Line 5, 6). Edge (v, u) is

Algorithm 1: cuPPA

n	Number of vertices
d	Number of outgoing edges from each vertex
p	Probability of creating a direct edge
V_i	The set of vertices processed by thread $\mathcal{T}_i$
$\mathbb{F}(u)$	The set of outgoing edges from vertex u
$F_i(u)$	The i-th outgoing edge from vertex u
$\mathcal{Q}$	A queue for the current set of unfinished edges
$\mathcal{Q}'$	A queue for the next set of unfinished edges

1 **with** T *threads* **do in parallel** `/* Each thread` $\mathcal{T}_i$ `executes the following in parallel: */`

`// Phase 1: Execute Copy Model`

3 **foreach** $v \in V_i$ **do**
4 **for** $\ell = 1$ *to* d **do**
5 $u \leftarrow$ a uniform random vertex in $[0, v-1]$
6 $c \leftarrow$ a uniform random number in $[0, 1]$
7 **if** $c < p$ **then** `// i.e., with prob.` p
8 **if** $u \notin \mathbb{F}(v)$ **then**
9 $F_\ell(v) \leftarrow u$
10 **else**
11 go to line 5
12 **else**
13 $l \leftarrow$ a uniform random integer in $[1, d]$
14 **if** $F_l(u) \neq NULL$ **then** `// Resolved`
15 **if** $F_l(u) \notin \mathbb{F}(v)$ **then**
16 $F_\ell(v) \leftarrow F_l(u)$
17 **else**
18 go to line 5
19 **else** `// Unresolved edge into` $\mathcal{Q}$
20 $F_\ell(v) \leftarrow NULL$
21 Add $\langle u, l \rangle$ to $\mathcal{Q}$

`// Phase 2: Resolve Incomplete Edges`

22 **while** $\mathcal{Q} \neq \emptyset$ **do**
23 **foreach** $\langle u, l \rangle \in \mathcal{Q}$ **do**
24 **if** $F_l(u) \neq NULL$ **then**
25 $F_\ell(v) = F_l(u)$
26 **else**
27 Append $\langle u, l \rangle$ to $\mathcal{Q}'$
28 Swap $\mathcal{Q}$ and $\mathcal{Q}'$
29 $\mathcal{Q}' \leftarrow \emptyset$

created with probability p (Line 7). However, before creating such an edge (v, u) in Line 8, the existence of such an edge is checked immediately before creating them in Line 9. If the edge already exists at that time, the edge is discarded and the process is repeated again (Line 5). With the remaining $1 - p$ probability, v is connected to some vertex in $\mathbb{F}(u)$; that is, we make an edge $(v, F_\ell(u))$, such that ℓ is chosen from $[1, d]$ uniformly at random.

After the first phase is completed, the algorithm starts to resolve all incomplete edges by processing the waiting queue (Lines 22–29). If an item in the current queue $\mathcal{Q}$ could not be resolved during this step, it is subsequently placed in another queue $\mathcal{Q}'$. After all incomplete edges on the queue $\mathcal{Q}$ are processed, the queues $\mathcal{Q}$ and $\mathcal{Q}'$ are swapped and $\mathcal{Q}'$ is cleared. We repeat this process until both the queues are empty.

A. Graph Representation

We use one array G of nd elements to represent and store the entire graph. Each vertex u connects to d existing vertices. The neighbors of u are stored between the indices inclusive from ud to $(u+1)d-1$ that represents the other end-point vertices. We call these indices the *outgoing vertex list* for vertex u. The initial network consists of the d^2 vertices from the start of the array. For any edge u, v where $u > v$ and $u, v > d$, the edge is represented by storing v in one of the d items in the *outgoint vertex list* of u. Note that the graph G contains exactly nd edges as defined by the Barabási–Albert or the copy model. Any vertex withe the index $0 \leq i < nd$ of the array G denotes the $(t \mod d)$-th end-point of the vertex $\frac{i}{d}$.

B. Partitioning and Load Balancing

Recall that we distribute the computation among the threads by partitioning the set of vertices $V = \{0, 1, \ldots, n-1\}$ into T subsets $V_0, V_1, \ldots, V_{T-1}$ as described at the beginning of Section III, where T is the number of available threads. Although several partitioning schemes are possible, our study suggests that the *Round Robin Partitioning* (RRP) scheme best suits our algorithm. In this scheme, vertices are distributed in a round robin fashion among all threads. Partition V_i contains the vertices $\langle i, i+T, i+2T, \ldots, i+kT \rangle$ such that $i + kT \leq n < i + (k+1)T$; that is, $V_i = \{j | j \mod T = i\}$. In other words, vertex i is assigned to set $V_{i \mod T}$. Therefore, the number of vertices in the sets are almost equal., i.e., the number of vertices in a set is either $\lceil \frac{n}{T} \rceil$ or $\lfloor \frac{n}{T} \rfloor$. The round robin partitioning scheme is illustrated in Figure 1.

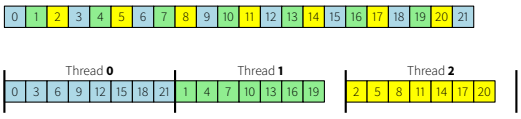

Figure 1: *Distributing* 21 *vertices among* 3 *threads using round robin partitioning.*

C. Segmented Round Robin Paritioning

However, the naïve round robin scheme discussed above also has some technical issues. As described in Section III, the first phase of the Algorihm 1 executes the copy model for every vertex assigned to it and stores any unresolved copy edge in the waiting queue. In the second phase, the algorithm takes out each unresolved edge from the waiting

queue and tries to resolve them. To reduce the memory latency accessing the waiting queue, we store the waiting queue $\mathcal{Q}$ in the GPU *shared memory* that offers many folds faster memory access than the global GPU memory. Note that this memory is limited in capacity and is shared among all threads running within the same block. Modern GPUs such as NVidia GeForce 1080 have 48 KB of ultra-fast shared memory per block. Since the amount of the shared memory is very limited, it can only store a limited number of unresolved items in the queue. Let $\mathcal{C}$ denotes the total capacity of the waiting queue. For example, with a 48 KB of shared memory, we have a total capacity to store $\mathcal{C} = \frac{48 \times 1024}{8} = 6144$ items in the waiting queue where each item takes 8 bytes of memory. If we use τ threads per block, each thread will have a capacity of $\frac{\mathcal{C}}{\tau}$ items to be placed in the waiting queue. Therefore, if the number of vertices assigned to a thread is too large, it may generate a large number of unresolved copy edges to be placed in the waiting queue, essentially forcing the algorithm to use a large amount of GPU memory instead of the available shared memory.

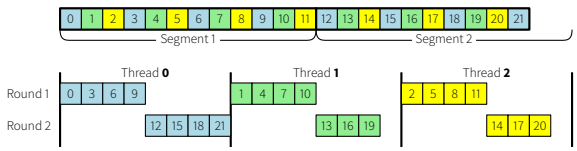

Figure 2: *Distributing 21 vertices among 3 threads using segmented round robin partitioning with 2 rounds.*

In order to exploit the faster shared memory without overflowing the waiting queue capacity, we use a modified round robin partitioning scheme called, *Segmented Round Robin Partitioning* (SRRP). In this scheme, the entire set of vertices V is first partitioned into some k consecutive subsets $S_1, S_2, S_3 \ldots S_k$ called *segments*. From the definition of the copy model, it is clear that vertices on a segment S_i may only depend on vertices on segment S_j where $i \geq j$ but not vice versa. Therefore, the segments have to be processed in a consecutive fashion. Let $B_i = |S_i|$ denotes the number of elements (also called the segment size) in segment S_i where $1 \leq i \leq k$. Next, the parallel algorithm is executed in k consecutive rounds where round i executes the parallel algorithm for all the vertices in segment S_i. In round i, the B_i vertices in segment S_i are further partitioned into T subsets $V_0(S_i), V_1(S_i), \ldots V_{T-1}(S_i)$, using the round robin scheme discussed above and executed in parallel using the T threads. The technique is illustrated in Figure 2.

Next, we need to determine the best segment size to avoid overflow while using the shared memory. From the copy model, it is easy to see that the lower the probability p is, the more likely it is to be in the waiting queue. In the worst case, when $p = 0$, all generated edges consist of copy edges. Therefore, at most d unresolved copy edges could be placed in the waiting queue per vertex. Additionally, as the value of d gets bigger, the number of copy edges increases and hence, the waiting queue size increases. Therefore, p and d both have a significant impact on the required size of the waiting queue. Having that in mind, we use two approaches for the segment size:

- **Fixed Segment Size:** The simplest way is to use a fixed sized segments in each round. From the previous discussion, it is clear that in the worst case we need d items per vertex to be placed on the waiting queue. Therefore, we can use up to $\tau = \min\left(\frac{\mathcal{C}}{d}, \theta\right)$ threads per block where $\mathcal{C}$ is the total queue capacity and θ is the maximum number of threads per block. Then the segment size is $\frac{\mathcal{C}}{d\tau}$ vertices per segment. Note that we can exploit the shared memory for $d \leq \mathcal{C}$, otherwise we need to use the global memory. However, in almost all practical scenarios we have $d \ll \mathcal{C}$, hence, we can take advantages of the shared memory.
- **Dynamic Segment Size:** Although the fixed segment size scheme ensures that the queue will not overflow in any round, it may not be the most efficient implementation. We use another scheme where the segment size is determined dynamically between two rounds based on the current state of the algorithm. In this scheme, we start with the number of threads per block τ and the segment size $\frac{\mathcal{C}}{d\tau}$ vertices per segment as was done in the Fixed Segment Size scheme. However, at the end of each round, we determine the maximum number of items that were placed in the waiting queue per thread. If the number of items placed in the waiting queue in the round is less than some f factor of the waiting queue capacity per thread $\frac{\mathcal{C}}{\tau}$, we increase the total capacity $\mathcal{C}$ by a factor of f (typically, we set $f = 2$). Before the next round, we recompute the required number of threads per block and update the segment size accordingly.

D. CUDA-Specific Deadlock Scenario

In the round robin scheme, completion of a copy edge of a vertex in a thread $\mathcal{T}_i$ may depend on some other thread $\mathcal{T}_j$ where $i \neq j$. Due to the nature of dependency, $\mathcal{T}_j$ also may have a copy edge that depends on another vertex that belongs to $\mathcal{T}_i$. Therefore, if any of these threads are not running simultaneously on the GPU, the other thread will not be able to complete and a deadlock situation may arise. To avoid such a situation, we must ensure that either all the GPU threads are running concurrently or the dependent threads are put to sleep for a while. In the current CUDA framework, the runtime engine schedules each kernel block to a streaming multiprocessor, and the blocks of running threads are non-preemptible. Therefore, to ensure that threads are running concurrently to avoid deadlock situation, we cannot use more blocks than the number of available streaming multiprocessors. Note that the upcoming CUDA runtime

supports *cooperative groups*. On such future systems, the deadlock situation could be avoided using block sizes larger than the number of shared multiprocessors *.

IV. Experimental Results

In this section, we evaluate our algorithm and its performance by experimental analysis. We demonstrate the accuracy of our algorithm by showing that our algorithm produces networks with power-law degree distribution as desired. We also compare the runtime of our algorithm using several sequential and parallel algorithms.

A. Hardware and Software

We used a computer consisting of 24 AMD Opteron(tm) 6174 processor with an 800 MHz clock speed and 64GB system memory. The machine also incorporates a NVidia 1080 GPU with 8GB memory. The operating system is `Ubuntu 16.04 LTS`, and all software on this machine was compiled with GNU `gcc 4.6.3` with optimization flags `-O3`. The `CUDA compilation tools V8` were used for the GPU code along with `nvcc` compiler.

B. Degree Distribution

To demonstrate the accuracy of cuPPA, we compared it with the Sequential Barabási–Albert (SBA) [7] and the Sequential Copy Model (SCM) algorithms. The degree distributions of the networks generated by SBA, SCM, and cuPPA are shown in Figure 3 in a log-log scale. We used $n = 500M$ vertices, each generating $d = 4$ new edges with a total of two billion ($2{\times}10^9$) edges. The distribution is heavy-tailed, which is a distinct feature of the power-law networks. The exponent γ of the power-law degree distribution is measured to be 2.7. This supports the fact that for the finite average degree of a scale-free network, the exponent should be $2 < \gamma < \infty$ [12]. Also notice that the degree distributions of SBA and SCM are quite identical, experimentally verifying Theorem 1. The degree distribution of cuPPA is also similar to both SBA and SCM.

C. Visualization of Generated Graphs

In order to gain an idea of the structure and degree distributions, we obtained a visualization of some of the networks generated by our algorithm. We generated the visualizations using a popular network visualization tool called `Gephi`. Bearing aesthetics in mind and to minimize undue clutter, we focused on a few small networks by choosing $n = 10000$, $p = 0.5$, and $d = 1, 2, 4$. The visualizations are shown in Figures 4 to 6.

*https://devblogs.nvidia.com/parallelforall/cuda-9-features-revealed/

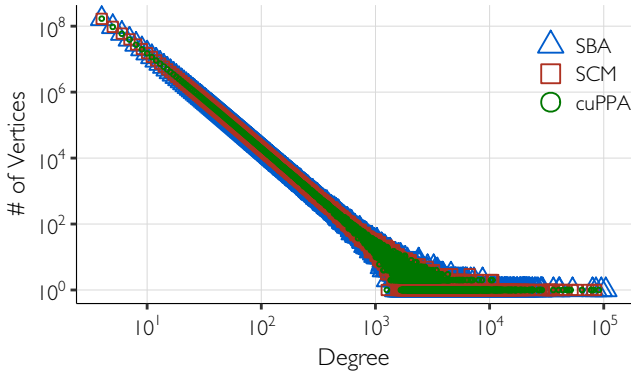

Figure 3: *The degree distributions of the PA Networks ($n = 500M$, $d = 4$). In* log-log *scale the degree distribution is a straight line validating the scale-free property. Further, all three models produce almost identical degree distributions showing that cuPPA produces networks with accurate degree distributions.*

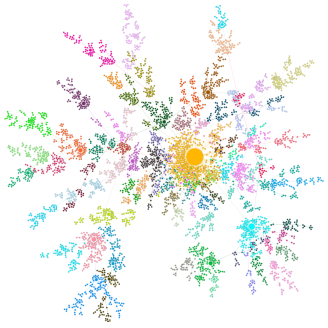

Figure 4: *Visualization of networks generated by cuPPA using $n = 10000$, $p = 0.5$ and $d = 1$.*

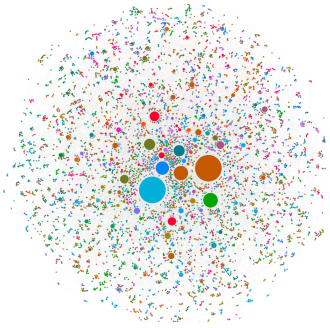

Figure 5: *Visualization of networks generated by cuPPA using $n = 10000$, $p = 0.5$ and $d = 2$.*

D. Effect of Edge Probability on Degree Distribution

As mentioned earlier, the strength of the copy model is the capability of generating other preferential attachment networks by simply varying one parameter, namely, the probability p. In Figure 7 we display the degree distribution of the generated networks by varying p. When $p = 0$, all

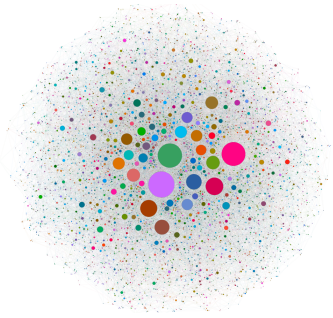

Figure 6: *Visualization of networks generated by cuPPA using $n = 10000$, $p = 0.5$ and $d = 4$.*

edges are produced by copy edges, and thus the network becomes a star network where all additional vertices connect to the d initial vertices. With a small value of p ($p = 0.01$), we can generate a network with a very long tail. When we set $p = 0.5$, we get the Barabási–Albert networks which exhibit a straight line in log–log scale. When we increase p to 1, we get a network consists entirely of direct edges that do not form any tail.

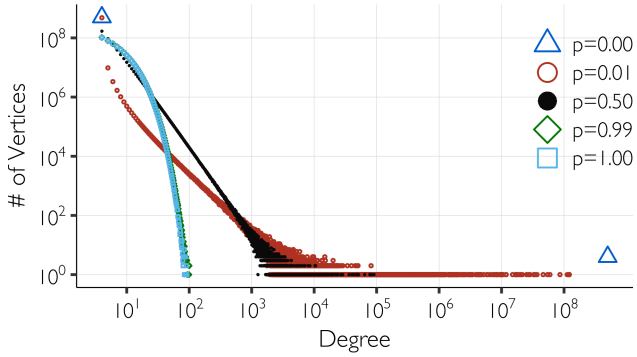

Figure 7: *The degree distributions of the networks by cuPPA ($n = 250M$, $d = 4$) with varying p.*

E. Waiting Queue Size

As mentioned in Section III-C, the waiting queue size depends on p and d. To evaluate the impact of p and d, we ran simulations using 1280 CUDA threads (20 blocks and 64 threads per block) where each thread only executed one vertex. The value of p is varied from 0 to 1 with different probability values. We also varied the value of d from 1 to 4096 as increasing powers of 2. In Figure 8, we show the number of items placed in the waiting queue per vertex for different combinations of p and d. We also added the worst case value as a line in the plot. As seen from the figure, in the worst case with $p = 0$, the maximum size of the waiting queue increases linearly with d for smaller values of d (up to 64) and afterward, it does not increase much compared to d. Therefore, for smaller values of d we need to have provisions for at least d items per vertex in the waiting queue.

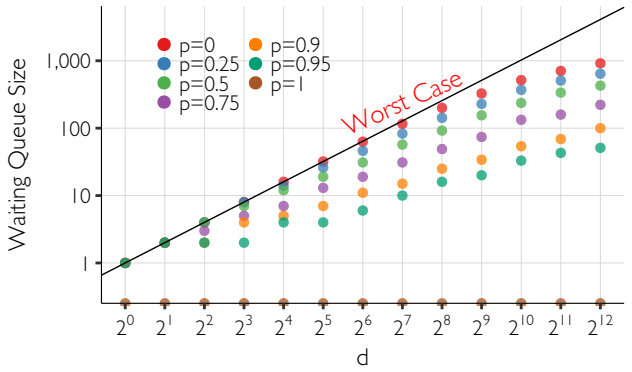

Figure 8: *Maximum size of the waiting queue per thread for different values of p and d (both axes in* log *scale). In the worst case ($p = 0$) the maximum size increases linearly with d for smaller values ($d \leq 64$). For larger d, the actual maximum size of waiting queue is comparatively smaller than the worst case.*

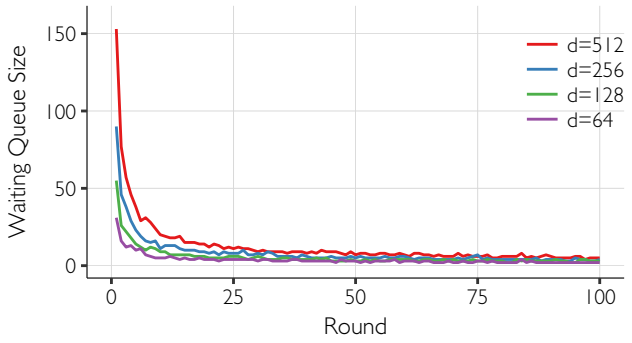

Figure 9: *Size of the waiting queue decreases significantly with rounds in SRRP scheme.*

However, as the round progresses, the maximum size of the waiting queue decreases significantly as shown in Figure 9. For this figure, we also ran cuPPA using 1280 CUDA threads (20 blocks and 64 threads per block) to generate networks with $d = 512, 256, 128, 64$ and $p = 0.5$. Each CUDA thread processes exactly one vertex per round. Only the first 100 rounds are shown for brevity. From Figure 9, we can see that as the round progresses, the size of the waiting queue per round decreases dramatically for all different values of d. This indicates that we could process more vertices in later rounds using the same amount of queue memory. Therefore, we can dynamically change the size of the segments between two consecutive rounds to increase parallelization. Based on these observations regarding the size of the waiting queue, we designed an adaptive version of cuPPA that monitors the maximum size of the waiting queue

and manages the segment size accordingly as discussed in Section III-C. We call this version **cuPPA-Dynamic** and use it for all other experiments.

F. Runtime Performance

In this section, we analyze the runtime and performance of cuPPA relative to other algorithms and show the variation of performances against various parameters.

1) Runtime Comparison with Existing Algorithms: To the best of our knowledge, our algorithm is the first GPU–based parallel algorithm to generate preferential attachment networks. Therefore, it is not possible to compare the runtime with other GPU–based algorithms. Instead, we compare with the existing non–GPU algorithms. The runtimes of cuPPA and the existing algorithms are shown in Figure 10 for generating two billion edges ($n = 500M$, $d = 4$).

- **Sequential Algorithms:** We compare cuPPA with two efficient sequential algorithms: SBA [7] and SCM [6]. As shown in Figure 10, cuPPA generates the network in just 2.72 seconds in the NVidia 1080 GPU with more than $100\times$ speedup.
- **Parallel Algorithms:** We also compared cuPPA with a distributed–memory (PPA-DM) [8] and a shared–memory (PPA-SM) [11] parallel algorithms. As shown in Figure 10, cuPPA outperforms PPA-DM on a system with 24 processors. The main reason is that unlike PPA-DM, cuPPA does not require complex synchronizations and message communications.
  Due to the unavailability of the PPA-SM code, we compared the runtime to generate the largest graph ($n = 10^7, d = 10$) reported in [11] with the corresponding runtime of cuPPA. PPA-SM generates the network using 16 cores of Intel Xeon CPU E5-2698 2.30 GHz in approximately 7.5 seconds, whereas cuPPA generates the same network in just 0.3 second.

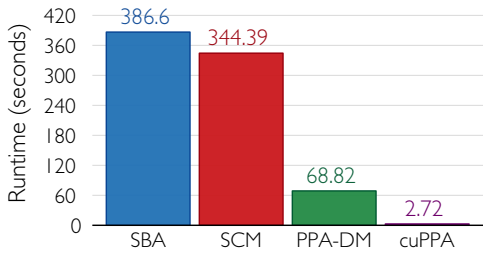

Figure 10: *Runtimes of SBA, SCM, PPA-DM, and cuPPA for generating two billion edges ($n = 500M$, $d = 4$). cuPPA can generate the network in less than three seconds.*

2) Runtime vs. Number of Vertices: First we examine the runtime performance of cuPPA with increasing number of vertices n. Here we examine two cases. In the first case we set $d = 4$, vary $p = \{0, 0.001, 0.25, 0.5, 0.75, 1\}$, and vary $n = \{1.9M, 3.9M, 7.8M, 15.6M, 31.25M, 62.5M, 125M, 250M, 500M\}$ to see how the runtime changes with increasing number of vertices for different p. The corresponding runtime is shown in Figure 11. In the second case, we set $p = 0.5$, vary $d = \{1, 2, 4, 8, 16, 32, 64, 128\}$, and vary $n = \{60K, 120K, 240K, 480K, 960K, 1.92M, 3.84M, 7.68M\}$ to see how the runtime changes with increasing number of vertices for different d. The corresponding runtime is shown in Figure 12.

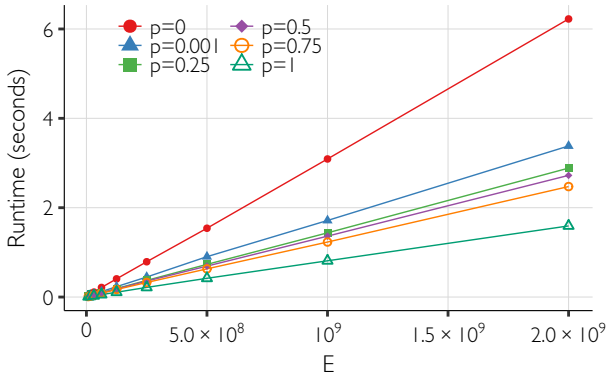

Figure 11: *Runtime vs. number of edges suggests that cuPPA is very scalable with increasing n for different values of p with a fixed value of $d = 4$.*

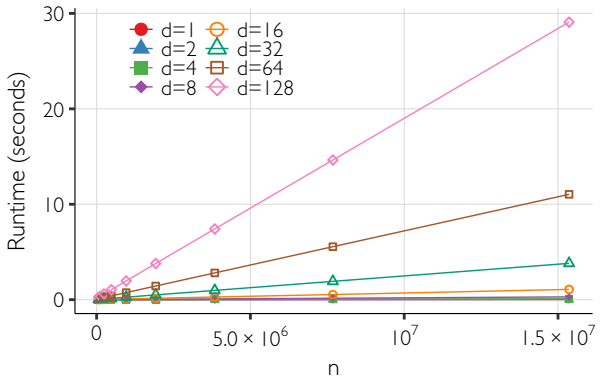

Figure 12: *Runtime vs. number of vertices suggests that cuPPA is very scalable with increasing n for different values of d with a fixed value of $p = 0.5$.*

From Figures 11 and 12, we can observe that for any fixed set of values for p and d, with increasing n, the runtime increases linearly, indicating that the algorithm scales very well with increasing value of n.

3) Runtime vs. Degree of Preferential Attachment: Next, we examine the runtime performance of cuPPA with increasing d. The runtime is shown in Figure 13. Here, we set $n = 7812500$, vary $p = \{0, 0.00001, 0.001, 0.25, 0.5, 0.75, 1\}$, and vary $d = \{1, 2, 4, 8, 16, 32, 64, 128\}$ to see how the runtime changes for increasing value of d for different p. As seen from the figure, with increasing d, the runtime increases

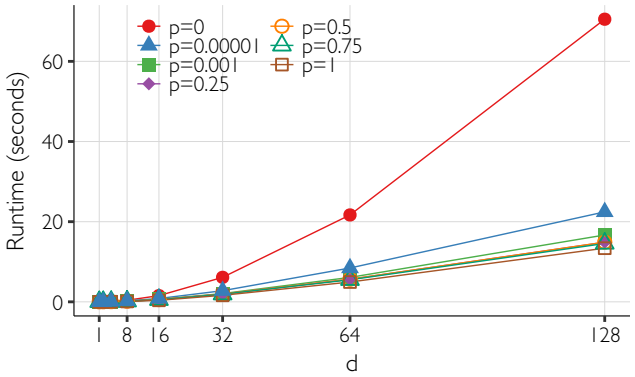

Figure 13: *Runtime vs. d for generating networks with $n = 7812500$ with varying $d = 1, 2, 4, 8, 16, 32, 64, 128$ for different values of p. The runtime almost increases linearly.*

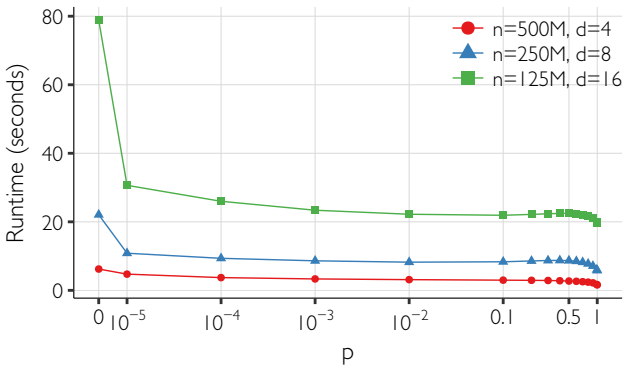

Figure 14: *Runtime vs. p for three sets of values for n and d (x-axis in log scale). At $p = 0$ the runtime is the largest which reduces significantly with a slight increase. As p increases the runtime reduces.*

almost linearly. Therefore the algorithm is observed to scale well for increasing value of d. Note that higher values of d are typically unlikely. However, we included higher values of d for performance measurement purpose. Also notice that the runtime is the largest for $p = 0$. With a small value of $p = 0.00001$ the runtime drops significantly and does not change much for higher values of p. Since the typical values of p are much larger than 0, this observation suggests that cuPPA performs well for real world scenarios.

4) Runtime vs. Probability of Copy-Edge: Next we examine the runtime performance of cuPPA with increasing p. The runtime is shown in Figure 14. Here, we used three different set of values for n and d ($\langle n = 500M, d = 4\rangle$, $\langle n = 125M, d = 16\rangle$, and $\langle n = 31.25M, d = 64\rangle$), and vary $p = \{0, 0.00001, 0.0001, 0.001, 0.01, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 0.99, 1.00\}$. As seen from the figure, the runtime reduces dramatically with a slight increase of $p = 0$ to $p = 0.00001$ up to $p = 0.1$ in all of these three cases. Then the runtime reduces almost linearly up to $p = 0.9$ and then reduces sharply towards $p = 1$. With lower values of p, most of the edges are produced by copy edges. Therefore, the size of the waiting queue increases, thereby increasing the runtime. As the value of p increases towards 1, most of the edges are created using direct edges and, therefore, fewer items are stored in the waiting queue.

5) Runtime varied with the number of Threads: To understand how the performance of cuPPA depends on the number of threads, we set $p = 0.5$ and used four different sets of n and d to generate networks. We also varied the number of CUDA threads per block from $64, 128, 256, 512$, to 1024. The runtime (solid lines) and the relative speedup (dashed lines) of the experiments are shown in Figure 15. Let t_T be the runtime of cuPPA using T threads. Then, the relative speedup is defined as $\frac{t_T}{t_{64}}$ in this experiments, i.e., the speedup gained compared to the runtime of cuPPA using 64 threads. Figure 15 is shown in two y-axes, the left and right axis correspond to the runtime and relative speedup respectively. From the figure, the best performance is observed with 512 threads per block for all cases. Therefore, in our final algorithm, we use up to 512 threads per block.

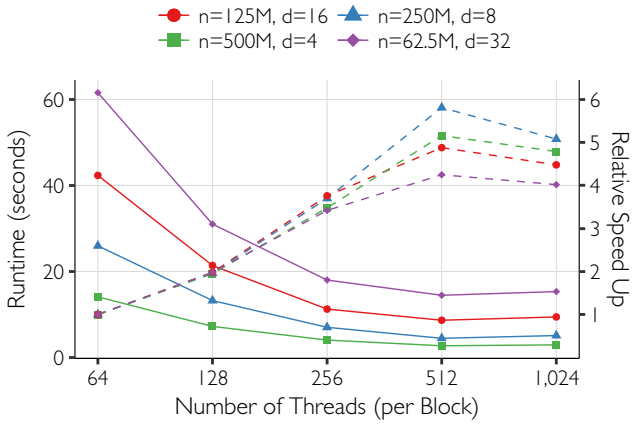

Figure 15: *Runtime (solid lines) and Relative Speedup (dashed lines) vs. Number of Threads. Best performance is observed with 512 threads per block.*

V. Related Work

Although the concepts of random networks have been used and well studied over the last several decades, efficient algorithms to generate the networks were not available until recently. The first efficient sequential algorithm to generate Erdős–Rényi and Barabási–Albert networks was proposed in [7]. A distributed memory–based algorithm to generate preferential attachment networks was proposed in [13]. However, their algorithm was not exact, rather an approximate algorithm and required manually adjusting several control parameters. The first exact distributed memory–based parallel algorithm using the copy model was proposed

in [8]. Another distributed memory–based parallel algorithm using the Barabási–Albert model was proposed in [14, 15]. However, instead of using pseudorandom number generators, they used hash functions to generate the networks. A shared–memory-based parallel algorithm using the copy model was proposed in [11]. A GPU–based algorithm for generating Erdős–Rényi networks was presented in [16]. Another GPU–based algorithm for generating networks using the small–world model [17] was presented in [18]. However, no GPU–based algorithm exists for the preferential attachment model.

Several other theoretical studies were done on the preferential attachment-based models. Machta and Machta [19] described how an evolving network can be generated in parallel. Dorogovtsev et al. [20] proposed a model that can generate graphs with fat-tailed degree distributions. In this model, starting with some random graphs, edges are randomly rewired according to some preferential choices.

VI. Conclusion

A novel GPU-based algorithm, named **cuPPA**, has been presented, with a detailed performance study, and its combination of its scale and speed has been tested by achieving the ability to generate networks with up to two billion edges in under three seconds of wall clock time. The algorithm is customizable with respect to the structure of the network by varying a single parameter, namely, a probability measure that captures the preference style of new edges in the preferential attachment model. Also, a high amount of concurrency in the generator's workload per thread or processor is observed when that probability is at very small fractions greater than zero. In future work, we intend to exploit code profiling tools for further optimization of the runtime and memory usage on the GPU. Also, the algorithm needs to be extended to exploit multiple GPUs that may be co-located within the same node. This would require periodic data synchronization across GPUs, which can be efficiently achieved using the NVidia Collective Communication Library (NCCL). Additional future work involves porting to GPUs spanning multiple nodes, and also hybrid CPU-GPU scenarios in order to utilize unused cores of multi-core CPUs. Methods to incorporate other network generator models can also be explored with our **cuPPA** as a starting point. Finally, future work is needed in converting our internal, GPU-based graph representation to other popular network formats for usability.

References

[1] P. Erdős and A. Rényi, "On the evolution of random graphs," *Publications of the Mathematical Institute of the Hungarian Academy of Sciences*, pp. 17–61, 1960.

[2] A.-L. Barabási and R. Albert, "Emergence of scaling in random networks," *Science*, vol. 286, no. 5439, 1999.

[3] R. Albert, H. Jeong, and A.-L. Barabási, "Error and attack tolerance of complex networks," *Nature*, 2000.

[4] D. P. Chassin and C. Posse, "Evaluating north american electric grid reliability using the barabási-albert network model," *Physica A*, vol. 355, no. 2-4, 2005.

[5] J. Leskovec and E. Horvitz, "Planetary-scale views on a large instant-messaging network," in *International Conference on World Wide Web*. ACM Press, 2008.

[6] R. Kumar, P. Raghavan, S. Rajagopalan, D. Sivakumar, A. Tomkins, and E. Upfal, "Stochastic models for the web graph," in *Annual Symposium on Foundations of Computer Science*. IEEE Comput. Soc, 2000.

[7] V. Batagelj and U. Brandes, "Efficient generation of large random networks," *Physical Review E*, 2005.

[8] M. Alam, M. Khan, and M. V. Marathe, "Distributed-memory parallel algorithms for generating massive scale-free networks using preferential attachment model," in *Intl. Conf. for High Performance Computing, Networking, Storage and Analysis*, 2013.

[9] J. Kleinberg, R. Kumar, P. Raghavan, S. Rajagopalan, and A. Tomkins, "The web as a graph: Measurements, models, and methods," in *Annual International Conference on Computing and Combinatorics*, 1999.

[10] M. Alam, "HPC-based parallel algorithms for generating random networks and some other network analysis problems," Ph.D. dissertation, Virginia Tech, 2016.

[11] K. Azadbakht, N. Bezirgiannis, F. S. de Boer, and S. Aliakbary, "A high-level and scalable approach for generating scale-free graphs using active objects," in *Proc. of the Annual ACM Symposium on Applied Computing*, 2016.

[12] S. Dorogovtsev and J. Mendes, "Evolution of networks," in *Advances in Physics*, vol. 51, no. 4, 2002.

[13] A. Yoo and K. Henderson, "Parallel generation of massive scale-free graphs," *CoRR*, 2010.

[14] P. Sanders and C. Schulz, "Scalable generation of scale-free graphs," *Info. Proc. Letters*, 2016.

[15] U. Meyer and M. Penschuck, "Generating massive scale-free networks under resource constraints," in *Proc. of the Workshop on Algorithm Engineering and Experiments (ALENEX)*, 2016.

[16] S. Nobari, X. Lu, P. Karras, and S. Bressan, "Fast random graph generation," in *International Conference on Extending Database Technology*, 2011, p. 331.

[17] D. J. Watts and S. H. Strogatz, "Collective dynamics of 'small-world' networks," *Nature*, no. 6684, 1998.

[18] A. Leist and K. Hawick, "Graph generation on GPUs using dynamic memory allocation," in *Intl. Conf. on Parallel and Distributed Processing Techniques and Applications*, 2011.

[19] B. Machta and J. Machta, "Parallel dynamics and computational complexity of network growth models," *Physical Review E*, vol. 71, no. 2, p. 26704, 2005.

[20] S. Dorogovtsev, J. Mendes, and A. Samukhin, "Principles of statistical mechanics of uncorrelated random networks," *Nuclear Physics B*, 2003.