# Accelerating Triangle Counting on GPU

Lin Hu
hulin@pku.edu.cn
Peking University
Beijing, China

Lei Zou
zoulei@pku.edu.cn
Peking University
Beijing, China

Yu Liu
dokiliu@pku.edu.cn
Peking University
Beijing, China

## ABSTRACT

Triangle counting is an important problem in graph mining, which has achieved great performance improvement on GPU in recent years. Instead of proposing a new GPU triangle counting algorithm, in this paper, we propose a novel lightweight graph preprocessing method to boost many state-of-the-art GPU triangle counting algorithms without changing their implementations and data structures. Specifically, we find common computing patterns in existing algorithms, and abstract two analytic models to measure how workload imbalance and diversity in these computing patterns affect performance exactly. Then, due to the NP-hardness of the model optimization, we propose approximate solutions by determining edge directions to balance workloads and reordering vertices to maximize the degree of parallelism within GPU blocks. Finally, extensive experiments confirm the significant performance improvement and high usability of our approach.

## KEYWORDS

Triangle counting, GPU, Edge directing, Vertex ordering

## 1 INTRODUCTION

Graph is a broadly used representation of data, which is getting increasingly important. Triangle counting [8, 15, 28, 31], whose task is computing the total number of triangles in a graph, lays the foundation for various graph problems, such as k-truss [34], clustering coefficient [37], and link recommendation [33].

With the scale of graphs getting larger continuously, lots of existing works resort to new hardwares with great parallel processing ability (such as GPU) to efficiently address graph computation tasks, such as BFS [24], SSSP [11], and subgraph isomorphism [32]. Triangle counting on GPU has also been extensively studied in the literature[13, 18, 20, 28]. Although GPU is massively parallel, the GPU architecture also brings about several technical challenges in designing graph algorithms. Firstly, warp is a basic running unit of GPU, and threads inside run in SIMT (single instruction multiple threads) manner. Therefore, workload imbalance, i.e., thread divergence, will cause massive stalls of threads, thus leading to severe performance decline. And the workload imbalance of graph algorithms on GPU often stems from skewed vertex degree distribution in many real-world graphs. Secondly, block is a programmer-level unit and tasks in a block will be dispatched to only one SM (Stream Multiprocesser), which has its own computational and memory resources. To fully utilize both resources, we should group tasks with different workload features (i.e., memory intensive or computing intensive) together and dispatch them to one SM. Otherwise, assigning tasks with the same resource preferences to one SM will lead to waste of the other resource.

In this paper, we focus on triangle counting problem. Considering the two optimization goals—workload balance and resource balance, most existing works focus on proposing a new GPU triangle counting algorithm to balance the workload and resource utilization as much as possible. However, we aim to propose a unified strategy to optimize multiple existing triangle counting algorithms by preprocessing graph data. Our work is inspired by an interesting observation: There are two common data preprocessing methods in existing triangle counting algorithms: edge directing and vertex ordering. For the same data graph and GPU triangle counting algorithm, different edge direction methods or vertex ordering methods result in different performances (see Table 2). Actually, these two preprocessing steps significantly impact the workload balance and resource balance. Therefore, we concentrate on data preprocessing strategies that benefit a group of relevant GPU triangle counting algorithms. To be more specific, we first abstract two common computing patterns (intra-block synchronization and binary search based list intersection) by analyzing existing GPU triangle counting algorithms. Then we propose two quantitative analytic models over those common computing patterns to reveal the effect of preprocessing steps (edge directing and vertex ordering). Next, we define the optimal preprecessing strategies from those models. We also study the hardness of computing optimal data preprocessing solutions. Due to the NP-hardness (proven in Theorems 4.1 and 5.1), we design lightweight approximate algorithms in this work.

We first analyze the relations between the preprocessing methods and the optimization goals (workload balance and resource balance) as follows:

*Edge Directing.* In triangle counting, an undirected graph[1] is firstly transformed into a directed one to avoid redundant computation [18, 28]. Obviously, edge direction changes the out-degree distribution. And the workload imbalance stems from the skewed degree distribution. Therefore, proper edge direction will potentially balance workloads. Two popular strategies are ID-based and degree-based. The former defines a directed edge from small vertex ID to large vertex ID, and the latter directs an edge from the small degree vertex to the large degree vertex. Experimentally, the later often leads to better performance, but it is a heuristic strategy. To the best of our knowledge, no existing work tries to figure out if there are better edge directing strategies or discuss this problem by analytic models. We will study this problem in Section 4.

*Vertex Ordering.* In triangle counting, vertices with different adjacency list lengths have different resource preferences. Given the order of vertices, consecutive vertices are grouped and assigned to the same block. Therefore, vertex ordering strategy determines the task assignment, which further influences the balance of resource usage. In other words, vertex ordering provides an opportunity

[1]Triangle counting definition [8, 13, 15, 18, 20] ignores the edge direction, even the underlying graph is directed.

to balance resource usage. Intuitively, grouping computing intensive and memory intensive tasks together will avoid the waste of resources. We will study this problem in Section 5.

To make our preprocessing methods have more generality, we propose analytic models based on two widely used computing patterns: intra-block Synchronization and binary search based list intersection. The models summarize various implementations and intuitively demonstrate how those two preprocessing methods impact the optimization goals. Naturally, we define quantitative optimal edge direction and vertex ordering solutions according to our models, which distinguish our methods from any other heuristic strategies, and then propose solutions to them. In conclusion, our proposed models bridge specific implementations and quantitative optimal solutions.

To summarize, we aim to find the optimal edge directing to balance workloads and the optimal vertex ordering to maximize resource utilization. Unfortunately, both problems are NP-hard (Theorems 4.1 and 5.1). In practice, the data preprocessing time should also be considered; data preprocessing provides an opportunity to improve the GPU kernel performance, but the overall performance would decline if preprocessing is time-consuming. Therefore, we propose lightweight approximate algorithms to find good solutions for edge directing and vertex ordering. Extensive experiments on both real-world and synthetic datasets confirm that our Analytic methods (called A-direction and A-order) significantly accelerate GPU kernel running time, and total time (including our data preprocessing step) is accelerated by up to 82%. Besides, the generality of our method comes from the generality of two computing patterns (which will be analyzed in detail later) in GPU triangle counting algorithms: once an algorithm, even not limited to triangle counting, has either computing pattern, our method is suitable for optimizing it without changing its implementation.

Generally, we have made the following contributions:

- We present two analytic models abstracted from common computing patterns in several state-of-the-art triangle counting implementations, considering GPU architecture features. These two models intuitively demonstrate how edge direction and vertex ordering impact the optimization goals.
- To alleviate workload imbalance, we aim to find an optimal edge directing scheme based on our analytic model. Due to the NP-hardness of this problem, we propose a lightweight approximate algorithm (A-direction) with performance guarantee.
- Considering workload diversity, we implement a better task assignment approach (A-order) by reordering vertices to balance resource requests and maximize the degree of parallelism. Specifically, we formalize vertex reordering as a model optimization problem.
- We conduct extensive experiments to verify the effectiveness of our analytic models and our data preprocessing methods. The results confirm that our approach can speed up state-of-the-art triangle counting algorithms significantly.

## 2 PRELIMINARIES

### 2.1 GPU Architecture

We will briefly introduce GPU architecture from both hardware and software perspectives.

**Software.** CUDA (Compute Unified Device Architecture) is the most popular GPU programming language. It uses *block* as the programmable unit for programmers, which has many warps. *Warp* is a basic unit of thread execution and memory access. A warp contains 32 threads, and they follow the lock-step rule strictly. Thus branches inside of it will lead to some threads in idle. The workload of different threads should be balanced because imbalance workload among threads will lead to severe thread divergence.

**Hardware.** Global memory on GPU has the slowest access rate, but can be accessed by all threads. GPU has many stream multiprocessors (SM), and each SM is an independent hardware unit, which not only contains many cores, but also has local fast-access memory, called *shared memory*, which is programmable but with limited space. SM can be seen as a match of *block*, because a block will be assigned to only one SM in run-time. A reasonable block task assignment is to group many tasks with different resource preferences together, then thread schedule mechanism will make full use of two kinds of resources in an SM for parallel processing. Otherwise, if tasks with the same resource preference are assigned to a block, many threads have to be suspended due to memory or computing conflict, and the other resource will be idle and wasted. That feature supports our reordering strategy from hardware level.

Global memory and shared memory access are both launched by a warp. If all required data of the threads inside a warp can be fetched within one memory transaction, we can achieve better efficiency, and we call this pattern as *coalesced memory access* in the following context.

### 2.2 Related Work

In this paper, we review related work in two categories: existing triangle counting algorithms and analytic model for GPU computing.

*2.2.1 Triangle Counting Algorithms.* Here we will review GPU and CPU triangle counting algorithms respectively.

**GPU implementations.** Workload balance is the key of GPU algorithms, and we will classify algorithms according to their workload distribution manner. The basic parallelized method [28] uses a thread to deal with an edge. Wang et al. implement the algorithm [35] using Gunrock [36] library with the same granularity. To better balance the workload of different threads, Green et al. present a method [13, 15] that estimates workload of each edge in advance and then assigns threads to edges accordingly. Bisson et al. [8] come up with an implementation which uses a block to deal with a vertex. This method uses bitmaps in shared memory or global memory for fast lookup of a list. And in TriCore [20], each warp is dispatched to deal with an edge, which makes full use of SIMT features. Hu et al. [18] present a finer-grained workload distribution method, in which a thread checks if a wedge "$u$-$v$-$w$" forms a triangle.

**CPU implementations.** Algorithms of triangle counting on CPU can be mainly divided into three categories [29]: node-iterator [2], edge-iterator [7] and forward algorithm [29]. Node-iterator algorithm iterates over all nodes and checks each pair of neighbors if they are connected by an edge; edge-iterator iterates over all edges and intersects the adjacency lists of both incident nodes, and forward algorithm is a refinement of edge-iterator. Besides, there are some methods based on map-reduce [22] and matrix multiplication [6, 38, 40]. Some algorithms are carried out on multi-core platform [31] or distributed system [5, 14]. In comparison, the idea of reducing overall workload such as forward algorithm [29] is similar among CPU and GPU implementations. However, in other aspects, methods on GPU are much different from those on CPU because of SIMT execution, jointly memory access of a warp and different threads communication mechanism. We have to pay more attention to workload balance and coalesced memory access. That is different from the CPU implementations, including serial methods, multi-core methods and SIMD (Single Instruction Multiple Data) parallelism [16].

**List intersection in triangle counting** Note that list intersection accounts for major time cost in triangle counting [16]. Generally, there are mainly two methods for list intersection: binary search [13, 15, 18–20, 36] and sort-merge [13, 15, 28, 36]. Also, there are other methods such as pivot-skip merge [12], bitmap-merge [8] and liner regression approach [4]. As a general preprocessing method to optimize the performance, we focus on the most common list intersection methods in GPU triangle counting algorithms. Besides, binary search is proven better than merge-based list intersection because of better independence, less work complexity and larger degree of parallelism [4, 18–20] in GPU triangle counting.

#### 2.2.2 *Analytic Model for GPU Computing.*

Existing analytic models for GPU computing mainly focus on performance predictions. For example, Hong et al. [17] propose a prediction model for algorithms on GPU, which introduces two metrics for judging resource preferences of parallel programs, and gives estimations of the number of memory requests. Amaris et al. [3] also propose an execution time prediction model by considering computation, memory access and cache usage. Ma et al. [25] design a performance analytic model for memory-limited kernels, focusing on tuning various configuration parameters. Yang et al. [39] work on matrix multiplication. They build lookup tables between tile shape and performance, and estimate the optimal workload size in each tile. In this paper, we propose our analytic models to quantify the performance effects concerning different preprocessing methods. To the best of our knowledge, no existing work studies this problem.

## 2.3 Synchronization in Triangle Counting

Synchronization among threads is common on GPU. Threads often share workspace in memory, thus synchronization is necessary for consistent access to memory. Bisson's work [8] and Hu's implementation [18] are two representative works that adopt synchronization among threads. The former is for consistent access to global memory, while the latter is for shared memory. Here we briefly introduce them to abstract run-time model later.

In Bisson's work [8], a block is responsible for the whole triangle counting tasks of a vertex. In Figure 1, a block with only two threads is working on vertex 9. Each thread is in charge of one vertex (vertex 4 or 5) in the adjacency list of 9 (denoted as $l$), and it intersects the adjacency list of that vertex with $l$. They use bitmaps for fast lookup of elements in $l$. Specifically, two threads firstly set their current

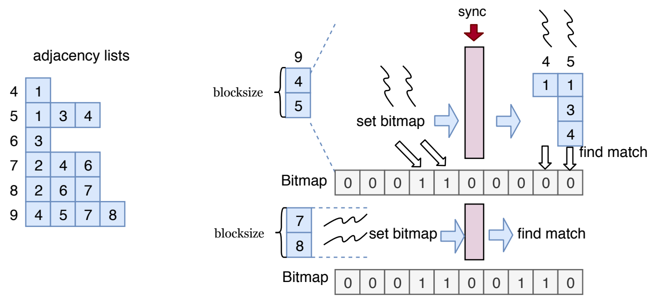

**Figure 1: Running process of Bisson's work**

vertices to 1 in the bitmap of $l$. Then follows a *synchronization* in the block to make sure both threads finish setting the bitmap. After that, each thread uses the bitmap to find matches for every element in the adjacency list of its vertex. Threads inside the block will move to next group of vertex 9's neighbors and continue the procedure until all vertices in $l$ are processed.

In Figure 1, neighbor lists of vertex 4 and 5 have different lengths, while each neighbor list gets one thread to find matches of all elements between two synchronization steps. Size of adjacency lists may vary a lot, leading to workload imbalance among threads.

In Hu's work [18], each thread checks if a wedge $u$-$v$-$w$ forms a triangle by doing binary searches for $w$ in $u$'s adjacency list. We use three rows in Figure 2 to represent the sets of $u$, $v$, and $w$, respectively. Assuming there are four threads in total, we show how this method works in Figure 2. Firstly, a piece of the second row ($u$'s adjacency lists) will be loaded into shared memory by a block for faster access. Then the synchronization step is necessary after loading. Next, binary searches will be carried out for the loaded piece of neighbor lists by threads inside the block. The above process forms a "copy-synchronize-search" pattern.

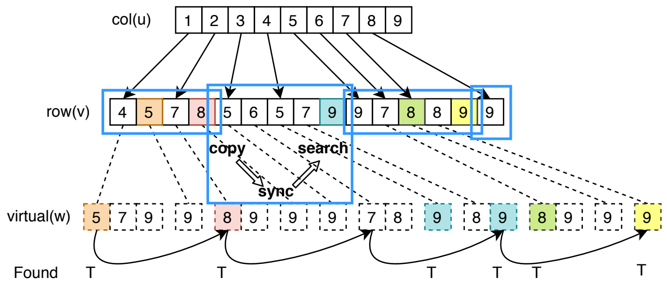

**Figure 2: Running process of Hu's fine-grained method**

In this work, threads are doing binary searches in lists with different lengths between two synchronization steps. Variations in the length of lists, which are significant in power-law graphs, cause workload imbalance among threads in above two methods.

Table 1 lists frequently used notations in the remainder of the paper.

## 3 ANALYTIC MODELS

In this section, two analytic models are abstracted based on two common computing patterns, *intra-block synchronization* and *binary search*, then we bring up two cost functions to quantitatively describe the influence of edge directing and vertex ordering.

**Table 1: Frequently used notations**

| variable | description |
|---|---|
| *Graph attributes* | |
| $u, v, w$ | three vertices for a triangle |
| $\delta_{uv}$ | variable denoting edge direction between u,v |
| $d(u)$ | vertex u's degree in undirected graphs |
| $d_o(u)$ | vertex u's outgoing degree in directed graphs |
| $\bar{d}_o$ | average outgoing degree in directed graphs |
| $(u, v)$ | undirected edge between u and v |
| $u \rightarrow v$ | directed edge from u to v |
| *variables in edge direction* | |
| $v_c$ | vertex with degree larger than $\bar{d}_o$ |
| $v_n$ | vertex with degree smaller than $\bar{d}_o$ |
| $V_n$ | vertices set of $v_n$ |
| $V_c$ | vertices set of $v_c$ |
| $\rho$ | approximate ratio of our algorithm for edge direction |
| *variables in graph reordering* | |
| $c$ | computing intensity of a vertex |
| $m$ | memory access intensity of a vertex |
| $F_c$, $F_m$ | functions transforming $d_o(v)$ to $c$ and $m$, respectively |
| $\lambda$ | variable transform $c$ to equal $m$ |
| $BW$ | shared memory bandwidth |
| $p_c$ | computing intensity pressure |
| $mem\_sup$ | sum of all vertices' memory superiority in a bucket |

## 3.1 Intra-block BSP Model

*3.1.1 Model description.* From Section 2.3 we know that synchronization plays a key role in many implementations, which assures consistent parallel access to global memory and shared memory among threads. This pattern can be modeled as BSP (Bulk Synchronous Parallel) within a block, in which memory setting, synchronization and all threads executions form a superstep. We call such process as *intra-block BSP Model.* It is shown in Figure 3, in which we assume that there are only four threads for a block. Supersteps are performed repeatedly to finish the whole task.

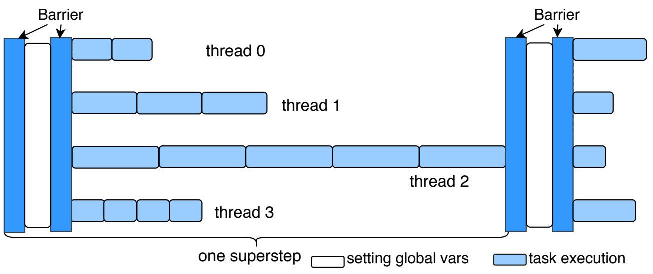

**Figure 3: Intra-block BSP model**

It is obvious that the running time of each superstep depends on the thread with the largest workload, e.g., thread 2 in Figure 3. In triangle counting, the workload is measured by each vertex's adjacency list length (i.e., vertex out-degree). For example, in Bisson's work [8], threads of vertices with long adjacency lists have to check the bitmap more times; while in Hu's work [18], threads searching in long adjacency lists have to suffer longer memory latency and more binary search times. Moreover, real-world graphs often employ the skewed vertex degree distribution [26], which leads to severe workload imbalance. On the other hand, edge direction is necessary for triangle counting to avoid redundancy computation. This preprocessing provides an opportunity to change the vertex out-degree distribution and our model's performance.

*3.1.2 Experimental rationality.* We use Hu's work [18] as an example, and show the running time under different edge directing methods in Table 2. In the last three columns of Table 2, given the same triangle-counting algorithm [18], we show the kernel running time of different edge direction strategies. Because of more balanced length of adjacency lists, the popular degree-based edge direction

| R-strategy | D-order | A-order | Original order | | |
|---|---|---|---|---|---|
| D-strategy | D-direction | | | ID-based | A-direction |
| com-lj | 186 | 144 | 201 | 265 | 142 |
| cit-Patent | 151 | 95 | 130 | 640 | 102 |
| soc-Live | 326 | 176 | 246 | 312 | 181 |
| kron-log21 | 9611 | 5020 | 8040 | 10982 | 5220 |

**Table 2: Running time (msec) on four datasets under different vertex Reorder strategies and edge Direction strategies**

(simplified as D-direction), which directs edges from vertices of small degree to those of large degree, significantly outperforms ID-based strategy, which directs edges from vertices of small ID to those of large ID. And our A-direction strategy (the last column in Table 2) achieves further improvements to D-direction.

*3.1.3 Analytic model.* From the intra-block BSP model, we can conclude that to balance the workload of threads, we need to balance the size of adjacent lists (i.e., vertex out-degree) as far as possible. Given an undirected graph $G$, it is transformed into the directed one $\mathcal{G}$ according to some edge directing scheme $\mathcal{P}$. We define the cost function Equation (1) to measure the workload balance, in which $d_o(u)$ stands for the out-degree of vertex $u$ in the directed graph, and $\bar{d}_o$ means the average of out-degree in the directed graph.

$$C(\mathcal{P}) = \sum_{u \in V(\mathcal{G})} |d_o(u) - \bar{d}_o| \qquad (1)$$

Our goal is to find the best edge directing scheme $\mathcal{P}$ that minimizes the cost function $C(\mathcal{P})$, which indicates a more even degree distribution. It then causes more balanced workload and better running performance. Unfortunately, this optimization problem is proven NP-hard (see Theorem 4.1). Therefore, a lightweight approximate algorithm with performance guarantee (approximate ratio under 1.8) is proposed. All these will be studied in Section 4.

## 3.2 Resource Balance Model

*3.2.1 Binary search based list intersection.* As mentioned earlier, previous works [4, 18, 20] have pointed out that binary search has better performance in GPU triangle counting. Thus, as a general preprocessing method, we focus on the binary search, which is a more popular manner of list intersection in state-of-the-art triangle GPU triangle counting algorithms, including Gunrock [36], TriCore [20], Hu's implementation [18] and Fox's work [13, 15].

Here we introduce two concepts: computing intensity $F_c$ and memory access intensity $F_m$. The former is defined by the number of computation steps per second (i.e., compute throughput), and the latter refers to the volume of accessed data per second (i.e., memory bandwidth). Both concepts are defined in a fixed time window. We find that binary search shows different resource preferences distinguished by the length of target list, as shown in Figure 4.

In Figure 4 (a), we assume that a warp with four threads is accessing the same list $L$ for different search keys. If $L$ is short, all data of the four threads can be loaded using a single memory transaction. However, if $L$ is too long for one memory transaction, the four threads tend to access scattered positions of the list, resulting in multiple memory transactions. Memory access and computation are carried out by turns, thus different memory access latency in Figure 4 (a) leads to different time proportions of memory access and computation in a fixed time window in Figure 4 (b): given a fixed time window, tasks on short lists have more computation steps (larger $F_c$), while those on long lists have larger volume of accessed data (larger $F_m$).According to the proportions of different resource usage in a time window, we conclude that short lists are computing intensive while long lists are memory intensive for binary search. Binary search naturally has good memory access patterns for multiple threads in first few searches, and the space locality of short list helps to keep coalesced access patterns in later searches.We also report experiment results of memory bandwidth and compute throughput varying with adjacency list lengths in Figure 7 (in Section 5.3), which support our above analysis.

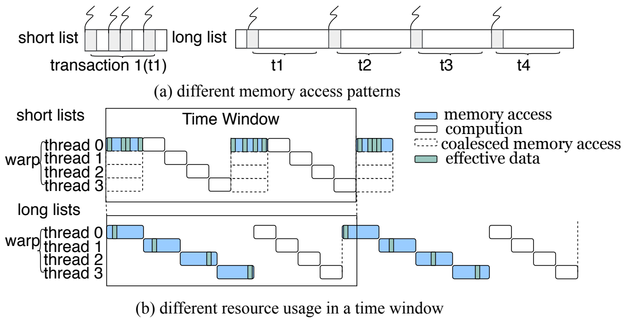

**Figure 4: Memory access patterns on lists with different length and concerned time windows**

### 3.2.2 *Model description.*

Existing GPU triangle counting algorithms always try to balance overall workloads, but they often ignore the diversity of workloads. In fact, workloads should also be balanced between computational cost and memory access cost.

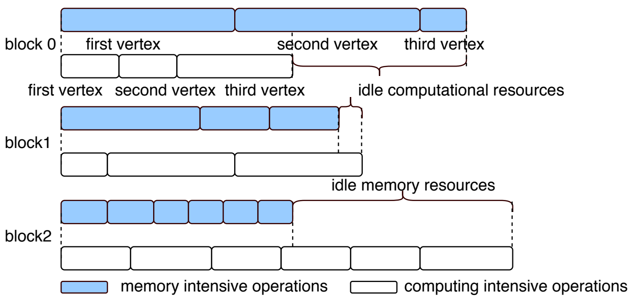

**Figure 5: Resource balance model**

Modern GPUs have mature mechanisms to schedule warps on and off cores, aiming at making the best use of computational and memory resources. Computing intensive warp can “steal” computational resource of the SM when memory intensive warp is scheduled off due to unfinished memory transactions. Therefore, tasks with different resource preference in a time window could make up for each other by hardware schedule, and a reasonable task assignment should group tasks with different workload features (i.e., memory intensive or computing intensive) together and dispatch them to one block. Consider the example in Figure 5, more computational resources are wasted in *block* 0 since the three vertices are all memory intensive tasks. *block* 2 in Figure 5 has an analogue problem. In contrast, *block* 1 has balanced resource requests, since it combines memory intensive tasks with computing intensive ones.

### 3.2.3 *Experimental rationality.*

The following problem is how to bring up a good block task assignment scheme with the above observation. Given the order of vertices, blocks usually fetch consecutive vertices as their work sets. So a possible solution is to reorder the vertices in a given graph to change the block task assignment.

Let us see the performance under different vertex reorder strategies in the first three columns for Table 2. Degree-based order (D-order) means reordering vertices in the degree descending (or ascending) order. It has the worst performance, even significantly slower than the original vertex ordering. In D-order, the vertices with similar degrees are grouped together, which aggravate the resource usage imbalance, since they have the same resource preferences. On the contrary, our balanced A-order scheme leads to the best performance.

### 3.2.4 *Analytic model.*

As discussed above, the memory access cost and computational cost are both related to the vertex out-degree $d_o(v)$, thus, we use the functions $F_m(d_o(v))$ and $F_c(d_o(v))$ to denote memory intensity $m$ and computing intensity $c$, respectively. Given a vertex ordering $\mathcal{R}$, every consecutive $k$ vertices are grouped into a bucket $B_i$. Suppose there are $b$ buckets in total. All triangle counting tasks with regard to a bucket are assigned to one block. To find the optimal block task assignment to improve the resource usage, we model it as the vertex ordering problem as follows:

For each bucket $B_i$, the corresponding computational cost $C_i$ and memory access cost $M_i$ are defined as:

$$C_i = \sum_{v \in B_i} F_c(d_o(v)),\ M_i = \sum_{v \in B_i} F_m(d_o(v)) \tag{2}$$

The optimization goal is to find a reordering scheme $\mathcal{R}$ such that:

$$\begin{aligned} &min_{\mathcal{R}} \sum_{i=1}^{b} |\lambda C_i - M_i| \\ &s.t.\ \ C_i \le C_{max}, \forall i \in \{1, b\} \\ &\qquad M_i \le M_{max}, \forall i \in \{1, b\} \end{aligned} \tag{3}$$

For any bucket $B_i$, $|\lambda C_i - M_i|$ denotes the wasted resource size, i.e., the idle resource in Figure 5. The $\lambda$ is a parameter which transforms computational cost to equal memory access cost. The setting of computing intensity function $F_c$, memory intensity function $F_m$, and the ratio $\lambda$ depends on the underlying GPU hardware.

Our analytic model's goal is to find the optimal reordering scheme $\mathcal{R}$ to minimize Equation (3). We will discuss the parameter setting, the hardness of this optimization problem and the corresponding algorithm in Section 5.

## 3.3 Correlation of Two Models

Our two models have their respective tendency: intra-block BSP model aims to achieve better workload balance *within* a block, while vertex ordering aims to realize better resource utilization *among* blocks. The hierarchical targets of the two strategies assure the compatibility of them. We mainly use the first model to guide edge direction, and the second one for vertex reordering. Since Formula (3) only considers dispatching tasks among blocks (vertices in one bucket are assigned to one block), we also reorder vertices in the same bucket according to their out-degrees from the consideration of workload balance within a block.

## 4 EDGE DIRECTION

In this section, we first study the hardness of the edge directing problem and then propose our A-direction method, which achieves a satisfying approximation ratio.

### 4.1 Hardness Analysis

THEOREM 4.1. *Given an undirected graph $G = (V, E)$, finding the optimal edge directing scheme $\mathcal{P}$ to transfer $G$ into the corresponding directed graph $\mathcal{G}$ to minimize Equation (1) (in Section 3.1.3) is an NP-hard problem.*

PROOF. Generally, we can reduce a 0-1 integer planning problem to this optimal edge direction problem.

Given a set $V$, paired variable $\delta_{uv}$ and $\delta_{vu}$ denote the connection between $u$ and $v$, for any $u, v \in V$. $C$ is a constant. Considering the following 0-1 integer liner programming problem:

$$\begin{aligned} &min \sum_{u\in V} | \sum_{v\in V} \delta_{uv} - C| \\ &s.t.\ \ \delta_{uv} \in \{0,1\}, \forall u, v \in V, \\ &\qquad \delta_{uv} + \delta_{vw} + \delta_{wu} \leq 2, \forall u, v, w \in V, \\ &\qquad \delta_{uv} + \delta_{vu} = 1, \forall u, v \in V. \end{aligned} \tag{4}$$

We reduce it to an edge direction problem. $V$ is used as vertex set of an undirected graph G, whose edge set $E$ consists of $(u, v)$ for all $\delta_{uv} \in$ Equation (4). We use $\delta_{uv} = 1$ (or $\delta_{vu} = 1$) to denote that the undirected edge (u,v) is turned into directed edge $u \longrightarrow v$ (or $v \longrightarrow u$). Obviously, an undirected edge can be turned into only one directed edge, so we have that $\forall(u, v) \in E$, $\delta_{uv} + \delta_{vu} = 1$. Furthermore, to guarantee the correctness (i.e. each triangle is counted exactly once), we also require that the transferred directed graph ($\mathcal{G}$) does not contain any 3-length directed loop [2], which says $\delta_{uv} + \delta_{vw} + \delta_{wu} \leq 2, \forall u, v, w \in V$.

If we set $C$ in Equation 4 as $\bar{d_o}$, Equation (4) is equivalent to Equation (1), which is an optimal edge direction problem.

The equivalence between the 0-1 integer planning and finding the optimal edge directing scheme $\mathcal{P}$ is straightforward. The former is a classical NP-hard problem. Thus, the theorem holds. □

### 4.2 Approximate Algorithm

Due to the NP-hardness, we propose A-direction strategy, a light-weight approximation algorithm to achieve a good trade-off between efficiency and effectiveness.

DEFINITION 4.1. *Given an undirected graph $G = (V, E)$, for each vertex $u \in G$, if $d(u) \geq \bar{d_o} = \frac{|E|}{|V|}$, we refer to $u$ as a* core *vertex; otherwise, $u$ is a* non-core *vertex.*

Generally, we use $v_c$ (resp. $v_n$) to represent a core (resp. non-core) vertex. In particular, all core vertices and non-core vertices are collected as a set $V_c$ and $V_n$ respectively. The following lemma states that for a large fraction of the edges, their direction can be determined without affecting the problem optimality under the intra-block BSP model.

[2]Otherwise, the 3-length directed loop will not be counted in almost all triangle counting implementations

LEMMA 4.1. *Given an undirected graph $G = (V, E)$, after the optimal edge directing scheme $\mathcal{P}_{Opt}$, $G$ is transformed into directed graph $\mathcal{G}$, the following claims hold:*

- *For any undirected edge $e = (v_c, v_n) \in E \wedge v_c \in V_c \wedge v_n \in V_n$ (i.e, an edge links a core vertex and a non-core vertex), the edge direction must be $v_n \longrightarrow v_c$ in $\mathcal{G}$.*
- *For any undirected edge $e = (v_n, v_n') \in E \wedge v_n, v_n' \in V_n$ (i.e, an edge links two non-core vertices), the edge direction between $v_n$ and $v_n'$ can be arbitrarily defined in $\mathcal{G}$, which does not affect the final cost $C(\mathcal{P}_{Opt})$.*

PROOF. Given a non-core vertex $v_n$, the corresponding term in Equation (1) is

$$|d_o(v_n) - \bar{d_o}| \tag{5}$$

because $d_o(v_n) \leq d(v_n) < \bar{d_o}$, Equation (5) can be simplified as:

$$\bar{d_o} - d_o(v_n) \tag{6}$$

That's the key of our edge directing strategy in the lemma. And there are two kinds of edges concerning $v_n$:

- edge ($v_n$,$v_n'$). Different direction of the edge makes no difference in that
$$\begin{aligned} &(\bar{d_o} - (d_o(v_n) + 1)) + (\bar{d_o} - d_o(v_n')) \\ = \ &(\bar{d_o} - d_o(v_n)) + (\bar{d_o} - (d_o(v_n') + 1)) \end{aligned} \tag{7}$$
- edge ($v_n$,$v_c$). Assigning the edge as out-edge of $v_n$ doesn't go worse than the opposite choice because:
$$\begin{aligned} &(\bar{d_o} - (d_o(v_n) + 1)) + |\bar{d_o} - d_o(v_c)| \\ = \ &(\bar{d_o} - d_o(v_n)) + |\bar{d_o} - d_o(v_c)| - 1 \\ \leq \ &(\bar{d_o} - d_o(v_n)) + |\bar{d_o} - (d_o(v_c) + 1)| \end{aligned} \tag{8}$$

So that the directed edges generated by our lemma don't go any worse than any other options. □

**Algorithm 1 Parallel A-direction Algorithm**

**Input:** undirected graph G, degree $d$ of all vertices
**Output:** directed graph $\mathcal{G}$

```
1: $threshold \leftarrow \bar{d_o}$
2: $Frontier \leftarrow \varnothing$, $reachedSet \leftarrow \varnothing$
3: $runtimeDegree \leftarrow \varnothing, priority \leftarrow \varnothing$
4: while !$all\ nodes\ peeled$ do
5:     $Frontier \leftarrow parallel\_collect\_vertex(threshold)$
6:     if $Frontier.size == 0$ then
7:         $threshold \leftarrow threshold + \bar{d_o}$
8:     $runtimeDegree, priority \leftarrow parallel\_set(Frontier)$
9:     $reachedSet \leftarrow parallel\_advance(Frontier)$
10:    $parallel\_back\_traversal\_and\_update\_deg(reachedSet)$
11: $parallel\_edge\_direction(runtimeDegree, priority)$
```

Based on Lemma 4.1, we design a *parallel peeling* algorithm on GPU (Algorithm 1). Intuitively, our method repeatedly peels off vertices with degree less than an increasing threshold. The *runtimeDegree* of a vertex denotes its remained neighbor size in the running process, while *priority* of a vertex denotes when the vertex is peeled. Firstly, vertices with degrees less than *threshold* are collected and their information are recorded (line 5 and line 8). If the *Frontier* is empty, we increase *threshold* to further peel vertices (line 6-7). Then we want to update the *runtimeDegree* of vertices adjacent to vertices in *Frontier*. However, updating from vertices in *Frontier* results in massive atomic operations of GPU, which will lead to severe performance decline. So we collect vertices which are neighbors of vertices in *Frontier* (line 9), and perform update of *runtimeDegree* from them (line 10). This process is performed repeatedly until the whole graph is processed. Finally, according to *priority* and *runtimeDegree*, edges are directed preferentially from earlier peeled vertices to later peeled vertices, then from vertices with smaller *runtimeDegree* to those with larger *runtimeDegree*.

According to Lemma 4.1, it is easy to know we get optimal edge direction when *threshold* is $\bar{d_o}$. Then we increase the *threshold*, and repeat the peeling procedure. Intuitively, this method effectively reduces the number of vertices with large out-degrees. And in experiments, we find that vertices of large out-degree in the processed graph are mainly from the last few frontiers in our algorithm. We slow the growth of *threshold* in the last few frontiers, hopefully to reach a more balanced degree distribution of the remained vertices, and that strategy works well in experiments.

Obviously, the algorithm with *threshold* greater than $\bar{d_o}$ is an approximate part; but we prove that the approximate ratio is small (less than 1.8) as follows.

## 4.3 Approximation Ratio

THEOREM 4.2. *The edge directing scheme of Algorithm 1 (resp. the one that optimizes Equation (1)) is denoted by $\mathcal{P}_{Alg}$ (resp. $\mathcal{P}_{Opt}$). Let $C(\mathcal{P})$ denote the cost of scheme $\mathcal{P}$, and $UB()$ (resp. $LB()$) its upper (resp. lower) bound. The approximation ratio $\rho$ of Algorithm 1 w.r.t the cost function is bounded by*

$$\rho = \frac{C(\mathcal{P}_{Alg})}{C(\mathcal{P}_{Opt})} \le 1 + \frac{UB(C(\mathcal{P}_{Alg}) - C(\mathcal{P}_{Opt}))}{LB(C(\mathcal{P}_{Opt}))}, \tag{9}$$

*where*

$$LB(C(\mathcal{P}_{Opt})) = \begin{cases} |E| - \mathcal{D}(V_N) - \dfrac{\mathcal{D}(V_N)}{2}, \mathit{if}\ \dfrac{\mathcal{D}(V_C)}{2} \le \bar{d_o}|V_C|, \\ \dfrac{1}{2}(\mathcal{D}(V_C) - 3\mathcal{D}(V_N)) + \bar{d_o}(|V_N| - |V_C|), \\ \qquad \mathit{if}\ \dfrac{1}{2}(\mathcal{D}(V_C) - \mathcal{D}(V_N)) \ge \bar{d_o}|V_C|, \\ \bar{d_o}|V_N| - \mathcal{D}(V_N), \mathit{otherwise.} \end{cases} \tag{10}$$

$$\begin{aligned} UB(C(\mathcal{P}_{Alg}) - C(\mathcal{P}_{Opt})) &= d_o \sum_{d=\bar{d_o}+1}^{d_p} |V_d|, \\ d_p &= \operatorname{argmin}_{d^*} \sum_{d=\bar{d_o}+1}^{d*} \mathcal{D}(V_d) \ge \frac{\mathcal{D}(V_C)}{2}. \end{aligned} \tag{11}$$

*$V_N = \{v|d(v) \le \bar{d_o}\}$ denotes the set of* non-core *nodes, $V_C = V \backslash V_n$ denotes the set of* core *nodes, $V_d = \{v|d(v) = d\}$, and $\mathcal{D}(S) = \sum_{v\in S} d(v)$.*

PROOF. By the definition of approximation ratio, we have $\rho = \frac{C(\mathcal{P}_{Alg})}{C(\mathcal{P}_{Opt})} = 1 + \frac{C(\mathcal{P}_{Alg}) - C(\mathcal{P}_{Opt})}{C(\mathcal{P}_{Opt})} \le 1 + \frac{UB(C(\mathcal{P}_{Alg}) - C(\mathcal{P}_{Opt}))}{LB(C(\mathcal{P}_{Opt}))}$. Given a graph with nodes $v_1, \ldots, v_n$ and the corresponding degrees $d_{v_1}, \ldots, d_{v_n}$, we consider the maximum possible $\rho$ for any graph with the same multi-set of degrees.

We first consider the lower bound of $C(\mathcal{P}_{Opt})$. From Equation (9), we have $C(\mathcal{P}_{Opt}) = \sum_{v\in V_N} |d(v) - \bar{d_o}| + \sum_{v \in V_C} |d_o(v) - \bar{d_o}|$. The first term is at least $\sum_{v\in V_N} (\bar{d_o} - d(v))$. For the second one, we have $\sum_{v\in V_C} |d_o(v) - \bar{d_o}| \ge |\frac{\sum_{v\in V_C} d(v) - |E_{NC}|}{2} - \bar{d_o}|V_C||$. By considering three cases: (1) $\frac{\sum_{v\in V_C} d(v)}{2} < \bar{d_o}|V_C|$; (2) $\frac{\sum_{v\in V_C} d(v) - |E_{NC}|}{2} - \bar{d_o}|V_C| \ge 0$; and (3) otherwise, Equation (10) follows.

For the upper bound of $C(\mathcal{P}_{Alg})$, first note its non-optimality only results from the edges between core nodes, while each edge increases the upper bound by at most 2. To maximize the upper bound, we adopt the following greedy strategy by directing all incident edges of nodes in $V_{\bar{d_o}+1}$ as in-edges, and then nodes in $V_{\bar{d_o}+2}$ and so on. This derives the worst-case upper bound as in Equation (11). □

In fact, Theorem 4.2 provides formulas of approximation ratio $\rho$ about degree distributions. And it does not depend on any specific generated models. To show that $\rho$ is bounded in practice, we use a popular graph generation model with different densities and real-world graphs to obtain the degree statistics, and further show how $\rho$ changes with them. Since the degree distributions of most real-world graphs follow power law, we use a power-law graph configuration model—ACL model [1] to generate a sequence of graphs varying the edge density. Then, we compute $\rho$ using Theorem 4.2 for these graphs and plot the relation between $\rho$ and $\bar{d_o}$ in Figure 6. It turns out that $\rho$ is less than 1.8 for graphs of arbitrary density.

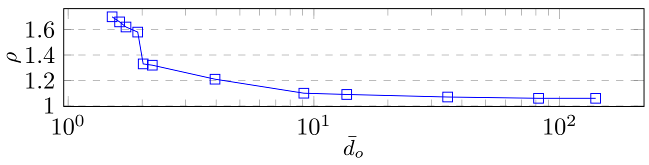

**Figure 6: Approximate ratio under power law graphs**

Table 3 shows the $\rho$ for several real-world graphs, which are all smaller than 1.8. That further confirms the result quality of our approximate algorithm.

**Table 3: $\rho$ in several real-world graphs**

| datasets | $\bar{d_o}$ | $\rho$ | datasets | $\bar{d_o}$ | $\rho$ |
|---|---|---|---|---|---|
| it2004 | 27.9 | 1.43 | com-lj | 8.5 | 1.46 |
| cit-Patent | 2.8 | 1.63 | kron-log21 | 1.0 | 1.16 |

## 5 GRAPH ORDERING

A block gets a group of continuous vertices as a work set, so we can reorder vertices to assure resource preferences of all vertices in this group make up for each other. Based on the resource balance model in Section 3.2, we first prove the NP-hardness of the corresponding optimization problem, then propose a solution named A-order to achieve near-optimal performance at low cost.

## 5.1 Hardness Analysis

THEOREM 5.1. *Given a directed graph $\mathcal{G}$ obtained from some specific edge directing strategy, finding the optimal vertex ordering to minimize Equation (3) is an NP-hard problem.*

To prove the theorem, we consider a sub-problem of the problem in Equation (3) by setting $b = 2$ and $\lambda = 1$, and present a decision version (denoted as *DP*) of the sub-problem as follows. Theorem 5.1 holds if *DP* is NP-complete.

*DP*: Given two buckets $B_1$, $B_2$ and a vertex set V. Each vertex $v_i$ has computing intensity $c_i$ and memory intensity $m_i$. We want to know that whether there is a schedule of vertex dispatch, satisfying $C_1 = M_1$, $C_2 = M_2$ and $C_1 \le C_{max}$, $C_2 \le C_{max}$.

Proof. It's clear that $DP \in$ NP, because a schedule can be verified in polynomial time.

Then we will prove that the *partition problem* can be reduced to *DP* in polynomial time. Consider the following *partition problem.* We have a set of $2t$ elements ($t \in \mathbb{Z}^+$), the sum of which is $2S$, and $S\%t^2 = 0$. The $i^{th}$ elements of the set is $a_it + 1$, $a_i \in \mathbb{Z}^+$, and we want the set to be evenly separated into two sets, the sum of which is $S$. In the corresponding *DP* problem, there are $2t$ vertices: $m_i = a_it+1$, while $c_i$ = S/t and $C_{max} = S$. This transformation is evidently polynomial time.

Suppose that two subsets $V_1$ and $V_2$ satisfies the partition problem: the sums of elements in them are both $S$, we can know that the number of elements of both subsets are $t$, otherwise the sums can not be multiple of $t$. Therefore if we assign related vertices of $V_1$ to $B_1$, $C_1 = S = M_1$, and $C_2 = S = M_2$. On the other hand, consider a schedule satisfies the DP problem, saying that $C_i = M_i$ for both buckets. Knowing that $C_i$ must be multiple of $t$ ( because $c_i$ is multiple of $t$), $M_i$ is multiple of $t$, too. So the vertices number of both buckets is $t$. And consequently $C_1 = C_2$ = S = $M_1 = M_2$, so this schedule is also the answer of *partition problem.*

Then we can conclude that the *DP* is NP-complete, so the original problem is NP-hard. □

## 5.2 Problem Solution

Because the problem is NP-hard, we propose a heuristic algorithm. Note that the concrete form of the function $F_c$, $F_m$ and the parameter $\lambda$ are the prerequisites for the algorithm. We will discuss the parameter determination later in Section 5.3.

Generally, the algorithm employs a greedy strategy. The pseudocode is demonstrated in Algorithm 2. We say a vertex is memory-dominated (resp. computing-dominated) if binary search on its adjacency list needs more memory (resp. computational) resources. Generally speaking, if a vertex is memory-dominated, we put it into the bucket with the least memory resources demand. Firstly, all buckets are initialized with their *mem_sup* set to 0 (line 1). For each vertex $v$, we refer to $F_m(d_o(v)) - \lambda F_c(d_o(v))$ as memory superiority. We use variable *mem_sup* to denote the sum of all vertices' memory superiority in a bucket. Then all buckets are added into a minimum priority queue according to *mem_sup* (line 2). Next, all vertices are separated into two sets: memory-dominated vertices and computing-dominated vertices (line 3-4). Each memory-dominated vertex is put into the bucket at the top of the queue (line 6-7). The *mem_sup* of the bucket is updated accordingly (line 8). Then all buckets are made into a maximum priority queue according to *mem_sup* (line 10), and we process computing-dominated vertices in a corresponding way (line 11-15). All above operations aim at better resource usage, and then we sort vertices in every bucket (line 16-17) to assure that adjacency threads deal with lists with similar length. That's from consideration of workload balance. Finally we reorder the whole graph, ensuring that vertices in the same bucket have consecutive ID (line 18-21).

**Algorithm 2** A-order Algorithm

```
Input: graph G, $F_c$, $F_m$, $\lambda$
Output: reordered graph G'
1: $B = init\_all\_buckets(\ )$
2: $B.make\_min\_queue(\ )$
3: $V_{mem} \longleftarrow$ memory-dominated vertices
4: $V_{comp} \longleftarrow$ computing-dominated vertices
5: for all $v \in V_{mem}$ do
6:     $b \longleftarrow B.pop\_queue(\ )$
7:     $b.vertices.push\_back(v)$
8:     $b.mem\_sup += (F_m(d_o(v)) - \lambda F_c(d_o(v)))$
9:     $B.insert(b)$
10: $B.make\_max\_queue(\ )$
11: for all $v \in V_{comp}$ do
12:     $b \longleftarrow B.pop\_queue(\ )$
13:     $b.vertices.push\_back(v)$
14:     $b.mem\_sup += (F_m(d_o(v)) - \lambda F_c(d_o(v)))$
15:     $B.insert(b)$
16: for all $b \in B$ do
17:     sort vertices in b by out degree
18: $reorderIdx \longleftarrow 0$
19: for all $b \in B$, $v \in b.vertices$ do
20:     $v.id \longleftarrow reorderIdx$
21:     $reorderIdx \longleftarrow reorderIdx + 1$
```

Sorting vertices in a bucket can be replaced by grouping vertices with similar out-degree with hash functions, then the complexity of our algorithm is bounded by $O(|V|log|B|)$. Since |B|«|V|, the algorithm achieves near-linear complexity.

## 5.3 Parameter Determination

To solve the optimization problem, we firstly need to figure out all the variables: computing intensity $c$ (or $F_c$), memory intensity $m$ (or $F_m$) and parameter $\lambda$. Although varying for different triangle counting algorithms, they can be estimated in a similar method. In the following text, we will describe parameter determination by taking Hu's implementation [18] as an example.

**Functions $\mathbf{F_m}$ and $\mathbf{F_c}$** Shared memory bandwidth $BW$ is a straightforward measurement of memory access intensity, so we directly use it as our memory intensity. We can obtain the relation between $BW$ and the adjacency list length by running $nvprof$, as shown in Figure 7. The compute throughput (denoted as CT), i.e., the compare and arithmetic addition times in a fixed time window, can also be measured by experiments, which is also shown in Figure 7. And we use it as our computing intensity. In conclusion, we have

$$F_m = BW(d_o(v)), F_c = CT(d_o(v)) \tag{12}$$

And Figure 7 also verifies our statement that short lists are computing intensive while long lists are memory intensive, which provides an opportunity to balance resource usage by task schedule.

**Parameter $\lambda$** It is a parameter that can transform $F_c$ to $F_m$. It can be regarded as the ratio of maximum memory ability to maximum

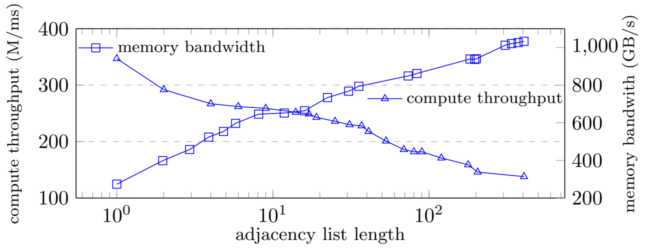

**Figure 7: Memory bandwidth and compute throughput with different adjacency list length**

computing ability. But we cannot get the exact maximum computing ability concerning a specific computational task. Hence, we design an experiment to estimate $\lambda$ as follows.

Firstly, we will give a definition of *balance point*, at which memory and computational resources are both fully utilized. Assume that we have two tasks, *Task* 1 and *Task* 2 in Figure 8, and the hardware could serve one memory unit and one computation unit in one period. *Task* 1 needs four periods, but computational resources in the last three periods are wasted. We increase computational needs (i.e. give $p_c$ times extra computation workload) progressively of *Task*1 to reach *Task* 2, which also needs four periods to finish, but both computational and memory resources are fully utilized. We say *Task* 2 reaches the balance point. Because of spare computation resources, the total time does not increase in this computation-increasing process until reaching the balance point, where any increase of computational or memory needs will immediately cost more time. In experiments we keep increasing $p_c$ until total time increases, which indicates reaching balance point. We use $m$ and $c$ to denote memory needs and computational needs of *Task* 1 respectively, then $c \times p_c$ denotes the computational needs in *Task* 2, which is balance point. So we have the following equation :

$$m = \lambda(p_c \times c) \tag{13}$$

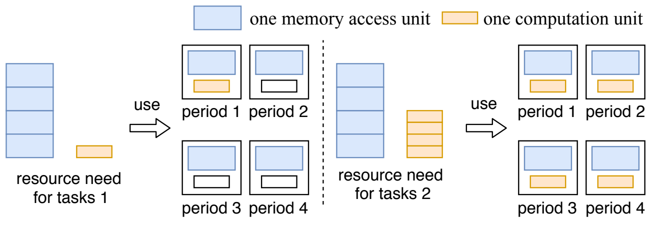

**Figure 8: Resource usage of two different tasks**

We carried out multiple experiments on several representative datasets to get $p_c$, $m$ and $c$ in different adjacency list lengths. According to our analysis, Equation (13) should be satisfied at balance point. And the functional image of $m$ related to $c \times p_c$ can be well fitted by a direct proportional function, as shown in Figure 9. In our experiment, the $\lambda$ is 0.332. Note that we only demonstrate the case of memory-dominated vertices, and it is vice versa for computing-dominated vertices.

## 6 EXPERIMENTS

In this section, we evaluate the effect of our preprocessing methods.

### 6.1 Experimental Settings

**Environments** : We use CUDA 8.0.61 toolkit and GCC 4.8.5 to compile all codes with *-O3* option. All experiments are carried out on a Linux server with the following configurations: Intel Xeon E5-2697 CPU, an 18-core processor; an NVIDIA Titan Xp, which has 12GB global memory and 3840 cuda cores.

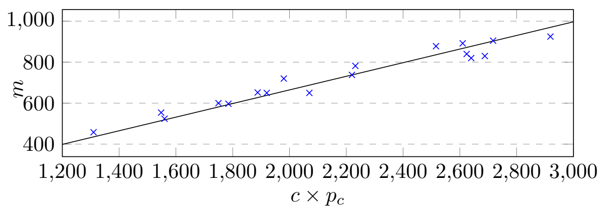

**Figure 9: Fitting function**

**Datasets** : Both real-world and synthetic datasets are used. We obtain real-world datasets from Stanford Network Analysis Project (SNAP)[3], HPEC graph challenge[4] and WebGraph [10][5], while synthetic datasets are obtained by Kronecker generator[6] and SNAP. We also use graph upscaling technique [27] to generate larger graphs from real-world graphs: we upscale *com-lj* and *cit-Patent* with the factor of 8. Table 4 shows statistics of all datasets we use, which covers many types of graphs such as social network, protein graphs, citation graphs, web graphs, BA graphs and ER graphs.

**Comparative Methods** : Since our method is a graph data preprocessing approach that is orthogonal to existing GPU triangle counting algorithms, to evaluate the effectiveness and usability of our method, we consider the following state-of-the-art GPU triangle counting algorithms.

(1) Gunrock [36]. It is a mature GPU graph processing library, which has been included into NVIDIA official acceleration library.
(2) TriCore [20]. This work uses a warp to deal with an edge, which adapts to SIMT features in GPU.
(3) Fox's work [13, 15]. These two papers use similar methods, which win graph challenge 2018's Innovation Awards and Finalists respectively.
(4) Bisson's work [8]. It matches computational resource with workload, which has a great influence on later works.
(5) Hu's work [18]. It proposes novel fine-grained workload distribution manner that is suitable for GPU architecture.

We implement four of above methods by ourselves except Gunrock, and our implementations [7] achieve similar performance as they reported in their original papers.

As baselines, we consider the degree-based ("D-direction" for short) for edge directing and original vertex order ("Origin" for short) for vertex ordering respectively. In the following experiments, our analytic model-based edge directing and vertex ordering strategies are denoted as "A-direction" and "A-order", respectively.

We will evaluate the effectiveness of our data preprocessing approaches in above-mentioned algorithms. Note that our A-direction strategy is based on the analysis of intra-block synchronize model. In the above five algorithms, only Bisson's and Hu's work adopt the explicit intra-block synchronization. Thus, we evaluate A-direction

[3]http://snap.stanford.edu/data/
[4]https://graphchallenge.mit.edu/data-sets
[5]http://law.di.unimi.it/datasets.php
[6]https://github.com/graph500/graph500
[7]We release all experiment codes, including comparative methods, at https://anonymous.4open.science/r/411937e1-e917-44e0-a6cb-5bf4bfb70369/

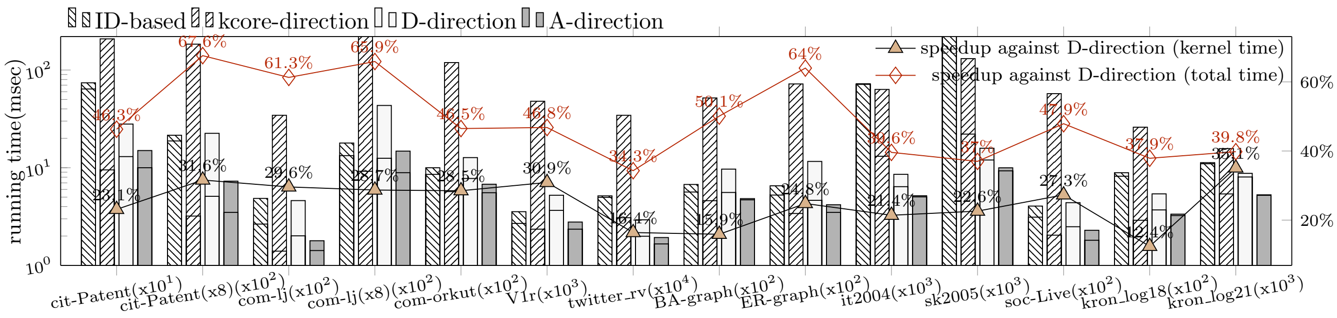

**Figure 10: Running time of different edge direction methods on Hu's work**[9]

in these two methods (Section 6.2). Except for Bisson's work[8], all other four algorithms adopt the binary search strategy for list intersection, which is suitable to our "A-order" scheme that is based on workload diversity analysis in binary search. Thus, we evaluate A-order in these four algorithms (Section 6.3). Since Hu's work adopts both intra-block synchronize model and binary search based list intersection, we measure the effectiveness of the two preprocessing strategies (A-direction and A-order) together in Section 6.4.

**Table 4: Datasets Infos**

| dataset | nodes | edges | triangles | types |
|---|---|---|---|---|
| cit-Patent | 6M | 17M | 7,515,023 | citation |
| cit-Patent(x8) | 21M | 128M | 17,597,025 | upscaled |
| kron-log18 | 25M | 25M | 281,814,846 | synthetic |
| com-lj | 4M | 34M | 177,820,130 | social |
| com-lj(x8) | 23M | 274M | 503,106,473 | upscaled |
| soc-Live | 5M | 69M | 285,730,264 | social |
| BA-graph | 1M | 90M | 24,675,343 | synthetic |
| ER-graph | 4M | 100M | 20,748 | synthetic |
| com-orkut | 3M | 117M | 627,584,181 | social |
| kron-log21 | 201M | 201M | 1,765,053,740 | synthetic |
| V1r | 214M | 465M | 49 | biology |
| twitter_rv | 62M | 1.5B | 34,824,916,864 | social |
| it2004 | 37M | 1.1B | 48,374,551,054 | web |
| sk2005 | 47M | 1.9B | 84,907,041,475 | web |

## 6.2 Edge-directing Strategy

In this subsection, we compare our A-direction scheme with the degree-based, ID-based and a kcore edge directing method [21]. The kcore direction is originally a peeling method about k-core decomposition problem. This method gives an order among vertices, which can be used for edge direction. It uses a flatten array for frequently updates, and it's efficient in CPU implementations. Note that we adopt the baseline vertex ordering (i.e., Origin) for all implementations.

**Evaluation of A-direction on Hu's algorithm.** The running times of four edge direction methods, i.e., ID-based, degree-based ("D-direction"), kcore-direction [21] and our method ("A-direction") are demonstrated by the "bars" of Figure 10. Note that the running time includes two parts: the upper part of the bar denotes the preprocessing time, and the bottom part denotes the GPU kernel running time. It is easy to conclude that both A-direction and D-direction are significantly faster than the ID-based scheme. The kernel of kcore direction performs better than that of D-direction on most datasets, while there are also datasets on which it performs worse. That's because the kcore direction focuses on the hierarchy of k-core decomposition instead of total order of all vertices, resulting in unstable performance. And the preprocessing for kcore direction is very time-consuming. The algorithm using flatten array [21] involves too many memory-write operations, making itself a burden of overall performance. And the flatten array makes it hard to parallelize this method. Therefore, we choose D-direction as the baseline. To compare our method with D-direction, we also show the speedup ratio of our method to D-direction in both GPU kernel time and total time (including preprocessing time) by separate lines in Figure 10. Our strategy outperforms D-direction on all datasets, achieving 12.4% ~ 35.1% improvement on kernel time and 34.3% ~ 67.6% improvement on total time.

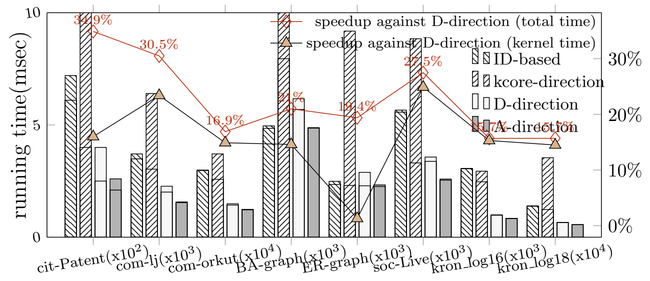

**Figure 11: Running time of different edge direction methods on Bisson's work**

**Evaluation of A-direction on Bisson's algorithm.** The experiment results are shown in Figure 11. We also compare the A-direction strategy with other three methods. In Bisson's algorithm, ID-based method works significantly worse than the degree-based methods, while our method still has 15.7% ~ 34.9% speedup than D-direction. The conclusion of kcore direction is the same with that on Hu's implementation. Bisson's algorithm cannot process large graphs such as *twitter_rv* and *V1r*, because its bitmaps require a lot of memory space. Figure 11 shows experimental results of all datasets that this algorithm can process. Generally, graphs with highly skewed degree distribution, such as most real-world graphs,

[8]Bisson's work uses bitmaps for list intersection
[9]In this figure and other following figures like this one, dataset label *cit-Patent* ($\times 10^1$) means the running time of this dataset is the value shown in bar scaled up by $10^1$.

| Datasets | Running times (ms) | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Origin | D-based | DFS | | BFS-R | | SlashBurn | | GRO | | A-order | | | |
| | kernel | kernel | kernel | total | kernel | total | kernel | total | kernel | total | kernel | speedup | total | speedup |
| cit-Patent | 130 | 151 | 122 | 786 | 120 | 2043 | 118 | 3234 | 123 | 10923 | 95 | 26.9% | 115 | 11.5% |
| cit-Patent(x8) | 510 | 526 | 483 | 8154 | 504 | 20544 | 489 | 138599 | 457 | 137827 | 340 | 33.3% | 445 | 12.8% |
| com-lj | 201 | 186 | 246 | 1211 | 236 | 1679 | 231 | 3877 | 264 | 59264 | 144 | 28.4% | 156 | 22.4% |
| com-lj(x8) | 1250 | 1269 | 1240 | 2226 | 1250 | 27710 | 1190 | 174380 | 1290 | 979290 | 890 | 28.8% | 965 | 22.8% |
| com-orkut | 776 | 729 | 770 | 3057 | 790 | 4093 | 760 | 4636 | 763 | 860763 | 674 | 13.1% | 684 | 11.9% |
| V1r | 3660 | 3900 | 1610 | 72030 | 2740 | 494440 | - | - | 5330 | 312330 | 1880 | 48.6% | 2496 | 31.8% |
| twitter_rv | 19890 | 32900 | 17670 | 89670 | 18200 | 195200 | - | - | - | - | 17410 | 12.5% | 17739 | 10.8% |
| BA-graph | 558 | 576 | 532 | 1868 | 530 | 4117 | 556 | 2895 | 528 | 1523528 | 480 | 14.0% | 497.9 | 10.8% |
| ER-graph | 464 | 465 | 455 | 3817 | 456 | 5165 | 471 | 3390 | 454 | 339454 | 420 | 9.5% | 434.9 | 6.3% |
| it2004 | 6400 | 6399 | 6190 | 68490 | 6200 | 43800 | 6330 | 627330 | - | - | 5600 | 12.5% | 5709 | 10.8% |
| sk2005 | 12000 | 11250 | 10950 | 1011770 | 10280 | 60480 | - | - | - | - | 9400 | 21.7% | 9517 | 20.7% |
| soc-Live | 246 | 326 | 151 | 1647 | 146 | 5011 | 158 | 1008 | 126 | 148126 | 176 | 28.5% | 191 | 22.4% |
| kron_log18 | 372 | 452 | 372 | 1361 | 373 | 960 | 367 | 6788 | 369 | 424369 | 320 | 14.0% | 330 | 11.3% |
| kron_log21 | 8040 | 9611 | 5250 | 16450 | 5240 | 13065 | 5100 | 126100 | - | - | 5020 | 37.6% | 5073 | 36.9% |

**Table 5: Different reorder strategies on Hu’s fine-grained implementation(“-” means the reordering time exceeds an hour.)**

| Datasets | Running times (ms) | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Origin | D-based | DFS | | BFS-R | | SlashBurn | | GRO | | A-order | | | |
| | kernel | kernel | kernel | total | kernel | total | kernel | total | kernel | total | kernel | speedup | total | speedup |
| cit-Patent | 86 | 103 | 90 | 754 | 99 | 2022 | 95 | 3211 | 93 | 10893 | 62 | 27.9% | 82 | 4.65% |
| cit-Patent(x8) | 435 | 619 | 384 | 8055 | 373 | 20413 | 381 | 138451 | 382 | 137752 | 310 | 28.7% | 415 | 4.6% |
| com-lj | 150 | 271 | 142 | 1107 | 162 | 1611 | 170 | 3816 | 163 | 59163 | 124 | 17.3% | 136 | 9.3% |
| com-lj(x8) | 938 | 929 | 930 | 10792 | 887 | 27337 | 909 | 174099 | 1110 | 979110 | 780 | 16.8% | 855 | 8.9% |
| com-orkut | 630 | 2729 | 570 | 2297 | 600 | 3903 | 564 | 4440 | 620 | 860620 | 550 | 12.7% | 560 | 11.1% |
| V1r | 1480 | 1310 | 1370 | 71790 | 1390 | 493090 | - | - | 1380 | 308380 | 770 | 50.0% | 1386 | 6.35% |
| twitter_rv | 27800 | 19090 | 24000 | 96000 | 21000 | 198000 | - | - | - | - | 17800 | 36.0% | 18129 | 34.8% |
| BA-graph | 560 | 505 | 513 | 1849 | 525 | 4112 | 536 | 2875 | 514 | 1523514 | 480 | 14.3% | 498 | 11.1% |
| ER-graph | 390 | 363 | 413 | 3775 | 404 | 5113 | 443 | 3362 | 389 | 1523389 | 300 | 23.1% | 314.9 | 19.3% |
| it2004 | 8350 | * | 9060 | 71360 | 8420 | 46020 | 10720 | 631720 | - | - | 6260 | 25.0% | 6369 | 23.7% |
| sk2005 | 17520 | * | 19060 | 1019880 | 18010 | 68210 | - | - | - | - | 13690 | 21.9% | 13870 | 21.2% |
| soc-Live | 183 | 380 | 199 | 1695 | 198 | 5063 | 195 | 1045 | 209 | 148209 | 156 | 14.8% | 171 | 6.6% |
| kron_log18 | 380 | 1515 | 330 | 1319 | 340 | 927 | 300 | 6721 | 330 | 424330 | 320 | 15.8% | 330 | 13.2% |
| kron_log21 | 5100 | 37641 | 4060 | 15260 | 3991 | 11816 | 3653 | 124653 | - | - | 4200 | 17.7% | 4253 | 16.6% |

**Table 6: Different reorder strategies on TriCore implementation(“-” means the reordering time exceeds an hour, and “*” means the running time of the kernel exceeds twenty minutes.)**

benefit more from our preprocessing method. And we add an ER-graph to show the effect of our method on non skewed-degree graphs. The kernel performance of ER-graph has few improvements, and we consider it as acceptable.

In conclusion, our edge direction method (“A-direction”) achieves significant performance improvements on both kernel running time and total time compared with the D-direction method, which is widely used in state-of-the-art GPU triangle counting algoroithms.

## 6.3 Graph Ordering Strategy

The vertex ordering strategy aims to address the problem of workload diversity. To isolate the effect of our vertex-reordering strategy, we adopt the D-direction for all the following experiments. We will compare our analytic model-based vertex ordering (“A-order”) with the original order and existing graph ordering methods, including DFS [30], BFS-R [9], SlashBurn [23] and GRO [16].

**Evaluation of A-order on Hu’s algorithm and TriCore.** Vertices are reorder units for both TriCore [20] and Hu’s work [18], and they both adopt binary searches for list intersection. Thus, they are both suitable for our analytic model in Section 3.2.

So far, we have three vertex reordering strategies: original vertex order (Origin), degree-based (D-order) and our method (A-order). Graph reordering is a well studied problem in the literature, thus we also compare our strategy with the following methods: DFS [30], BFS-R [9], SlashBurn [23] and GRO [16]. DFS reorders vertices according to depth-first traversal; BFS-R performs BFS to find a vertex with the largest depth, then performs BFS from it until half of all vertices are visited. This method recursively partitions the graph vertices to build a separator tree, according to which all nodes are reordered; SlashBurn groups vertices in the same adjacency list and gives them continuous order; GRO proposes the notion of compactness scores to make neighbor vertices consecutive in ID, and proposes a greedy algorithm to minimize it.

Tables 5 and 6 show the performance of different vertex orderings on Hu’s algorithm and TriCore. “kernel” in the table denotes kernel time, “total” denotes the sum of kernel time and reordering time. Here the “speedup” is obtained by comparing A-order with original order on both kernel time and total time. Note that the total time is the same with kernel time in “Origin”, since it does not need extra vertex ordering. As we conclude, the D-order strategy is generally worse than the original one because of resource usage imbalance. All four state-of-the-art reordering strategies improve the performance in most datatsets to some extend. However, their preprocessing time far surpasses their kernel time, making total time unacceptable. That’s why we need a lightweight reordering strategy. As we can see, our method achieves 9.5% ~ 48.6%

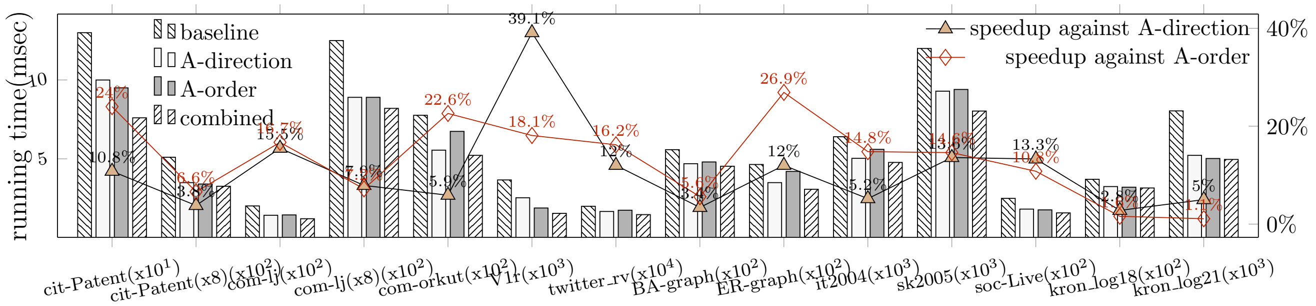

**Figure 12: Combining two models on Hu's work**

speedup of kernel time than original order, which is better than all the other strategies, and 6.3% ~ 36.9% speedup even including preprocessing time. Table 6 shows experiment results of TriCore with different reordering strategies. The conclusion of four state-of-the-art reordering strategies is the same with Table 5, and our method ("A-order") achieves 12.7% ~ 50.0% speed up of kernel time, and 4.6% ~ 34.8% speed up of total time.

**Evaluation of A-order on Gunrock**. Gunrock [36] uses both binary search and sort-merge for list intersections. As analyzed in Section 3.2, we can conclude that GPU binary search based list intersection has different resource preferences (computing intensive or memory intensive) on short and long lists. In GPU triangle counting algorithms, binary search is more efficient than sort-merge [4, 18, 20] and thus more popular in existing algorithms. Since Gunrock is a general GPU graph processing library, it provides two kinds of list intersection algorithms (binary search and sort-merge). The sort-merge implementation in Gunrock also employs the same resource preferences with binary search because of coalesced memory access feature of warps. Gunrock crashes on big datasets such as *twitter_rv* and *V1r*, thus, we report evaluation results on all datasets that can be processed by it. Comparing with the original order, we find that our A-order strategy achieves 4.1% ~ 82.5% performance improvement in total time, and even better in kernel time (see Figure 13). We have mentioned that the baseline, i.e., original order, does not have preprocess time, so the overall performance of several datasets (such as *cit-Patent* and *com-lj*) has minor improvements because the ordering preprocessing is too time-consuming compared with kernel time. And kernel promotions on those datasets are satisfying. Obviously, the degree-based order has the worst performance due to more resource conflicts. We do not compare with other ordering strategies in Tables 5 and 6, since their preprocessing are too time-consuming, making them impractical in triangle counting tasks.

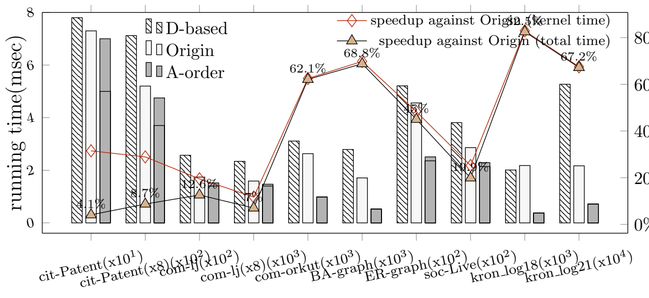

**Figure 13: Vertex ordering results of Gunrock**

**Evaluation of A-order on Fox's algorithm**. Edge is the reorder unit of Fox's algorithm [13], because edges of a vertex are separated according to their work complexity in this method. Blocks work on edges instead of vertices, and "reordering edges" refers to changing edge sets of blocks. The reordering progress is similar to vertex reordering. Memory intensive and computing intensive operations are defined analogous to Hu's implementation [18], except that we consider resource usage tensity of edges instead of vertices. Figure 14 shows results of our method and original edge distribution in several datasets. Similarly, there are some datasets that this implementation can not work on, and we give all datasets that this method is capable of processing. Finally, we achieve 6.0% ~ 45.2% performance improvement in total time, and higher improvements in kernel time.

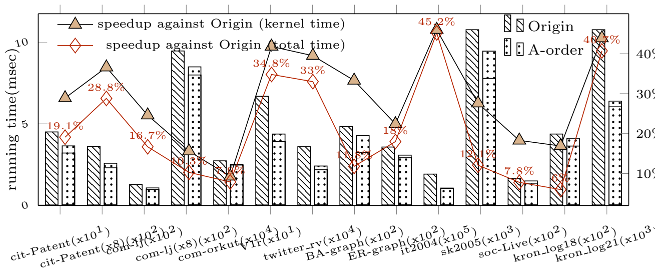

**Figure 14: Vertex ordering results of Fox's work**

## 6.4 Evaluating the Combined Approach

Since some algorithms include both intra-block synchronization and binary search based list intersection, such as Hu's work [18], we combine both A-direction and A-order on it. And the evaluation results are shown in Figure 12. The lines show performance improvement of the combined method to only adopting A-direction or A-order. Generally, the combined approach can speed up the overall running time by 10.7% on average compared with A-direction only, and 13.3% on average compared with A-order. This further confirms the effectiveness of our analytic model based preprocessing.

## 7 CONCLUSION

Triangle counting is a fundamental graph computational task due to its wide applications. Meanwhile, GPU-based implementations have been extensively studied in the literature. This paper does not intend to propose a new algorithm. Instead, we study the workload imbalance and diversity problems by abstracting common models from state-of-the-art triangle counting algorithms. Based on our proposed analytic models, we propose model-guided edge directing and vertex ordering strategies to preprocess the graph data. The two strategies optimize the workload balance and further improve the degree of parallelism. Without revising any existing algorithm, we significantly improve the performance of these algorithms over both large real-world and synthetic graph datasets.

## ACKNOWLEDGMENTS

This work was supported by The National Key Research and Development Program of China under grant 2018YFB1003504 and NSFC under grant 61932001, 61961130390, U20A20174. This work was also supported by Beijing Academy of Artificial Intelligence (BAAI). The corresponding author of this paper is Lei Zou (zoulei@pku.edu.cn).

## REFERENCES

[1] William Aiello, Fan Chung, and Linyuan Lu. 2000. A random graph model for massive graphs. In *Proceedings of the thirty-second annual ACM symposium on Theory of computing*. 171–180.

[2] Noga Alon, Raphael Yuster, and Uri Zwick. 1997. Finding and Counting Given Length Cycles. *Algorithmica* 17, 3 (1997), 209–223. https://doi.org/10.1007/BF02523189

[3] Marcos Amaris, Daniel Cordeiro, Alfredo Goldman, and Raphael Y de Camargo. 2015. A simple bsp-based model to predict execution time in gpu applications. In *2015 IEEE 22nd International Conference on High Performance Computing (HiPC)*. IEEE, 285–294.

[4] Naiyong Ao, Fan Zhang, Di Wu, Douglas S Stones, Gang Wang, Xiaoguang Liu, Jing Liu, and Sheng Lin. 2011. Efficient parallel lists intersection and index compression algorithms using graphics processing units. *Proceedings of the VLDB Endowment* 4, 8 (2011), 470–481.

[5] Shaikh Arifuzzaman, Maleq Khan, and Madhav V. Marathe. 2013. PATRIC: a parallel algorithm for counting triangles in massive networks. In *22nd ACM International Conference on Information and Knowledge Management, CIKM'13, San Francisco, CA, USA, October 27 - November 1, 2013*, Qi He, Arun Iyengar, Wolfgang Nejdl, Jian Pei, and Rajeev Rastogi (Eds.). ACM, 529–538. https://doi.org/10.1145/2505515.2505545

[6] Ariful Azad, Aydin Buluç, and John R. Gilbert. 2015. Parallel Triangle Counting and Enumeration Using Matrix Algebra. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop, IPDPS 2015, Hyderabad, India, May 25-29, 2015*. IEEE Computer Society, 804–811. https://doi.org/10.1109/IPDPSW.2015.75

[7] Vladimir Batagelj and Andrej Mrvar. 2001. A subquadratic triad census algorithm for large sparse networks with small maximum degree. *Soc. Networks* 23, 3 (2001), 237–243. https://doi.org/10.1016/S0378-8733(01)00035-1

[8] Mauro Bisson and Massimiliano Fatica. 2017. High Performance Exact Triangle Counting on GPUs. *IEEE Transactions on Parallel and Distributed Systems* 28, 12 (2017), 3501–3510.

[9] Daniel K. Blandford, Guy E. Blelloch, and Ian A. Kash. 2003. Compact representations of separable graphs. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms, January 12-14, 2003, Baltimore, Maryland, USA*. ACM/SIAM, 679–688. http://dl.acm.org/citation.cfm?id=644108.644219

[10] Paolo Boldi and Sebastiano Vigna. 2004. The WebGraph Framework I: Compression Techniques. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*. ACM Press, Manhattan, USA, 595–601.

[11] Federico Busato and Nicola Bombieri. 2015. An efficient implementation of the Bellman-Ford algorithm for Kepler GPU architectures. *IEEE Transactions on Parallel and Distributed Systems* 27, 8 (2015), 2222–2233.

[12] Yulin Che, Zhuohang Lai, Shixuan Sun, Yue Wang, and Qiong Luo. 2020. Accelerating Truss Decomposition on Heterogeneous Processors. *Proc. VLDB Endow.* 13, 10 (2020), 1751–1764. http://www.vldb.org/pvldb/vol13/p1751-che.pdf

[13] James Fox, Oded Green, Kasimir Gabert, Xiaojing An, and David A Bader. 2018. Fast and Adaptive List Intersections on the GPU. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*. IEEE, 1–7.

[14] Ilias Giechaskiel, George Panagopoulos, and Eiko Yoneki. 2015. PDTL: Parallel and Distributed Triangle Listing for Massive Graphs. In *44th International Conference on Parallel Processing, ICPP 2015, Beijing, China, September 1-4, 2015*. IEEE Computer Society, 370–379. https://doi.org/10.1109/ICPP.2015.46

[15] Oded Green, James Fox, Alex Watkins, Alok Tripathy, Kasimir Gabert, Euna Kim, Xiaojing An, Kumar Aatish, and David A Bader. 2018. Logarithmic Radix Binning and Vectorized Triangle Counting. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*. IEEE, 1–7.

[16] Shuo Han, Lei Zou, and Jeffrey Xu Yu. 2018. Speeding Up Set Intersections in Graph Algorithms using SIMD Instructions. In *Proceedings of the 2018 International Conference on Management of Data*. ACM, 1587–1602.

[17] Sunpyo Hong and Hyesoon Kim. 2009. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. In *ACM SIGARCH Computer Architecture News*, Vol. 37. ACM, 152–163.

[18] Lin Hu, Naiqing Guan, and Lei Zou. 2019. Triangle counting on GPU using fine-grained task distribution. In *2019 IEEE 35th International Conference on Data Engineering Workshops (ICDEW)*. IEEE, 225–232.

[19] Yang Hu, Pradeep Kumar, Guy Swope, and H Howie Huang. 2017. Trix: Triangle counting at extreme scale. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–7.

[20] Yang Hu, Hang Liu, and H Howie Huang. 2018. Tricore: Parallel triangle counting on gpus. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 171–182.

[21] Wissam Khaouid, Marina Barsky, S. Venkatesh, and Alex Thomo. 2015. K-Core Decomposition of Large Networks on a Single PC. *Proc. VLDB Endow.* 9, 1 (2015), 13–23. https://doi.org/10.14778/2850469.2850471

[22] Tamara G. Kolda, Ali Pinar, Todd D. Plantenga, C. Seshadhri, and Christine Task. 2014. Counting Triangles in Massive Graphs with MapReduce. *SIAM J. Scientific Computing* 36, 5 (2014). https://doi.org/10.1137/13090729X

[23] Yongsub Lim, U Kang, and Christos Faloutsos. 2014. SlashBurn: Graph Compression and Mining beyond Caveman Communities. *IEEE Trans. Knowl. Data Eng.* 26, 12 (2014), 3077–3089. https://doi.org/10.1109/TKDE.2014.2320716

[24] Hang Liu, H Howie Huang, and Yang Hu. 2016. ibfs: Concurrent breadth-first search on gpus. In *Proceedings of the 2016 International Conference on Management of Data*. ACM, 403–416.

[25] Lin Ma and Roger D Chamberlain. 2012. A performance model for memory bandwidth constrained applications on graphics engines. In *2012 IEEE 23rd International Conference on Application-Specific Systems, Architectures and Processors*. IEEE, 24–31.

[26] Mark EJ Newman. 2005. Power laws, Pareto distributions and Zipf's law. *Contemporary physics* 46, 5 (2005), 323–351.

[27] Himchan Park and Min-Soo Kim. 2018. EvoGraph: An Effective and Efficient Graph Upscaling Method for Preserving Graph Properties. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD 2018, London, UK, August 19-23, 2018*, Yike Guo and Faisal Farooq (Eds.). ACM, 2051–2059. https://doi.org/10.1145/3219819.3220123

[28] Adam Polak. 2016. Counting triangles in large graphs on GPU. In *Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International*. IEEE, 740–746.

[29] Thomas Schank and Dorothea Wagner. 2005. Finding, Counting and Listing All Triangles in Large Graphs, an Experimental Study. In *Experimental and Efficient Algorithms, 4th InternationalWorkshop, WEA 2005, Santorini Island, Greece, May 10-13, 2005, Proceedings (Lecture Notes in Computer Science)*, Sotiris E. Nikoletseas (Ed.), Vol. 3503. Springer, 606–609. https://doi.org/10.1007/11427186_54

[30] Julian Shun. 2017. *Shared-memory parallelism can be simple, fast, and scalable.* PUB7255 Association for Computing Machinery and Morgan & Claypool.

[31] Julian Shun and Kanat Tangwongsan. 2015. Multicore triangle computations without tuning. In *Data Engineering (ICDE), 2015 IEEE 31st International Conference on*. IEEE, 149–160.

[32] Ha-Nguyen Tran, Jung-jae Kim, and Bingsheng He. 2015. Fast subgraph matching on large graphs using graphics processors. In *International Conference on Database Systems for Advanced Applications*. Springer, 299–315.

[33] Charalampos E Tsourakakis, Petros Drineas, Eirinaios Michelakis, Ioannis Koutis, and Christos Faloutsos. 2011. Spectral counting of triangles via element-wise sparsification and triangle-based link recommendation. *Social Network Analysis and Mining* 1, 2 (2011), 75–81.

[34] Jia Wang and James Cheng. 2012. Truss decomposition in massive networks. *arXiv preprint arXiv:1205.6693* (2012).

[35] Leyuan Wang, Yangzihao Wang, Carl Yang, and John D Owens. 2016. A comparative study on exact triangle counting algorithms on the gpu. In *Proceedings of the ACM Workshop on High Performance Graph Processing*. ACM, 1–8.

[36] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D Owens. 2016. Gunrock: A high-performance graph processing library on the GPU. In *ACM SIGPLAN Notices*, Vol. 51. ACM, 11.

[37] Duncan J Watts and Steven H Strogatz. 1998. Collective dynamics of 'small-world'networks. *nature* 393, 6684 (1998), 440.

[38] Michael M. Wolf, Mehmet Deveci, Jonathan W. Berry, Simon D. Hammond, and Sivasankaran Rajamanickam. 2017. Fast linear algebra-based triangle counting with KokkosKernels. In *2017 IEEE High Performance Extreme Computing Conference, HPEC 2017, Waltham, MA, USA, September 12-14, 2017*. IEEE, 1–7. https://doi.org/10.1109/HPEC.2017.8091043

[39] Xintian Yang, Srinivasan Parthasarathy, and Ponnuswamy Sadayappan. 2011. Fast sparse matrix-vector multiplication on GPUs: implications for graph mining. *Proceedings of the VLDB Endowment* 4, 4 (2011), 231–242.

[40] Abdurrahman Yasar, Sivasankaran Rajamanickam, Michael M. Wolf, Jonathan W. Berry, and Ümit V. Çatalyürek. 2018. Fast Triangle Counting Using Cilk. In *2018 IEEE High Performance Extreme Computing Conference, HPEC 2018, Waltham, MA, USA, September 25-27, 2018*. IEEE, 1–7. https://doi.org/10.1109/HPEC.2018.8547563