Accelerating Triangle Counting on GPU

Anonymous Author(s)

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 a bunch of 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 exactly affect the performance. Then, due to the NP-complete 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 [7, 11, 24, 27], whose task is computing the total number of triangles in a graph, lays the foundation for various graph problems, such as k-truss [31], clustering coefficient [34], and link recommendation [29].

With the scale of graph getting larger continuously, lots of existing works resort to new hardware with great parallel processing ability (such as GPU) to efficiently address graph computation tasks, such as BFS [20], SSSP [9], and subgraph isomorphism [28]. Triangle counting works on GPU [10, 15, 17, 24] also achieve great acceleration, specifically, 9∼260x faster compared with serial implementations on CPU [32]. In this paper, we do not intend to propose a new GPU triangle counting algorithm. Instead, we find some common computing patterns in GPU triangle counting algorithms, such as intra-block synchronization and binary search-based list intersection (see Section 3), and aim to optimize them by preprocessing graph data. Our work is inspired by an interesting observation: for the same data graph and GPU triangle-counting algorithm, different graph preprocessings result in significantly different performances (see Table 2). Therefore, we concentrate ourselves on a lightweight data preprocessing strategy that can benefit a group of relevant GPU triangle counting algorithms. For any triangle counting algorithm, there are two common data preprocessings: edge directing and vertex ordering.

Edge Directing. In most triangle counting algorithms, an undirected graph is firstly transformed into a directed one to avoid redundant computation [15, 24]. 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 studies the edge directing problem based on an analytic model; Also, no prior study tries to figure out whether there exists better edge directing strategies. We will study this problem in Section 4.

Vertex Ordering. The basic programmable unit in GPU is a *block* that has many parallel-processing threads. A batch of computation tasks of a block usually has different computational and memory resource preferences. Given the order of vertices, consecutive vertices are grouped and assigned to the same block. In other words, vertex ordering strategy determines the task assignment in GPU, and different assignments lead to different resource usage, as well as different running times. We will study this problem in Section 5.

The above observation tells us an important fact: different data preprocessings result in different triangle-counting performances. However, there are two important issues to be addressed: (1) how to measure the GPU triangle counting performance with regard to different data preprocessings? Obviously, the quantitative analysis model about data preprocessings is desirable. (2) what is the hardness of computing optimal data preprocessing according to the analytic model? If the complexity is high, how to design a lightweight approximate algorithm? Apparently, the performance-guarantee approximate algorithm is enticing. We will study these problems in this work.

Edge directing and vertex ordering have direct influence on GPU workload imbalance and diversity, respectively. To figure out how they work, we propose two analytic models, which originate from two common computing patterns (intra-block synchronization and binary search-based list intersection) in triangle counting algorithms on GPU.

Intra-block Synchronization & Workload Imbalance. In many GPU triangle counting algorithms [7, 15], threads insides a block often need to synchronize in a BSP (Bulk Synchronous Parallel) [30] model , such as setting global variables or perfecting data to shared memory. But workloads between two synchronization steps of different threads usually vary a lot, and they depend on the number of each vertex's out-neighbors (i.e., vertex out-degree). The skewed degree distribution exacerbates the workload imbalance. Fortunately, in GPU-based triangle counting, edge directing strategy provides an opportunity to change the out-degree distribution so as to balance workloads. Degree-based directing method is a widely used heuristic solution, but no analytic model about edge directing is proposed in the literature. A better analytic model-guided directing scheme is one of our contributions.

Binary Search-based List Intersection & Workload Diversity. Neighbor list intersection accounts for 94.3% of the whole triangle-counting running time [13], and binary search is proved to be a better choice than sort-merge on GPU [4, 15–17]. However, we find that the same binary search plays different workload features in GPU triangle counting algorithms. Binary search on long neighbor list is memory access intensive, but that on short one is computing intensive. We will discuss more details in Section 3.

Stream multiprocesser (SM) is a hardware-level basic unit of GPU which has its own cores and fast access memory. Block is programmer-level unit, and tasks in a block will only be dispatched to one SM. A good block task assignment is to group a batch of tasks with different resource preferences (i.e., computing intensive and memory intensive workloads) together. In this way, the thread schedule mechanism will make the full use of two kinds of resources in an SM for parallel processing. Otherwise, if tasks with similar workloads (i.e., the same resource preference) are assigned to a block, more threads have to be suspended due to memory or computing conflict, and the other resource will be idle and wasted in SM. Therefore, for binary search, it is better to group tasks with long and short lists together for task assignment. To achieve that, we propose an analytic model-based vertex reordering scheme.

To summarize, we aim to find optimal edge directing to balance workloads, and the optimal vertex ordering to maximize the degree of parallelism. Unfortunately, both problems are NP-complete (Theorems 4.1 and 5.1). In practice, the data preprocessing time should also be considered; otherwise, even though we speed up GPU kernel running time, the whole performance is sacrificed. Therefore, we propose lightweight approximate algorithms to find good solutions for edge directing and vertex ordering. Extensive experiments on both real and synthetic datasets confirm that our Analytic methods (called A-direction and A-order) can accelerate GPU kernel running time and total time (including our data preprocessing step) is accelerated by up to 82%. Last but not least, our acceleration method can boost multiple state-of-the-art GPU triangle counting algorithms without changing their implementations, which proves the usability of our method.

Generally, we have made the following contributions:

- We present two run-time models abstracted from common computing patterns in several state-of-the-art triangle counting implementations, considering GPU architecture features. These two models measure the performance with regard to workload imbalance and diversity, from a mathematical analysis perspective.
- To alleviate workload imbalance, we aim to find an optimal edge directing scheme based on our analytic model. Due to NP-hardness of this problem, we propose a lightweight (linear) approximate algorithm (A-direction) with performance guarantee.
- Considering workload diversity, we reorder vertices to implement a better task assignment approach (A-order) to maximize the degree of parallelism. Specifically, we formalize vertex reordering as a model optimization problem.
- We conduct extensive experiments to verify 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 the GPU architecture from both hardware and software perspectives.

Software. CUDA (Compute Unified Device Archtecture) is the most popular programming language of GPU, in which a *block* is a 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. Workload of different threads should be balanced, because imbalance workload among threads will lead to severe thread divergence.

Hardware. Similar to CPU, GPU has its own main memory and cache. Global memory 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 contains many cores to run hundreds of threads in parallel. It also has its fast-access memory, i.e., the *shared memory*, which is programmable but with limited space. Therefore, a carefully designed memory strategies will lead to better performance. In fact, SM can be seen as a match of *block*, because a block will be assigned to only one SM in run-time. 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 in one memory transaction, we can achieve better efficiency because of much fewer memory accesses.

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.*

Triangle counting is an important problem in graph mining, which lays the foundation for many other graph analysis, such as k-truss [31], clustering coefficient [34], and link recommendation [29]. We 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 [24] uses a thread to deal with an edge. Wang et al. implement the algorithm [32] using Gunrock [33] library with the same granularity. To better balance workload of different threads, Green et al. present a method [10, 11] that estimates workload of each edge in advance and then assigns threads to edges accordingly. Bission et al. [7] come up with an implementation which uses a block to deal with a vertex. This method uses bitmap in shared memory or global memory for fast look up of a list. And in TriCore [17], each warp is dispatched to deal with an edge, which makes full use of SIMT features. Hu et al. [15] present a finer-grained workload distribution method, in which a thread checks if a wedge "u-v-w" forms a triangle. It achieves better workload balance by fine-grained workload distribution.

CPU implementations. Algorithms of triangle counting on CPU can be mainly divided into three categories [25]: node-iterator [2], edge-iterator [6] and forward algorithm [25]. Node-iterator algorithm iterates over all nodes and tests for each pair of neighbors if they are connected by an edge; edge-iterator iterates over all edges and compares the adjacency data structure of both incident nodes, and forward algorithm is a refinement of edge-iterator. Besides, there are some methods based on map-reduce [18] and matrix multiplication [5, 35, 37]. Some algorithms are carried out on multicore platform and Shun's work [27] is rather representative in them. It performs triangle counting algorithm based on merging and hashing on multi-core architecture and achieves great speedup compared with single-core algorithms. In comparison, for GPU triangle counting, due to SIMT (Single Instruction Multiple Threads) execution and jointly memory access of a warp, we have to pay more attention to workload balance and coalesced memory access. That is different from the multi-core architecture.

Note that list intersection accounts for major time cost in triangle counting no matter in GPU-based or CPU-based implementations [13]. Generally, there are mainly two ways for list intersection: binary search [10, 11, 15–17, 33] and sort-merge [10, 11, 24, 33]. In GPU triangle counting, binary search is proven better than merge-based list intersection because of better independency and larger degree of parallelism [4, 15–17].

2.2.2 Analytic Model for GPU Computing. Existing analytic models for GPU computing mainly focus on performance predictions. For example, Hong et al. [14] propose a prediction model for algorithms on GPU, which introduces two metrics for judging resource preferences of parallel programs, and estimating the number of memory requests by considering the number of running threads and memory bandwidth. Amaris et al. [3] also propose an execution time prediction model, by considering computation, memory access and cache usage. Ma et al. [21] design a performance analytic model for memory-limited kernels, focusing on tuning various configuration parameters such as block number in kernel and thread number in block. Yang et al. [36] work on matrix multiplication. They build lookup tables between tile shape and performance, and use that to estimate the optimal workload size in each tile. In this paper, we propose our analytic models to quantify the performance effects with regard to different preprocessings. Furthermore, our approach can speed up a bunch of related algorithms, and is not designed for one specific algorithm. To the best of our knowledge, no existing work studies this problem. Table 1 lists the notations that are frequently used in the remainder of the paper.

Table 1: Frequently Used Notations

variable	description
Graph attributes	
u, v, w	three vertices for a triangle
δ_{uv}	variable denoting edge direction between u,v
$d(u)$	vertex u's degree in undirected graphs
$\widetilde{d}(u)$	vertex u's outgoing degree in directed graphs
$\widetilde{d}_{avg}$	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 $\widetilde{d}_{avg}$
v_n	vertex with degree smaller than $\widetilde{d}_{avg}$
V_n	vertices set of v_n
V_c	vertices set of v_c
ρ	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 $\widetilde{d}(v)$ to c and m, respectively
λ	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 ANALYTIC MODELS

In this section, we introduce two common computing patterns, *intra-block synchronization* and *binary search*, by analyzing several state-of-the-art triangle counting implementations on GPU architecture. Two analytic models are abstracted based on those patterns, then we bring up two cost functions to precisely describe the influence of edge directing and graph ordering.

3.1 Intra-block BSP Model

3.1.1 Synchronize computing. Synchronization among threads is common on GPU. Threads often share workspace in memory, thus synchronization is necessary for consistency access to memory. Bission's work [7] and Hu's fine-grained implementation [15] 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.

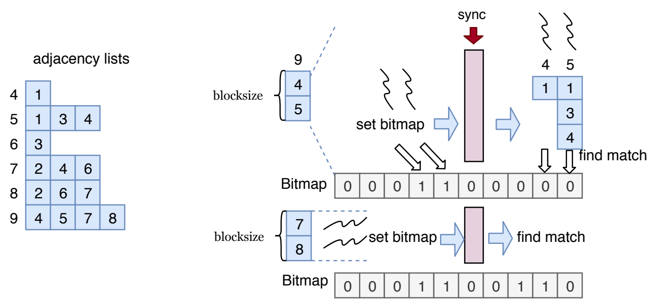

Figure 1: Running process of Bission's work

In Bission's work [7], a block is responsible for the whole triangle counting tasks of a vertex. Consider the example shown in Figure 1, where the block handles the adjacency list of vertex 9. Assume that a block has only two threads. Each thread is in charge of one vertex in the adjacency list, doing intersection of this vertex's adjacency list with that of vertex 9. They use global memory bitmaps for fast look up of target elements in the adjacency list. Two threads firstly set the their current vertices to 1 in the bitmap of vertex 9's adjacency list. Then follows a *synchronization* in the whole block to make sure all threads finish setting the bitmap. After that, threads use the bitmap to find matches for every element in their adjacency lists. Threads inside the block will move to next group of vertex 9's neighbors and continue the procedure until all neighbors of vertex 9 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. Length of adjacency lists may vary a lot, leading to workload imbalance among threads.

In Hu's work [15], 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, Figure 2 shows how *thread* 0 gets its work. 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.

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

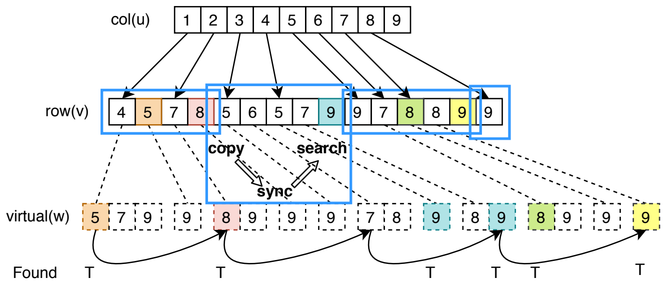

Figure 2: Running process of Hu's fine-grained method

3.1.2 Model description. We note that synchronization plays a key rule in the above algorithms, which assures consistency parallel access to global memory and shared memory among threads. This pattern can be modeled as BSP 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.

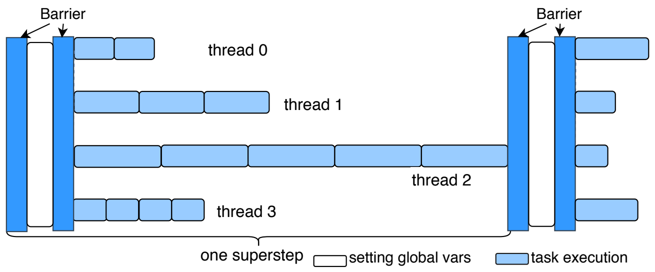

Figure 3: Intra-block BSP model

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 the adjacency list length of each vertex (i.e., vertex out-degree). For example, in Bission's work [7], threads of vertices with long adjacency lists have to check the bitmap more times; while in Hu's work [15], 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 [22], which definitely leads to severe workload imbalance. On the other hand, in all triangle counting implementations, an undirected graph is firstly transformed into a directed one to avoid redundancy computation. This preprocessing provides an opportunity to change the vertex out-degree distribution.

R-strategy	D-order	A-order	Original order		
D-strategy	D-direction			ID-based	A-direction
gowalla	26	7	9	13	6
cit-patent	4900	104	130	648	102
roadcentral	499	420	463	996	382
kron-log21	9611	5020	8042	10982	5230

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

3.1.3 Experimental rationality. We use Hu's work [15] 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 [15], 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 (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 ids to those of large ids. Furthermore, our A-direction strategy (the last column in Table 2) can further improve the performance by 17.4% to 34.9% comparing with D-direction.

3.1.4 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 $\widetilde{d}(u)$ stands for the out-degree of vertex u in the directed graph, and $\widetilde{d}_{avg}$ means the average of out-degree in the directed graph.

$$C(\mathcal{P}) = \sum_{u \in V(\mathcal{G})} |\widetilde{d}(u) - \widetilde{d}_{avg}| \tag{1}$$

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. In turn, it can get more balanced workload and better running performance. Unfortunately, this optimization problem is proven NP-compete (see Theorem 4.1). Therefore, a linear approximate algorithm with performance guarantee (with approximate ratio under 1.8) is proposed. All of these will be studied in Section 4. The effectiveness of this edge directing strategy is also confirmed in experiments over large real-world graphs (see Section 7).

3.2 Resource Balance Model

3.2.1 Binary search-based list intersection. As mentioned earlier, list intersection accounts for major cost in triangle counting. There are two typical classes of list intersection algorithms, *sort-merge* and *binary search*. Previous work [4, 15, 17] has pointed out that binary search has better performance in GPU triangle counting, since it has better thread independency, larger parallelism and less work complexity in parallel computing compared with the sort-merge method. Comparative experiments are also carried out to varify the advantages of binary search over sort-merge on two state-of-the-art implementations in Section 6.2. Thus, binary search is a more popular manner of list intersection in several state-of-the-art triangle GPU triangle counting algorithms, including Gunrock [33], TriCore [17], Hu's fine-grained implementation [15] and Fox's work [10, 11].

However, binary-search shows different resource preferences distinguished by length of target list. As shown in Figure 4, binary search requires three memory transactions on the long list while only one on the short list, although they both perform three searches. Accessing elements on long lists for a thread means more scattered memory access; and short lists, which exist widely in power-law graphs, usually mean good access pattern because data can be fetched within much less memory transactions.

When it comes to multiple threads, the problem becomes more severe, as shown in Figure 5, which assumes that a warp with

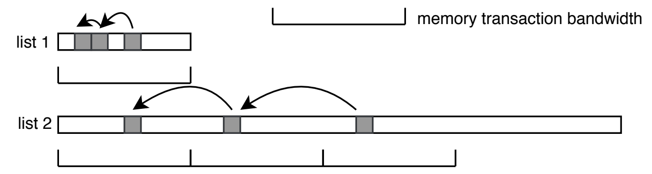

Figure 4: Memory access patterns of binary search using one thread

four threads is doing binary searches. Four threads conduct binary search over the same list L for different search keys. If L is short, due to the coalesced memory access, all data of the four threads can be loaded using a *single* memory transaction, as shown in dash rectangles of Figure 5(a). 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 (as shown in Figure 5(b)). Given a fixed time window, the workload is more dominated by memory access cost in long lists than short lists.

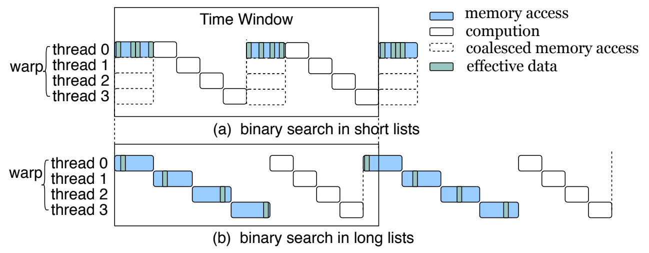

Figure 5: Memory access patterns of binary search using multiple threads

In conclusion, binary searches on long lists tend to be memory intensive, i.e. large memory bandwidth is desirable, while short lists are computing intensive, requiring more computational resources. Therefore, a better task assignment should make full use of the two resources within each hardware unit.

3.2.2 Model description. To balance the workloads, existing GPU triangle-counting algorithms always try to assign tasks with similar workload, but they often ignore the diversity of workloads. In fact, workloads should be balanced between computing cost and memory access cost.

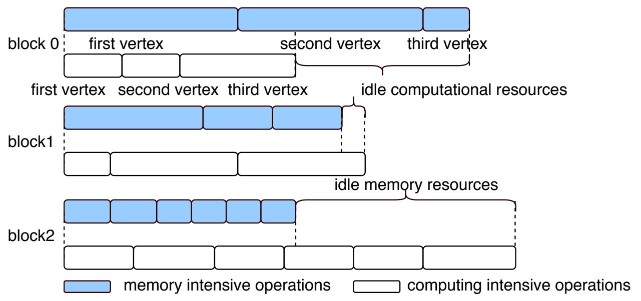

Figure 6: Resource balance model

On the other hand, modern GPUs have mature thread schedule mechanism to make best use of computing and memory resources. When the memory request issued by a warp is being processed, the warp will be scheduled off the SM until the data is prepared, and some other warps will be executed during the break to avoid waste of computational resources. For example, if two binary search warps on short lists and long lists (such as Figure 5(a) and (b)) is assigned to the same block, the first warp can "steal" computational resource of the SM when the second warp is scheduled off due to more memory transactions. Therefore, a clever task assignment is to group the tasks with different workload features (i.e., memory intensive or computing intensive) together and to dispatch them to one block. Consider the example in Figure 6, more computational resources are wasted in *block* 0 since the three vertices are all memory intensive tasks. *block* 2 in Figure 6 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. Having the above observation, the following problem is how to bring up a good block task assignment scheme. 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 always 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 $\widetilde{d}(v)$, thus, we define the functions $F_m(\widetilde{d}(v))$ and $F_c(\widetilde{d}(v))$ to denote memory intensity m and computing intensity c, respectively, of which the concrete form will be amplified later. 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:

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

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

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

For any bucket B_i, $|\lambda C_i - M_i|$ denotes the wasted resource size, i.e., the idle one in Figure 6. The parameter λ denotes the relative weight ratio between computational cost and memory access cost. The setting of parameters computing intensity function F_c, memory access intensity function F_m, and the ratio λ 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.

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 satisfying approximation ratio with linear preprocessing cost.

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 3.1.4) is a NP-complete problem.*

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

Given an undirected graph $G = (V, E)$, for each undirected edge $\forall(u, v) \in E$, we introduce two variables δ_{uv} and $\delta_{vu} \in \{0, 1\}$, where $\delta_{uv} = 1$ (or $\delta_{vu} = 1$) denotes 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 $\forall(u, v) \in E$, $\delta_{uv} + \delta_{vu} = 1$. Let $d(v)$ (resp. $\widetilde{d}(u)$) denote the number of edges (resp. out-going edges) connected to node u. It's easy to verify that

$$\widetilde{d}(u) = \sum_{v \in V} \delta_{uv}, \widetilde{d}_{avg} = \frac{1}{n} \sum_{u \in V} \widetilde{d}(u) = \frac{|E|}{|V|} \tag{4}$$

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 [1], which says $\forall u, v, w \in V$ and $u \neq v \neq w$, then $\delta_{uv} + \delta_{vw} + \delta_{wu} \leq 2$.

So the following 0-1 integer linear programming is equivalent to minimize Equation 1:

$$\begin{aligned} min_{\mathcal{P}} & \sum_{u \in V} | \sum_{v \in V} \delta_{uv} - \widetilde{d}_{avg}| \\ s.t. \; & \delta_{uv} \in \{0, 1\}, \forall u, v \in V, \\ & \delta_{uv} + \delta_{vw} + \delta_{wu} \leq 2, \forall u, v, w \in V, \\ & \delta_{uv} + \delta_{vu} = 1, \forall (u, v) \in E. \end{aligned} \tag{5}$$

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-complete problem. Thus, the theorem holds. □

4.2 Approximate Algorithm

Due to the NP-complete hardness, we propose A-direction strategy, a linear approximate algorithm to achieve a good tradeoff between efficiency and effectiveness.

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

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

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.

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

$$|\widetilde{d}(v_n) - \widetilde{d}_{avg}| \tag{6}$$

because $\widetilde{d}(v_n) \leq d(v_n) < \widetilde{d}_{avg}$, Equation 6 can be simplified as:

$$\widetilde{d}_{avg} - \widetilde{d}(v_n) \tag{7}$$

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

$$\begin{aligned} & (\widetilde{d}_{avg} - (\widetilde{d}(v_n) + 1)) + (\widetilde{d}_{avg} - \widetilde{d}(v_n')) \\ = \; & (\widetilde{d}_{avg} - \widetilde{d}(v_n)) + (\widetilde{d}_{avg} - (\widetilde{d}(v_n') + 1)) \end{aligned} \tag{8}$$

- edge (v_n,v_c). Assigning the edge as out-edge of v_n doesn't go worse than the opposite choice because:

$$\begin{aligned} & (\widetilde{d}_{avg} - (\widetilde{d}(v_n) + 1)) + |\widetilde{d}_{avg} - \widetilde{d}(v_c)| \\ = \; & (\widetilde{d}_{avg} - \widetilde{d}(v_n)) + |\widetilde{d}_{avg} - \widetilde{d}(v_c)| - 1 \\ \leq \; & (\widetilde{d}_{avg} - \widetilde{d}(v_n)) + |\widetilde{d}_{avg} - (\widetilde{d}(v_c) + 1)| \end{aligned} \tag{9}$$

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

Based on Lemma 4.1, we design a two-phase algorithm, named the *peeling* algorithm. The pseudocode is demonstrated in Algorithm 1. In the first phase, we collect every non-core vertex v such that $d(v) \leq \widetilde{d}_{avg}$ (Lines 5-6), and insert it into $Frontier$. For each vertex $v \in Frontier$, we check each neighbor nbr of v (Line 8). If the edge direction between v and nbr has not been defined, we will set edge direction $v \longrightarrow nbr$ (Line 9-16). The process guarantees that (1) for any edge between a core vertex and a non-core vertex, the edge direction must be from the non-core vertex to the core one (line 12-16); (2) for any edge between two non-core vertices, we define the edge direction from the vertex of smaller degree to the one of larger degree (line 9-11).

According to Lemma 4.1, it is easy to know the first phase is an exact algorithm. The critical problem is how to determine the edge direction between two core-vertices. For the second phase, a

Algorithm 1 A-direction Algorithm

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

```
1:  $Frontier \leftarrow \varnothing, nextFrontier \leftarrow \varnothing$
2:  $threshold \leftarrow \widetilde{d}_{avg}$
3:  while !all nodes peeled do
4:      for $v \in allNodes$ do
5:          if $d(v) \leq threshold$ then
6:              $Frontier.push\_back(v)$
7:      while Frontier not empty do
8:          for $v \in Frontier,\ nbr \in v.nbrList$ do
9:              if $nbr \in Frontier$ and $d(nbr) \geq d(v)$ then
10:                 $set\ edge: v \longrightarrow nbr$
11:                 $d(nbr) - -$
12:             if $nbr \notin Frontier$ and $(v, nbr)$ is unset then
13:                 $set\ edge: v \longrightarrow nbr$
14:                 $d(nbr) - -$
15:                 if $d(nbr) \leq threshold$ then
16:                     $nextFrontier.push\_back(nbr)$
17:         $Frontier \leftarrow nextFroniter$
18:         $nextFrontier.clear()$
19:     $threshold \leftarrow threashold * 2$
```

heuristic idea is inspired by Lemma 1. Specifically, we double the threshold in Line 19 and repeat the peeling procedure, by using an edge directing strategy analogous to the first phase. Intuitively, this method effectively reduces the number of vertices with large out-degrees. The above process is iterated until that no vertex is left. Obviously, the second phase is an approximate process; but we prove that the approximate ratio is small (less than 1.8) as follows. In other words, the whole Algorithm 1 is a performance-guarantee approximate algorithm.

In practice, the whole graph will be processed within constant times of doubling *threshold*, and the overall time complexity of the peeling algorithm is bounded by $O(|E|)$.

4.3 Approximation Ratio

Assume that the edge-directing scheme generated by Algorithm 1 is denoted by $\mathcal{P}_{Alg}$ and the optimal one is denoted as $\mathcal{P}_{Opt}$. We have the following theorem about the approximation ratio ρ.

THEOREM 4.2.

$$\rho = \frac{C(\mathcal{P}_{Alg})}{C(\mathcal{P}_{Opt})} \leq 1 + \frac{UB(C(\mathcal{P}_{Alg}) - C(\mathcal{P}_{Opt}))}{LB(C(\mathcal{P}_{Opt}))} \tag{10}$$

where UB and LB represent the lower bound and upper bound, respectively. Then we have

$$\begin{aligned} & LB(C(\mathcal{P}_{Opt})) \\ = & \begin{cases} \widetilde{d}_{avg}|V| - \sum\limits_{v \in V_n} d(v) - \sum\limits_{v \in V_c} \frac{d(v)}{2}, \\ \qquad if\ \frac{\sum_{v \in V_c} d(v)}{2} < \widetilde{d}_{avg}|V_c|. \\ \frac{1}{2}(\sum\limits_{v \in V_c} d(v) - 3 \sum\limits_{v \in V_n} d(v)) + \widetilde{d}_{avg}(|V_n| - |V_c|), \\ \qquad if\ \frac{\sum_{v \in V_c} d(v) - \sum_{v \in V_n} d(v)}{2} - \widetilde{d}_{avg}|V_c| \geq 0. \\ \sum\limits_{v \in V_n} (\widetilde{d}_{avg} - d(v)), \\ \qquad otherwise. \end{cases} \end{aligned}$$

And

$$\begin{aligned} & UB(C(\mathcal{P}_{Alg}) - C(\mathcal{P}_{Opt})) \\ \leq & \widetilde{d}_{avg} \cdot (|V_{\widetilde{d}_{avg}+1}| + |V_{\widetilde{d}_{avg}+2}| + \ldots + |V_{d_{peel}}|) \end{aligned}$$

where $|V_{\widetilde{d}_{avg}+1}|$ *denotes number of vertices whose degree is* $\widetilde{d}_{avg} + 1$, *and* d_{peel} *is the maximum degree when the whole graph is processed.*

PROOF. We will discuss the ρ in two parts.

LB(C($\mathcal{P}_{Opt}$)) This cost can be divided into two parts. The first part is contributed by vertices $v \in V_n$. The best case is that vertices in V_n don't have inter-edges, they are all connected to vertices $v \in V_c$. In that condition, $d(v) = \widetilde{d}(v)$ for any $v \in V_n$. For general cases, we denote the edge set containing edges between V_n and V_c as *CrossEdge*, and edge set for inter-edges of V_n as *LowLowEdge*. Denoting the number of *LowLowEdge* as N_{ll}, we have

$$\sum_{v \in V_n} d(v) = \sum_{v \in V_n} \widetilde{d}(v) + N_{ll}.$$

And the contribution of V_n to the total cost is:

$$\begin{aligned} & \sum_{v \in V_n} |\widetilde{d}(v) - \widetilde{d}_{avg}| \\ & = \sum_{v \in V_n} (\widetilde{d}_{avg} - \widetilde{d}(v)) \\ & = \sum_{v \in V_n} (\widetilde{d}_{avg} - d(v)) + N_{ll} \end{aligned} \tag{11}$$

In fact, the N_{ll} can be omitted in the scaling.

The second part of the cost $C(\mathcal{P}_{Opt})$ comes from vertices in V_c. The cost is

$$\begin{aligned} & \sum_{v \in V_c} |\widetilde{d}(v) - \widetilde{d}_{avg}| \\ \geq & \left| \frac{\sum_{v \in V_c} d(v) - |CrossEdge|}{2} - \widetilde{d}_{avg}|V_c| \right| \end{aligned} \tag{12}$$

So we have

$$
\begin{aligned}
&LB(C(\mathcal{P}_{Opt})) \\
&= \sum_{v \in V_n} (\widetilde{d}_{avg} - d(v)) + N_{ll} + \\
&\left| \frac{\sum\limits_{v \in V_c} d(v) - |CrossEdge|}{2} - \widetilde{d}_{avg}|V_c| \right|
\end{aligned} \tag{13}
$$

To give a more simplified results of this formular, we will discuss in the following three conditions.

(a) $\frac{\sum_{v \in V_c} d(v)}{2} < \widetilde{d}_{avg}|V_c|$. Then $\frac{\sum_{v \in V_c} d(v) - |CrossEdge|}{2} < \widetilde{d}_{avg}|V_c|$. Thus the lower bound of $C(\mathcal{P}_{Opt})$

$$
\begin{aligned}
LB(C(\mathcal{P}_{Opt})) &\geq \sum_{v \in V_n} (\widetilde{d}_{avg} - d(v)) + \widetilde{d}_{avg}|V_c| - \frac{\sum\limits_{v \in V_c} d(v)}{2} \\
&= \widetilde{d}_{avg}|V| - \sum_{v \in V_n} d(v) - \sum_{v \in V_c} \frac{d(v)}{2}.
\end{aligned} \tag{14}
$$

(b) $\frac{\sum_{v \in V_c} d(v) - \sum_{v \in V_n} d(v)}{2} - \widetilde{d}_{avg}|V_c| \geq 0$. Considering that $|CrossEdge| \leq \sum_{v \in V_n} d(v)$, $\frac{\sum_{v \in V_c} d(v) - |CrossEdge|}{2} - \widetilde{d}_{avg}|V_c| \geq 0$. And we have the observation that $\sum_{v \in V_c} d(v)$ is much larger than $\sum_{v \in V_n} d(v)$ in pow law graph model, when

$$
\begin{aligned}
&LB(C(\mathcal{P}_{Opt})) \\
&\geq \sum_{v \in V_n} (\widetilde{d}_{avg} - d(v)) + \frac{\sum\limits_{v \in V_c} d(v) - \sum\limits_{v \in V_n} d(v)}{2} - \widetilde{d}_{avg}|V_c| \\
&= \frac{1}{2}\Big(\sum_{v \in V_c} d(v) - 3 \sum_{v \in V_n} d(v)\Big) + \widetilde{d}_{avg}(|V_n| - |V_c|).
\end{aligned} \tag{15}
$$

(c) Otherwise. We scale the $LB(C(\mathcal{P}_{Opt}))$ to $\sum_{v \in V_n} (\widetilde{d}_{avg} - d(v))$.

UB(C($\mathcal{P}_{Alg}$)-C($\mathcal{P}_{Opt}$)) Firstly, according to the defination of cost function, we can have the conclusion that

$$
UB(C(\mathcal{P}_{Alg}) - C(\mathcal{P}_{Opt})) \leq \sum_{v \in V_c} |\widetilde{d}(v) - \widetilde{d}_{avg}|. \tag{16}
$$

For the above equation, we only need to consider vertices satisfying $d(v) > \widetilde{d}_{avg}$ with all edges in *CrossEdge*, because other vertices obtain optimal results using peeling strategy. And if a vertex v in V_c satisfies $\widetilde{d}(v) < \widetilde{d}_{avg}$, there will be an edge which adds 2 to the final cost. Otherwise, all edges all add 1 to the cost. For the worse case, optimal solution doesn't have edges adding 2 to the cost. Then the number of edges adding 2 to cost generated by our algorithm is just $UB(C(\mathcal{P}_{Alg}) - C(\mathcal{P}_{Opt}))$. For those vertices with $d(v) = \widetilde{d}_{avg} + 1$, we can direct all its $\widetilde{d}_{avg} + 1$ edges into its in-edge, getting $\widetilde{d}_{avg}$ extra cost; Then for those vertices with $d(v) = \widetilde{d}_{avg} + 2$... until at most $\sum_{v \in V_c} d(v)/2$ edges are all used up, at which time we are processing vertices with degree d_{peel}. So

$$
\begin{aligned}
&UB(C(\mathcal{P}_{Alg}) - C(\mathcal{P}_{Opt})) \\
&\leq \widetilde{d}_{avg} \cdot (|V_{\widetilde{d}_{avg}+1}| + |V_{\widetilde{d}_{avg}+2}| + \ldots + |V_{d_{peel}}|)
\end{aligned} \tag{17}
$$

□

Due to the complex analytic expression in Theorem 4.2, we present some intuitions about ρ. Since the degree distributions of most real-world graphs follow power law, we assume that the number of vertices of degree d is in reverse proportion to d^{γ}:

$$
Pr[d(v) = d] \propto 1/d^{\gamma}. \tag{18}
$$

Specifically, we use the configuration model (e.g., ACL model [1]) as an example, and generate a sequence of graphs varying the edge density. Then, we compute ρ using Theorem 4.2 for these graphs and plot the relation between ρ and d_{avg} in Figure 7. It turns out that ρ is less than 1.8 for graphs of arbitrary density.

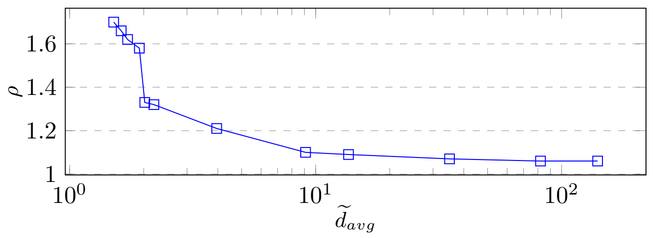

Figure 7: Approximate ratio under power law graphs

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

Table 3: ρ in several real-world graphs

datasets	$\widetilde{d}_{avg}$	ρ
email-Euall	2.85	1.31
gowalla	10.15	1.53
cit-patents	2.83	1.63
com-lj	8.5	1.46
kron-log21	1	1.16

5 GRAPH ORDERING

A block gets a group of continuous vertices as work set, so we can reorder all vertices to assure vertex's resource preferences in this group make up for each other. Based on the resource balance model in Section 3, we firstly prove the NP-hardness of the corresponding optimization problem, then propose an approximate 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 a NP-complete 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 $C_1 \leq C_{max}$, $C_2 \leq C_{max}$.

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 get the conclusion the DP is NP-complete, so is the original problem. □

5.2 Problem Solution

Because of hardness of the problem, we use a heuristic algorithm to deal with the graph reordering problem. Specifically, the algorithm employs a greedy strategy. The pseudocode is demonstrated in Algorithm 2. Note that the concrete form of the function F_c, F_m and the parameter λ are the prerequisites for the algorithm. We will discuss the parameter estimation later in Section 5.3.

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: $V_{mem} \leftarrow memory\ dominated\ vertices$
4: $V_{comp} \leftarrow computing\ dominated\ vertices$
5: for all $v \in V_{mem}$ do
6:     $b \leftarrow B.pop\_queue(\,)$
7:     $b.vertices.push\_back(v)$
8:     $b.mem\_sup+ = (F_m(\widetilde{d}(v)) - \lambda F_c(\widetilde{d}(v)))$
9:     $B.insert(b)$
10: $B.make\_max\_queue(\,)$
11: for all $v \in V_{comp}$ do
12:     $b \leftarrow B.pop\_queue(\,)$
13:     $b.vertices.push\_back(v)$
14:     $b.mem\_sup + = (F_m(\widetilde{d}(v)) - \lambda F_c(\widetilde{d}(v)))$
15:     $B.insert(b)$
16: $reorderIdx \leftarrow 0$
17: for all $b \in B$ do
18:     for all $v \in b.vertices$ do
19:         $v.id \leftarrow reorderIdx$
20:         $reorderIdx \leftarrow reorderIdx + 1$
```

The aim of A-order algorithm is to dispatch vertices to buckets so that the demands of computational resources and memory resources are balanced in each bucket. We say a vertex is memory-dominated (resp. computing-dominated), if it needs more memory (resp. computational) resources. Generally speaking, if a vertex is memory-dominated, we put it into the bucket with the least demand of memory resources. Similarly, for a computing-dominated vertex, it is added to the bucket of which the memory access cost is larger than the computational cost. Firstly, all buckets are initialized with their *mem_sup* set to 0 (line 1). For each vertex v, we refer to $F_m(\widetilde{d}(v)) - \lambda F_c(\widetilde{d}(v))$ as memory superiority. We use variable *mem_sup* denotes 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 priority of the bucket is then updated accordingly. And then all buckets are made into a maximum priority queue according to *mem_sup* (line 10). All the computing-dominated vertices are dispatched to buckets in a way that the demand of two resources are balanced as much as possible. Finally we reorder the whole graph, making sure that vertices in the same bucket have consecutive ids (line 16-20).

It can be easily proved that complexity of the algorithm is bounded by $O(|V|log|B|)$, Since |B|«|V|, the algorithm achieves near-linear complexity.

5.3 Parameter Determination

As discussed above, to solve the optimization problem, we need firstly figure out all the variables: computing intensity c (or F_c), memory intensity m (or F_m) and parameter λ. Note that the values of these parameters vary for different triangle counting algorithms as well as different reordering units (i.e., vertices or edges). However, they can be estimated in similar method. In the following text, we will describe parameter determination by taking Hu's implementation [15] as an example, which uses vertices as reordering units. Since these parameters only depends on algorithms, we can estimate them once and apply it for the following triangle counting tasks (i.e., other datasets).

Functions $\mathbf{F_m}$ and $\mathbf{F_c}$ Shared memory bandwidth BW is a straightforward measurement of memory access intensity. We can obtain the relation between BW and the out-degree $\widetilde{d}(v)$ by running $nvprof$, as shown in Figure 8. The memory intensity is positive correlated with BW, so we assume that

$$F_m(\widetilde{d}(v)) = BW(\widetilde{d}(v))^{\alpha} (\alpha > 0) \tag{19}$$

However, there are no direct evidence of F_c. As we have analysed, binary search on short lists needs more computational resources than that on long lists because coalesced memory access saves memory transactions. Consider Hu's work [15], where the endpoint u is iterated over the vertices in graph, and other two endpoints v and w are iterated over the corresponding adjacency lists. Technically, it takes more times to find a u-v-w triple for a short adjacency list than a long one. Therefore, we conclude that the F_c of a vertex v is negative correlated with length of adjacency list ($\widetilde{d}(v)$), so we assume that

$$F_c(\widetilde{d}(v)) = 1/\widetilde{d}(v)^{\beta} (\beta > 0) \tag{20}$$

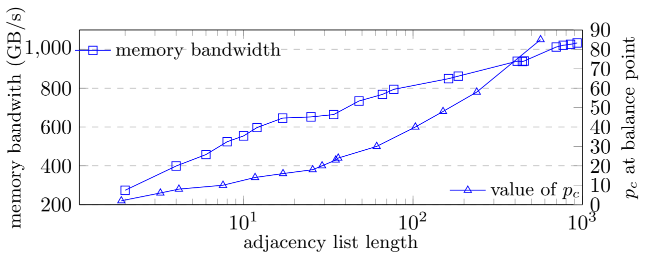

Figure 8: Left vertical axis: the relation between shared memory bandwidth and adjacency list length; right vertical axis: the relation between p_c and adjacency list length.

Parameter λ 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 computing ability. But we can't get exact maximum computing ability concerning a specific computational task. Hence, we design an experiment to estimate λ as follows.

We have known that vertices with long adjacency lists need more memory resources than computing resources. Therefore, if we add a little more computational tasks to them, the overall executing time will remain unchanged. But if we keep forcing more computing pressure, i.e. executing codes for getting u, v, w or doing searches in prefetched memory for extra (denoted as p_c) times, it will finally reach the equilibrium where both the computational and the memory resources are fully utilized. We call this situation *balance point*. At *balance point*, we have:

$$m = \lambda(p_c \times c) \tag{21}$$

By collecting vertices of same adjacency list length together and conducting the above experiment, we can get p_c in different degree vertices. We carried out multiple experiments on several representative datasets to get more accurate p_c. Specifically, in our experiments, we say the overall execution time remains unchanged if the variation between different executions is below 5%. Figure 8 shows the relation between the length of adjacency list and p_c. We can see that p_c grows as the length of adjacency list grows. Note that we only demonstrate the case of memory-dominated vertices, and it is vice versa for computing-dominated vertices.

According to our analysis, Equation 21 should be satisfied at *balance point*. And it is a direct proportional function. We have known the representation of the two variables c (or F_c) and m (or F_m), and after trying several options, we find that if

$$F_c(\widetilde{d}(v)) = \sqrt{1/\widetilde{d}(v)},\ F_m(\widetilde{d}(v)) = \sqrt{BW(\widetilde{d}(v))} \tag{22}$$

, the functional image of m related to $c \times p_c$ can be well fitted by a linear function, as shown in Figure 9. In our experiment, the λ is 9.682.

With these functions and parameters, our A-order algorithm can be successfully carried out. Similar parameter determination process applies to other triangle counting works, which we will talk about in Section 6.

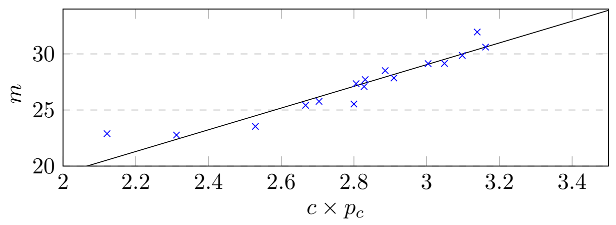

Figure 9: Fitting function

6 EXPERIMENTS

In this section, we evaluate our method on both synthetic and real graph datasets.

6.1 Experimental Settings

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

Datasets : Both real-world and synthetic datasets are used. We obtain real-world datasets from Stanford Network Analysis Project (SNAP)[2] and HPEC graph challenge[3], while synthetic datasets are obtained by Kronecker generator[4]. Table 4 shows statistics of all real-world datasets we use in the following experiments.

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 [33]. It is a mature GPU graph processing library, which has been included into NVIDIA official acceleration library.
2. TriCore [17]. This work uses a warp to deal with an edge, which adapts to SIMT features in GPU.
3. Fox's work [10, 11]. These two papers use similar methods, which win graph challenge 2018's Innovation Awards and Finalists, respectively.
4. Bission's work [7]. It emphasizes match computational resource with workload, which has great influence on later works.
5. Hu's work [15]. It proposes novel fine-grained workload distribution manner that is suitable for GPU architecture.

As baselines, we consider the degree-based ("D-direction" for short) for edge directing and original vertex ID-based ("Original" 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 approach 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 Bission's and Hu's work adopt the explicit intra-block synchronization. Thus, we evaluate A-direction

[2]http://snap.stanford.edu/data/
[3]https://graphchallenge.mit.edu/data-sets
[4]https://github.com/graph500/graph500

in these two methods (Section 6.3). Except for Bission’s work[5], 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 6.4). Since Hu’s work adopts both intra-block synchronize model and binary search-based list intersection, we measure the effectiveness of the two preprocessings (A-direction and A-order) together in Section 6.5.

Table 4: Datasets Infos

dataset	nodes	edges	triangles
email-Eucore	934	16K	105,461
email-Euall	265K	729K	267,313
gowalla	197K	2M	2,273,138
road_central	14M	17M	228,918
soc-pokec	1.5M	22M	32,557,458
soc-LJ	5M	69M	285,730,264
com-orkut	3M	117M	627,584,181
com-LJ	4M	34M	177,820,130
cit-Patent	6M	17M	7,515,023
kron-log18	25M	25M	281,814,846
kron-log21	201M	201M	1,765,053,740
twitter_rv	62M	1.5B	34,824,916,864
wiki-topcats	2M	19M	17,864,012
s24.kron.edgelist	17M	268M	10,286,638,314
s26.kron.edgelist	67M	1.1B	49,167,172,995

6.2 Set Intersection Strategies

We choose five state-of-art implementations in our work to verify our analytic model, and all of them use binary search as set intersection strategy, except that Gunrock uses both binary search and sort-merge methods. Although there are also some works[12, 23] who adopt sort-merge method, we believe binary search is a better option because of better independency and larger parallelism, which has been stated in some literature [4, 15, 17].

To make this conclusion more credible, we compare those two strategies by separately using them in one same implementation. Gunrock and TriCore are two significant triangle counting implementations, thus we use them in the comparison. Figure 10 shows results of two implementations adopting two strategies alone, in which “bs” is short for “binary search” and “sm” for “sort-merge”. We can conclude binary search performs better than sort-merge as set intersection strategy in triangle counting on GPU.

6.3 Edge-directing Strategy

In this experiment, we compare analytic model-based edge directing scheme with the degree-based and ID-based edge directing methods. Note that we adopt the baseline vertex ordering (i.e., Original) for all implementations.

Firstly we want to confirm the effect of our edge direction method to our cost function (Euqation 1). Figure 11 shows the overall cost decline in percentage of our method compared with the other two methods on four datasets, where the cost is computed by Equation 1. It’s easy to conclude that vertices with large degree have especially

[5]Bission’s work uses bitmap for list intersection

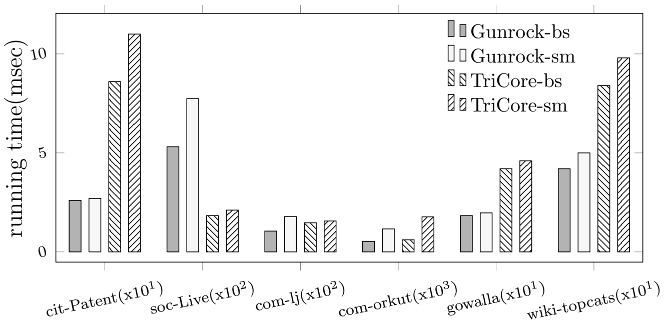

Figure 10: Binary search and sort merge strategy on Gunrock and TriCore

great influence on the performance, so we pay extra attention to those vertices. *Degree threshold* as k in Figure 11 means that we only add up cost of vertices whose degree is larger than $k * \widetilde{d}_{avg}$. For example, our method achieve about 10% cost decline for all four datasets ($k \geq 4$) in Figure 11(a), which means better load balance.

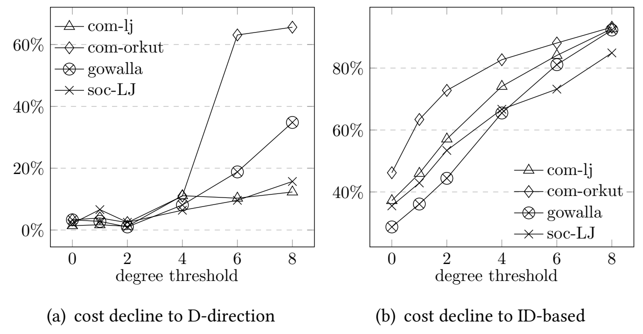

Figure 11: Cost decline of our methods compared with other edge direction strategies

Evaluation of A-direction on Hu’s algorithm. The running times of three edge direction methods, i.e., id-based, degree-based (“D-direction”) and our method (“A-direction”) on several datasets are demonstrated by the “bars” of Figure 12. 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 our method and degree-based approach (D-direction) are significantly faster than the id-based scheme. To compare our method with D-direction, we also show the speedup ratio of our method with D-direction in both GPU kernel running time and total time (including preprocessing time) by separate lines in Figure 12. Our strategy outperforms D-direction on all datasets; we achieve 9.4% ∼ 42.4% improvement on kernel time and 6.3% ∼ 34.5% improvement on total time.

Evaluation of A-direction on Bission’s algorithm. The experiment results are shown in Figure 13. We also compare A-direction strategy with ID-based method and degree-based method. In this implementation, ID-based method works significantly worse than the degree-based methods, while our method still have 2.6% ∼ 54.9% speedup than degree-based one. Kernel time in this implementation far surpasses the preprocess time, making two lines in Figure 13, speedup of kernel and total time, very close to each other.

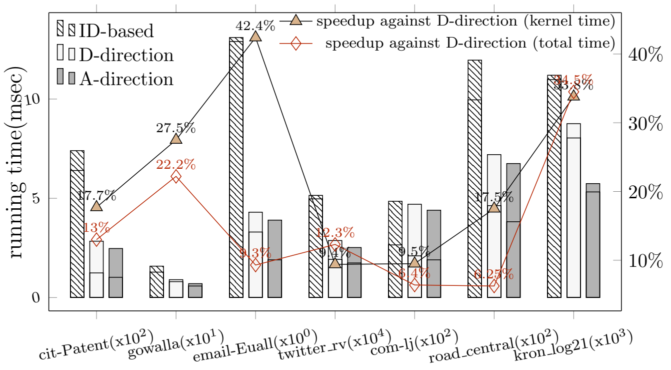

Figure 12: Running time of different edge direction methods on Hu's work

In conclusion, our edge direction method ("A-direction") effectively reduces the cost raised in Equation 1, and achieves significant performance improvements on both kernel running time and total time compared with degree-based edge direction method, which is widely used in state-of-the-art implementations.

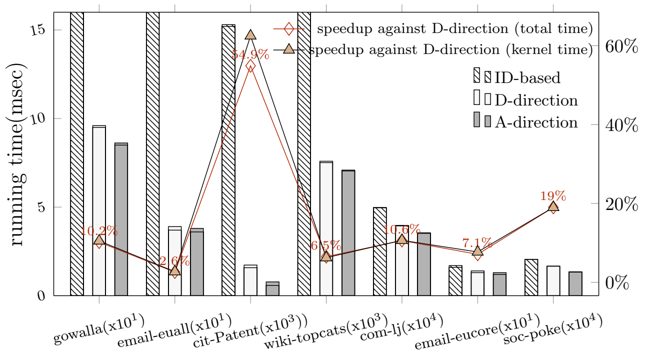

Figure 13: Running time of different edge direction methods on Bission's work

6.4 Graph Ordering Strategy

The vertex ordering strategy aims to find a good ordering to address the problem of workload diversity. To isolate the effect of our vertex-reordering strategy, we adopt the default degree-based edge direction in 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 [26], BFS-R [8], SlashBurn [19] and GRO [13].

Evaluation of A-order on Hu's algorithm and TriCore. We can use vertices as reorder units for both TriCore [17] and Hu's work [15], and they both adopt binary search for list intersection, thus, they are both suitable to our analytic model in Section 3.2.

So far we have three vertex reordering strategies: original vertex order (Original), 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 [26], BFS-R [8], SlashBurn [19] and GRO [13]. DFS reorders vertices according to depth-first traversal; BFS-R firstly performs BFS to find a vertex with largest depth, then performs BFS from it until half of all vertices are visited. This method recursively partitions the graph vertices into visited and unvisited parts to build a separator tree, finally all nodes are reordered according to their appearance order in the leaves of separator tree; SlashBurn groups vertices in the same adjacency list and gives them continuous order; GRO propose the notion of compactness scores to make neighbor vertices consecutive in ids, and proposes a greedy algorithm to minimize it.

Tables 5 and 6 shows 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 vertex ordering. As we have analyzed, the degree-based ordering strategy is significantly 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 3.5% ∼ 57.1% speedup of kernel time than original order, which is better than all the other strategies, and 2.2% ∼ 41.3% speedup even including preprocessing time. Our total time for two datasets(email-Euall and gowalla) is worse than the original order, because the kernel time itself is short in those two datasets, leaving few space for improvement. Therefore, the kernel time earning can not make up the preprocessing time. Moreover, A-order outperforms original order significantly on big datasets (such as kron-logn21 and com-lj). 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 9.8% ∼ 50.0% speed up of kernel time, and 1.2% ∼ 47.5% speed up of total time.

Evaluation of A-order on Gunrock. Gunrock [33] 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 based [4, 15, 17] 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). For the sort-merge GPU list intersection implementation in Gunrock, it also employs the same resource preferences with binary search one (computing intensive on short lists and memory intensive on long lists).Parallelized sort-merge on GPU firstly divides one list into multiple segments before all threads start working on each segment. Then each segment performs binary searches of their start and end vertices in the other list to obtain the corresponding segment, after which threads use sort-merge method for intersection of two segments. Intersection of two long lists needs more binary searches than that of short lists, because list is divided into more segments. Obviously, binary search is more memory intensive than sort-merge. Therefore, sort-merge based set intersection is memory intensive on long lists, while computing intensive on short lists.

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
soc_LJ	178	236	151	1647	146	5011	158	1008	126	148126	127	28.7%	142	20.2%
cit-patents	130	4900	122	786	120	2043	118	3234	123	10923	104	20.0%	124	4.6%
com-lj	262	210	246	1228	230	1679	231	3877	264	59264	194	26.0%	206	21.4%
com-orkut	776	729	770	3597	790	4093	760	4636	763	860763	749	3.5%	759	2.2%
email-Enron	6.3	21	4.8	14.9	3	17.2	5	18.8	6.5	271.5	2.7	57.1%	3.7	41.3%
email-Euall	2.8	22	3	21.8	3.5	58.2	4	65	6	1204	2.3	17.9%	3.3	-17.9%
gowalla	8.8	26.1	13	55	8.1	109.7	10	1708	12	3551	6.6	25.0%	9.6	-9.1%
wiki-topcat	101	98	120	904	109	1254	112	173	129	159129	87	13.9%	96	5.0%
kron-logn18	372	452	372	1361	373	960	367	6788	369	424369	330	11.3%	340	8.6%
kron-logn21	8042	9611	5250	16450	5248	13073	5107	126107	5270	$1.77\text{x}10^7$	5020	37.6%	5073	36.9%

Table 5: Different reorder strategies on Hu's fine-grained implementation

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
soc_LJ	184	380	199	1695	198	5063	195	1045	209	148209	163	11.4%	178	3.3%
cit-patents	86	103	90	754	99	2022	95	3211	93	10893	65	24.4%	85	1.2%
com-lj	147	271	142	1107	162	1611	170	3816	163	59163	132	10.2%	144	2.0%
com-orkut	610	2729	570	3397	600	3903	564	4440	623	860623	550	9.8%	560	8.2%
email-Enron	34	34	34	44	32	46	34	47	34	299	21	38.2%	22	35.3%
email-Euall	40	42	39	57	38	93	38	99	38	1236	20	50.0%	21	47.5%
gowalla	42.1	44.9	40	82	40	141.6	40	1738	41	3580	23	45.4%	26	38.2%
wiki-topcat	98	108	91	875	97	1242	89	150	88	159088	78	20.4%	87	11.2%
kron-logn18	380	1514	330	1319	340	927	300	6721	330	424330	340	10.5%	350	7.9%
kron-logn21	5100	37640	4060	15260	3990	11815	3650	124650	3850	$1.77\text{x}10^7$	4500	11.8%	4553	10.7%

Table 6: Different reorder strategies on TriCore implementation

Comparing with the original order, we find that our analytic model-based vertex ordering (A-order) can achieves 6.0% ~ 82.4% performance improvement in total time, and even better in kernel time (see Figure 14). 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.

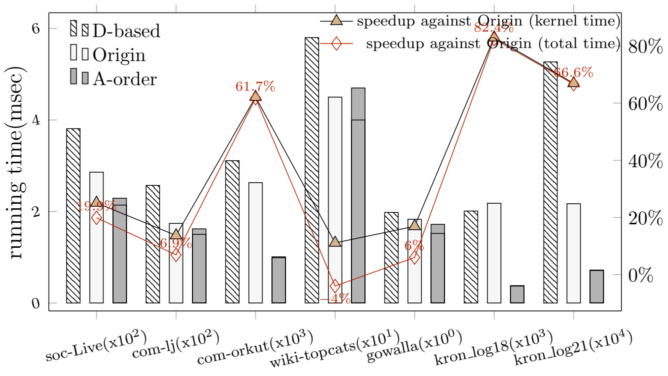

Figure 14: Vertex ordering results of Gunrock

Evaluation of A-order on Fox's algorithm. Edge is reorder unit of Fox's algorithm [10], 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 [15], except that we consider resource usage tense of edges instead of vertices. Fig 15 shows results of our method and original edge distribution in several datasets. Finally we achieve 2% ~ 26.2% performance improvement in total time.

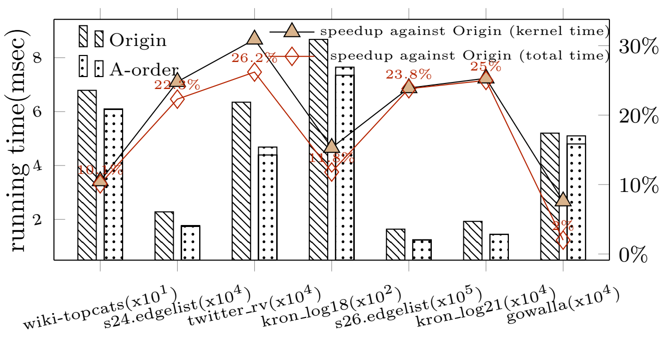

Figure 15: Vertex ordering results of Fox's work

6.5 Evaluating the Combined Approach

Since some algorithms include both intra-block synchronization and binary search-based list intersection, such as Hu's work [15], we combine both the analytic model-based edge direction (A-direction) and vertex ordering (A-order), and the evaluation result is shown in Figure 16. The lines show that the 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 7.6% on average comparing with A-direction only, and 13.6% on average comparing with A-order. This further confirms the effectiveness of our analytic model-based preprocessing.

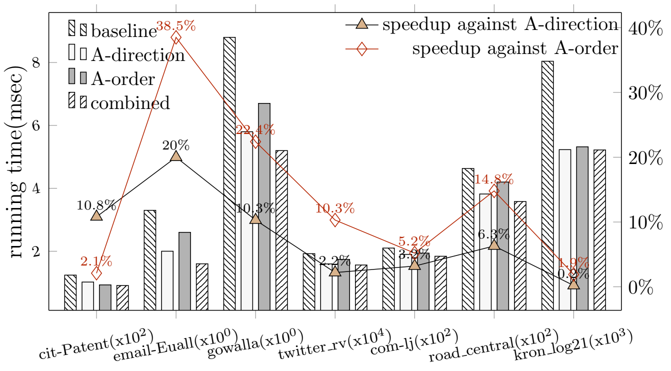

Figure 16: Combining two models on Hu's work

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 one 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 improve the performance of these algorithms significantly over both large real-world and synthetic graph datasets.

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] 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

[6] 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

[7] 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.

[8] 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

[9] 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.

[10] 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.

[11] 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.

[12] Oded Green, Pavan Yalamanchili, and Lluís-Miquel Munguía. 2014. Fast triangle counting on the GPU. In *Proceedings of the Fourth Workshop on Irregular Applications - Architectures and Algorithms, IA3 2014, New Orleans, Louisiana, USA, November 16-21, 2014*, Antonino Tumeo, John Feo, and Oreste Villa (Eds.). IEEE, 1–8. https://doi.org/10.1109/IA3.2014.7

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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

[19] 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

[20] 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.

[21] 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.

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

[23] Carl Pearson, Mohammad Almasri, Omer Anjum, Vikram S. Mailthody, Zaid Qureshi, Rakesh Nagi, Jinjun Xiong, and Wen-Mei W. Hwu. 2019. Update on Triangle Counting on GPU. In *2019 IEEE High Performance Extreme Computing Conference, HPEC 2019, Waltham, MA, USA, September 24-26, 2019*. IEEE, 1–7. https://doi.org/10.1109/HPEC.2019.8916547

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

[25] 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

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

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

[28] 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.

[29] 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.

[30] Leslie G Valiant. 1990. A bridging model for parallel computation. *Commun. ACM* 33, 8 (1990), 103–111.

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

[32] 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.

[33] 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.

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

[35] 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

[36] 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.

[37] 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