Accelerating Triangle Counting on GPU via Analytic Models

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

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

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

ABSTRACT

Triangle counting is an important algorithm for graph analysis, which has achieved great performance promotions on GPU in recent years. But there are still drawbacks for it because of the irregularity of graphs and the architecture of GPU. This paper focuses on the GPU underutilization caused by workload imbalance and diversities, which are ubiquitous in almost all state-of-art GPU implementations for triangle counting. To figure out how those two problems exactly effect the performance, we analyze recent implementations by abstracting two novel analytic models and building up cost functions. Then, due to the NP-complete hardness of the model optimization, we bring up approximate solutions by directing edge directions to balance workloads and reordering vertices to maximize the degree of parallelism within blocks. Finally, extensive experiments confirm that our method achieves great performance promotions for several state-of-art GPU triangle counting algorithms without changing their implementations and data structures.

1. INTRODUCTION

Graph is a broadly used representation of data, which is getting increasingly important because of its flexibility. Triangle counting [6, 10, 18, 19], computing the total number of triangles in a graph, lays foundation for many graph computing tasks, such as k-truss [24], clustering coefficient [27], and link recommendation [22].

With the scale of graph getting larger continuously, traditional serial graph algorithms fail to satisfy the performance requirement. Thus, lots of existing works resort to new hardwares with great parallel processing ability (such as GPU) to address efficient graph computing tasks, such as BFS [16], SSSP [8], subgraph isomorphism [21]. Triangle counting works on GPU also achieve great acceleration, specifically, 9∼260x faster compared with serial implementations on CPU [25] and many of algorithms [9, 13, 14, 18] have been proposed in the literature. In this paper, we do not intend to propose a new GPU-based triangle counting algorithm. Instead, we aim to optimize existing algorithms by preprocessing graph data. Our work is inspired by an interesting observation: for the same data graph and the same GPU triangle-counting algorithm, different graph preprocessings result in significantly different performance (see Table 2). Therefore, we concentrate ourselves on a light-weight data preprocessing strategy that can benefit a group of relevant GPU-based 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 redundancy computation [13,18]. Thus, each edge's direction should be defined before running triangle-counting algorithms. Two popular strategies are id-based and degree-based. The former defines a directed edge from small vertex id to big vertex id, and the latter directs an edge from small-degree vertex to high-degree vertex. Experimentally, the later often leads to faster 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 a better edge-directing strategy. Figure 13 shows that our analytic model-based edge-directing scheme leads to 9% to 42% improvement, comparing with degree-based one. Section 4 studies this problem.

Vertex-ordering. The basic programmable unit in GPU is a *block* that has many of parallel-processing threads. A batch of computing tasks are always grouped and assigned to a block. Therefore, tasks of the consecutive vertices are more likely to be assigned to the same block. In other words, vertex ordering strategy always determines the task assignment in GPU, and different assignments lead to different resource usage, as well as different running times. Figure 16 demonstrates that our proposed vertex-ordering accelerates the performance by 7% to 34%, comparing with the commonly used original vertex ordering. This problem is studied in Section 5.

The above observation tells us an important fact: different data preprocessings result in different triangle-counting performance. 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 mathematical analytic model 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 light-weight (such as linear) approximate algorithm? Apparently, the performance-guarantee

Proceedings of the VLDB Endowment, Vol. 12, No. xxx
ISSN 2150-8097.
DOI: https://doi.org/10.14778/xxxxxxx.xxxxxxx

approximate algorithm is enticing. We will study these problems in this work.

Actually, our analytic model design originates from the performance analysis of workloads. We consider two aspects of workloads: *imbalance* and *diversity*, which cause above-mentioned two types of performance differences, respectively.

Workload Imbalance. As stated in many previous works [7, 15, 16, 20], when graph algorithms are transplanted to GPU, thread divergence should be avoided as much as possible. Warp is a basic running unit of GPU, and threads inside run in SIMT (single instruction multiple threads) manner. Therefore, branches and workload imbalance, i.e. thread divergence, will cause massive stalls of threads, thus leading to severe performance decline. Workload imbalance of GPU graph algorithms often stems from skew vertex degree distribution in many real-world graphs. Fortunately, in GPU-based triangle counting, edge-directing strategy provides an opportunity to change the 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.

Workload Diversity. There are two kinds of workloads for a given task: computing cost and memory-access cost. The diversity of workloads is few studied in GPU graph algorithms. In triangle counting, the list intersection step costs 94.3% of the running time [11]. Yet we find that the same list intersection algorithm plays different kinds of workload. Long list intersection is memory-intensive; but the short list one is more likely to be computing-intensive. More details and reasons will be discussed 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 can only be dispatched to one SM. A good block task assignment is to group a batch of tasks with different resource prefers (i.e., computing-intensive and memory-intensive workloads) together, which are assigned to one SM. In this way, the thread schedule mechanism can make the full use of two kinds of resource in an SM for parallel processing. Otherwise, if tasks with similar workloads (i.e., the same resource prefer) are assigned to a block, more threads have to be suspended due to one resource conflict, but the other resource will be idle and wasted in SM. In this paper, we propose to implement a better task assignment through reordering vertices of a given data graph. An analytic model for vertex-ordering is proposed for this.

According to our proposed analytic models, we aim to find optimal edge directing to balance workloads, and find the optimal vertex ordering to maximize the degree of parallelism. Unfortunately, both problems are NP-complete (see 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 bring up a light-weight (linear complexity) approximate algorithms to find good solutions for edge directing and vertex ordering. Finally, our method can accelerate GPU kernel running time by up to 51% in multiple implementations, and the total time (including data preprocessing step) is speeded up by over 25%.

Generally, we made the following contributions:

- We present two run-time models abstracted from several state-of-art triangle counting implementations, considering GPU architecture features. And these two models measures 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 based on our analytic model. Due to high hardness of this problem, we propose a lightweight (linear) approximate algorithm with performance guarantee.

- Considering workload diversity, we reorder vertices to implement a better task assignment to maximize the degree of parallelism. Specifically, we crystallize vertex reordering as a model optimization problem.

- We conduct massive experiments to verify effectiveness of our analytic models and our data preprocessing methods. Extensive experiments confirm that our approach can speed up state-of-art triangle-counting algorithms significantly.

The remainder of the paper is organized as follows. Section 2 introduces GPU architechture and some state-of-art triangle counting implementations, while Section 3 describes two general run-time models abstracted from state-of-art implementations. In Section 4 and Section 5 we present two optimization solutions, edge direction and graph reordering, aiming at two run-time models respectively. Section 6 discusses related work of modeling for graph algorithms and triangle counting on GPU, and Section 7 shows our experiments results. Finally Section 8 concludes our works.

2. PRELIMINARIES

2.1 GPU Architecture

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

Software. CUDA 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 executing 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 as balanced as possible, because imbalance workload among threads will lead to severe thread divergence.

Hardware. Each stream multiprocessor (SM) has many cores, who can run in parallel, and each SM could run hundreds of threads and GPU has tens of SMs in general, that's where the computing power comes from.

Just like CPU, GPU has its own main memory and cache. Global memory is the slowest memory, but can be accessed by all threads. Each SM has fast-access memory: shared memory, which is programmable with limited space, and carefully designed using strategies will lead to great performance. We can see that an SM is a rather isolated hareware unit, in that it has not only computing cores but also private fast-access memory. In fact, it can been seen as match of *block*, because a block will be assigned to only one SM in run-time.

Coalesced memory access on GPU means that neighbor threads are accessing neighbor and continuous memory positions, and thread indexes are aligned with memory address, in which case a warp can fetch all the required data of threads inside by one memory transaction, which greatly improves the performance. Otherwise, more memory transaction will be needed. The rule applies to both shared memory and global memory: coalesced memory access leads to better performance.

2.2 Typical Triangle Counting Implementations

Based on the work distribution units, there are three different kinds of methods: vertex-based, edge-based and wedge-based, as shown in Figure 2. For each category, we introduce one typical algorithm in the following. We will abstract two analytic models among these existing methods. For the convenience of demonstration, a running example graph is given in Figure 1.

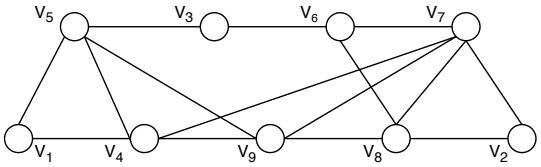


Figure 1: A graph example

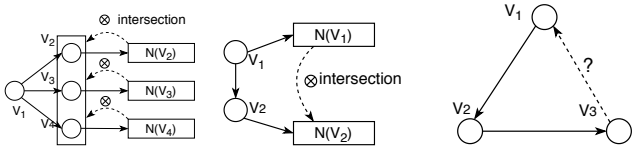


Figure 2: Three different work units

Vertex-based Method

We introduce block-level bitmap implementation [6] as a typical solution in this category. Before running example, we need to transfer an undirected graph G into a directed graph by defining edge directions. In this method, edges are directed form vertices of big id to vertices with small id. As for task distribution, a block is responsible for triangle counting tasks of a whole vertex, making it a typical vertex-based work.

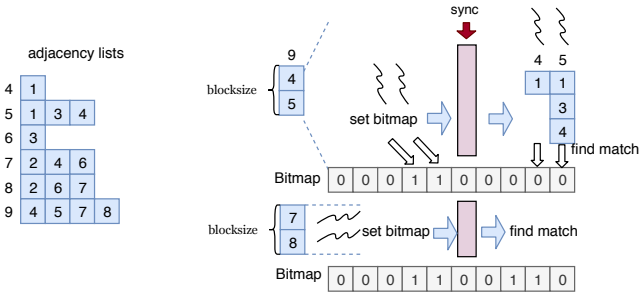


Figure 3: Vertex-based methods with bitmaps

In Figure 3, let us consider 9's adjacency list, and assume that a block has only two threads. A thread is in charge of one vertex in the adjacency list, doing intersection of this vertex's adjacency list with vertex 9's. This work uses bitmaps for fast look up of target lists, so elements can check their matches in one look up during list intersection. While doing this, threads firstly set the their current vertex 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 could move on and use the bitmap to find matches for every elements in their adjacency lists. Threads inside the block will move to next group of vertex 9's neighbors and continue doing similar works until all neighbors of vertex 9 are processed.

In Figure 3 neighbor lists of vertex 4 and 5 have different lengths, while they get both one thread to find matches of all elements in it. Lengths of adjacency lists may vary a lot, leading to severe work load imbalance among threads.

Edge-based Method

In edge-based methods, we introduce logarithmic radix binning implementation [9, 10]. The typical step in edge-based methods is: for each edge, we perform the intersection between its two end-points' adjacency lists.

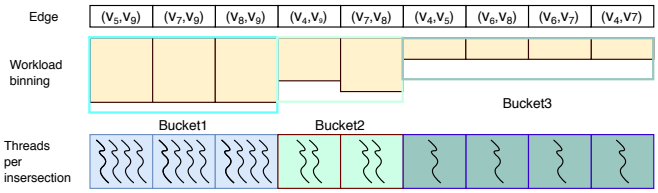


Figure 4: Logarithmic radix binning

Figure 4 shows how it works. Firstly, workload of all edges are estimated, then edges will be rearranged and those with similar workloads will be put into a same bucket. The step among buckets is exponent of 2 in reality, so the implementation is called "logarithmic radix binning". Finally, threads are assigned for them and edges in buckets with high workload will get more threads. In the Figure 4, edges in the first bucket will get 4 threads each, edges in the second bucket get 2 threads, and 1 threads for the other edges. Tasks of different edges are independent, and threads for a same edge will work collaboratively to finish their tasks.

This work achieves good load balance by assigning threads to edges according to their workloads. However, the same binary-search list intersection algorithm shows different resource prefers distinguished by target list length. Accessing elements on long lists for multiple threads means more scattered memory access; and short lists, which exist widely in power-law graphs, usually mean good access pattern because all data can be fetched within much less memory transactions. In other words, binary searches on short lists tend to be more computing intensive, requiring more computing resources; while, the long lists are likely to be memory-access intensive, large memory bandwidth is desirable. A better task assignment should make the full use of the two resources.

Wedge-based Method

A wedge refers to a three-vertex fragment u-v-w. In [13], a wedge is the basic task distribution unit and each thread just check if a wedge can form a triangle, as shown in Figure 2(c). They do binary searches in u's adjacency list to answer them. As many other works, this method uses CSR(compressed sparse row) to store the whole graph. In Figure 5 the first row (vertices in which denoted as u) and

the second row (v) are CSR. And the third row (w) contains adjacency lists of all vertices in the second row, which can be obtained by accessing the above two rows. Assuming there are four threads in total, Figure 5 shows how *thread* 0 gets its work. While running, a piece of the second row(v) will firstly be loaded into shared memory by a block for faster access. Similarly, the synchronization step is necessary after loading. Then all concerning binary searches will be carried out by threads inside the block. The whole graph is loaded into shared memory in pieces and processed repeatedly in this way.

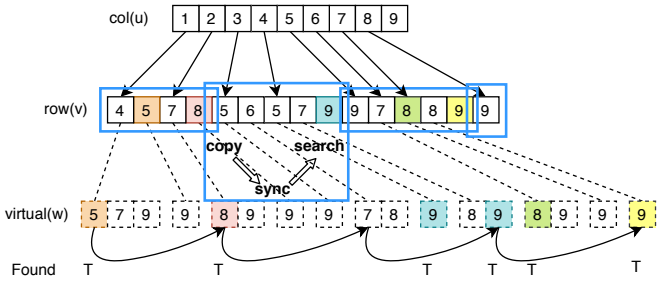


Figure 5: The CSR and w lists of given example in wedge-based method

How a thread get its own wedge (u, v and w) is a big issue. In fact, they are obtained by run-time computing. Every time a thread finishes its subtask, it will compute new u, v and w to get a new subtask. Algorithm 1 is pseudocode of getting all three variables.

Algorithm 1 Getting u, v and w

Input: graph CSR $col[\,]$, $row[\,]$ and uid, vid, $wpos$
Output: uid, vid, wpos

```
1: define adjLength[id] as col[id + 1] − col[id]
2: while wpos >= adjLength[vid] do
3:     wpos ← wpos − adjLength[vid]
4:     vpos + +
5:     vid ← row[vpos]
6:     while vpos >= col[uid + 1] do
7:         uid + +
```

This implementation has similar problems as both above two. Threads are doing binary searches, and variations in target list length, which are significant in power-law graphs, may cause load imbalance problems among threads. At the same time, as we have mentioned in edge-based implementation, adjacency list length has great effect on resource prefers for binary search, which is broadly used in this methods. What's more, the process of computing u, v and w is a computing intensive operation, making resource prefers a more remarkable problem.

3. ANALYTIC MODELS

By analysing existing GPU-based triangle counting algorithms, we will abstract two analytic models to evaluate the performance under different workloads.

3.1 Intra-block BSP Model

3.1.1 model description

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)	undirect edge between u and v
$u \to v$	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
α	exponent of $1/\widetilde{d}$ to c
β	exponent of BW to m
p_c	computing intensity pressure
mem_sup	sum of all vertices' memory superiority in a bucket

We find a common pattern in many triangle counting implementations: every time threads inside a block get their own tasks, some data process (such as setting global variables or prefetching some data to shared memory) and synchronization are always needed before all tasks begin. For example, Bission's work [6] needs to set bitmaps before threads start intersection steps; In Hu's work [13], a chunk of adjacency lists is firstly loaded into shared memory before starting all concerning binary searches. This pattern can be seen as a BSP (Bulk Synchronize Parallelism) model [23] within a block, in which global variables setting and all threads' private works form a superstep. We call such process as *intra-block BSP Model*. It is shown in Figure 6, in which we assume that there are only four threads for a block. Supersteps are performed repeatedly to finish the whole task.

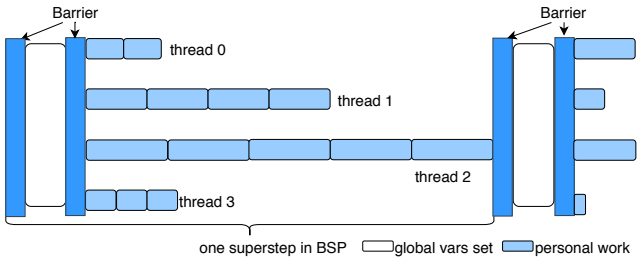


Figure 6: Intra-block BSP model

Intra-block BSP model adapts to GPU architecture, because threads within blocks often need to work collaboratively on the same workspace. To assure consistancy access to workspace, synchronization is necessary between read and write. A typical common workspace is shared memory, which is popular in recent triangle counting works [6, 13, 14] for low memory latency and high throughput. The limited space of shared memory makes it hard for threads to have their private space in it, thus threads have to work collaboratively on it, which conforms to intra-block BSP model.

It is easy to know the running time of each superstep always depends on the thread with the largest workload, e.g., thread 2 in Figure 6. In triangle-counting, the workload

size is measured by the adjacency list length of each vertex (i.e., vertex degree). For example, in Bission's work [6], threads of vertices with long adjacency lists have to check more times in bitmap; while in Hu's work [13], threads of vertices with long adjacency lists have to suffer longer memory latency and more binary search times. On the other hand, real-world graphs often employ the skew vertex degree distribution [1], which definitely leads to severe workload balance. However, in all triangle counting implementations, an undirected graph is transformed into directed one to avoid redundancy computation. This preprocessing provides an opportunity to change the vertex degree distribution. The following Table 2 shows the running time under different edge-directings.

3.1.2 Experimental Rationality

R-strategy	D-order	ours	Origin order		
D-strategy	D-direction			id-based	ours
gowalla	0.026	0.007	0.009	0.013	0.006
cit-patent	1.9	0.093	0.124	0.64	0.102
roadcentral	0.499	0.42	0.463	0.996	0.382
kron-log21	9.61	5.32	8.04	10.98	5.23

Table 2: Running time (sec) of four datasets under different vertex Reorder-strategies and edge Direction-strategies

In the last three columns of Table 2, given the same triangle-counting algorithm [13], we show the kernel running time of different edge direction strategies. The popular method degree-based edge directions (simplified as D-direction), i.e. edges are from vertices with smaller degree to those with bigger degree, is much better than id-based strategy, i.e. edges are from the small vertex ids to large vertex ids, because of more balanced adjacency lists lengths. Actually, our analytic model-based edge directing strategy (the last column in Table 2) can further improve the performance by 17.4% to 34.9%.

3.1.3 Analytic Model—the mathematical perspective

From the intra-block BSP model, we can conclude that to balance the workload of threads, we need to balance the adjacent lists size (i.e., vertex 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.

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

Our goal is to find the best edge-directing scheme $\mathcal{P}$ that minimizes the cost function $\mathcal{C}(\mathcal{P})$, which indicates the degree distribution is more evenly. In turn, it can get more balanced workload and better running performances. 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 model description

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

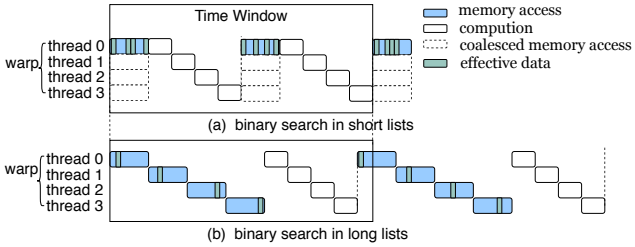


Figure 7: Binary search memory access patterns

As mentioned in Section 2, for an GPU algorithm, the computing cost and memory-access cost are not naturally balanced in existing methods. The binary search is a common computing pattern in GPU triangle counting algorithms, but, it shows totally different workload features. For long lists, the binary search is a memory access intensive task; but the computing cost dominates the task for short lists. We analyze the reason using Figure 7, assuming that a warp with four threads is doing binary searches. Four threads conduct the binary search over the same list L for different search keys. If L is short, due to the coalesced memory, all concerning data in the four threads can be loaded using a *single* memory transaction, as shown dash rectangles in Figure 7(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 7(b)). Given a fixed time window, the workload is more dominated by memory access cost in long lists than short lists.

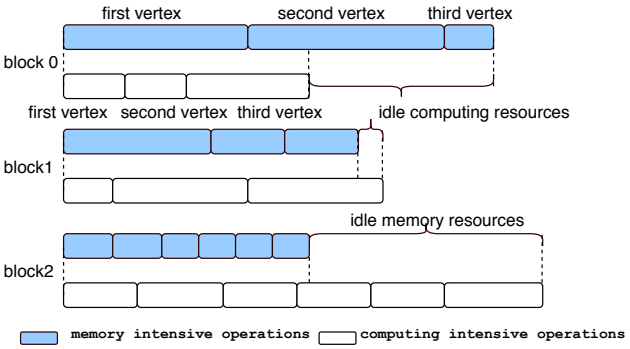


Figure 8: Resource-balanced model

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, some other warps will

be executed during the break to avoid waste of computing resources. For example, if two binary search warps on short lists and long lists (such as Figure 7(a) and (b)) is assigned to the same block, the warp W_1 can "steal" computing resource of the SM when the W_2 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-access intensive or computing intensive) together, which are dispatched to one block. In Figure 8, more computing resources are wasted in *block* 0 since the three vertices are all memory-access intensive tasks. *block* 2 in Figure 8 has an analogue problem. In contrast, *block* 1 has balanced resource requests, since it combines memory-intensive tasks with computing-intensive ones.

3.2.2 Experimental Rationality

Having the above observation, the following problem is how to bring up a good block task assignment scheme. In triangle counting problems, blocks usually fetch consecutive vertices as their work sets. So, a possible solution is to reorder the vertices in a given graph so as 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 order. It has the worst performance, even slower than the original vertex ordering greatly. In D-order, the vertices with similar degrees are always grouped together, which aggravate the resource usage imbalance, since they have the same resource prefers. On the contrary, our balanced resource model-based reordering scheme leads to the best performance.

3.2.3 Analytic Model—the mathematical perspective

As discussed above, the memory-access cost and computing 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. We will detail the function format and the parameters in Section 5.3. Given a vertex ordering $\mathcal{R}$, every consecutive k vertices are grouped into a bucket B_i. And there are b buckets in total. All concerning tasks with regard to a bucket are assigned to one block. We model the vertex reordering to change the block task assignment as follows:

For each bucket B_i, the corresponding compute 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 schema $\mathcal{R}$ 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 one in Figure 8. The parameter λ denotes the relative weight ratio between computing 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 hardwares, which will be discussed in Section 5.

Our analytic model's goal is to find the optimal reordering schema $\mathcal{R}$ to minimize Equation 3. The hardness of this optimization problem and the corresponding algorithm will be discussed in Section 5.

4. EDGE DIRECTION

In this section, we first study the hardness of the edge-directing problem and then propose the corresponding algorithm.

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 a NP-complete problem.*

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

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 $u \to v$ (or $v \to u$). 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, which says $\forall u, v, w \in V$ and $u \ne v \ne w$, then $\delta_{uv} + \delta_{vw} + \delta_{wu} \le 2$.

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

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

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 linear approximate algorithm.

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

All core vertices and non-core vertices are collected as a set V_c and V_n, respectively.

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 $v_n \to v_c$ 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 concerning part of it in the Equation 1 is

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

beacuse $\widetilde{d}(v_n) \leq d(v_n) < \widetilde{d}_{avg}$, 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 a 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-phrase algorithm, named "peeling algorithm". In the first phrase, we try to collect all non-core vertices v ($d(v) \leq \widetilde{d}_{avg}$) (see Lines 5-6 in Algorithm 2), inserting them 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 $v \to nbr$ (Line 9-16). The process can guarantee 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 smaller degree vertex to the larger one (line 9-11).

According to Lemma 4.1, it is easy to know the first phrase is an exact algorithm. The different problem is how to determine the edge direction between two core-vertices. Heuristically, we increase the threshold for the peeling processing (Line 19) and repeat the above peeling process as the second phase. The edge directing strategy is analogue to the first phrase. The above process is iterated until that no vertex is left. Obviously, the second phrase 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 2 is a performance-guarantee approximate algorithm. Actually, the first loop (Lines 4-18) is the first phrase and the rest loops are the second phrase.

Algorithm 2 Approximate Algorithm

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

```
1: Frontier ← ∅, nextFrontier ← ∅
2: threshold ← d̃_avg
3: while !all nodes peeled do
4:     for v ∈ allNodes do
5:         if d(v) ≤ threshold then
6:             Frontier.push_back(v)
7:     while Frontier not empty do
8:         for v ∈ Frontier, nbr ∈ v.nbrList do
9:             if nbr ∈ Frontier and d(nbr) ≥ d(v) then
10:                set edge : v → nbr
11:                d(nbr) − −
12:            if nbr ∉ Frontier and (v, nbr) is unset then
13:                set edge : v → nbr
14:                d(nbr) − −
15:                if d(nbr) ≤ threshold then
16:                    nextFrontier.push_back(nbr)
17:        Frontier ← nextFroniter
18:        nextFrontier.clear( )
19:    threshold ← threashold * 2
```

In reality, we increase *threshold* within limited times, and the overall time complexity of the peeling algorithm can be seen as $O(|E|)$.

4.3 Approximation Ratio

Assume that the redirecting-scheme generated by Algorithm 2 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.

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

where UB and LB represent the lower bound and upper bound, then we have

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

And

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

where $|V_{\widetilde{d}_{avg}+1}|$ denotes number of vertices whose degree is $\widetilde{d}_{avg} + 1$, and d_{peel} is a peeling threshold in a graph.

PROOF. We will discuss the ρ in two parts.

LB($\mathcal{C}(\mathcal{P}_{Opt})$) 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:

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

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

The Second part of the cost comes from vertices in V_c. The cost is

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

So we have

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

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 $\mathcal{C}(\mathcal{P}_{Opt})$

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

(b) $\frac{\sum_{v \in V_c} d(v) - \sum_{v \in V_n} d(v)}{2} - \widetilde{d}_{avg}|V_c| \ge 0$. Considering that $|CrossEdge| \le \sum_{v \in V_n} d(v)$, $\frac{\sum_{v \in V_c} d(v) - |CrossEdge|}{2} - \widetilde{d}_{avg}|V_c| \ge 0$. 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

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

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

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

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

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 $UB(\mathcal{C}(\mathcal{P}_{Alg}) - \mathcal{C}(\mathcal{P}_{Opt}))$. 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

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

□

Due to the complex analytical expression in Theorem 4.2, we present some intuitions about ρ. Power-law graphs are very popular in real-world. Assuming a power-law graph conforming to ACL graph model [1], we generate graphs and plot the curve of ρ with regard to the $\widetilde{d}_{avg}$ in Figure 9. It turns out that ρ is less than 1.8 in ACL graph model.

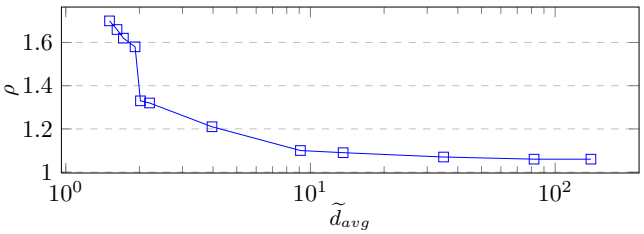


Figure 9: Approximate ratio under ACL power law graphs

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.

5. GRAPH REORDERING

A block gets a group of continuous vertices as work set, therefore we can reorder all vertices to make vertex's resource prefers in this group make up for each other. Based

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_g500-log21	1	1.16

on the resource balance model and mathematical defination in Section 3, we can focus on finding an optimal reordering method.

5.1 Hardness Analysis

THEOREM 5.1. *Given a directed graph $\mathcal{G}$ (that is obtained by applying the edge-direction in Section 4), 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 problem (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, satisfing $C_1 = M_1$, $C_2 = M_2$ 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 *partition problem* polynomially reducts to DP. Consider a *partition problem* instance: $t \in \mathbb{Z}^+$, we have a set of $2t$ elements, the sum of which is $2S$, and $S\%t^2 = 0$. The i^{th} elements of the set is $a_i t + 1$, $a_i \in \mathbb{Z}^+$, and we want the set to be evenly separated into two sets, sum of which is S. In the correlated DP problem, there are $2t$ vertices: $m_i = a_i t + 1$, while $c_i = \mathrm{S/t}$ and $C_{max} = S$. This transformation is evidently polynomial time.

Supposing 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 all 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 $C_1 = C_2 = \mathrm{S} = M_1 = M_2$, 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 for dealing with the graph reordering problem, assuming vertices as basic units. Algorithm 3 gives the pseudocode of it. It is basically a greedy algorithm. Before it, we need to obtain the function F_c, F_m and parameter λ. We will talk about those parameters in next subsection.

The aim of this algorithm is to dispatch vertices to buckets so that computing resources and memory resources are

Algorithm 3 Algorithm for Graph Reordering

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

under balanced use in each bucket. Generally speaking, every time we deal with a memory-dominated vertex, we give it to bucket with least memory resources usage superiority, and computing-dominated vertices to bucket with most memory resource usage superiority. Firstly, all buckets are initalized, with their *mem_sup* set to 0 (line 1). For a vertex v, we call $F_m(\widetilde{d}(v)) - \lambda F_c(\widetilde{d}(v))$ as memory superiority. The variable *mem_sup* denotes sum of all vertices' memory superiority in a bucket. Then all buckets are made into a minimum priority queue according to *mem_sup* (line 2). All vertices are seperated into two sets: memory-dominated vertices and computing-dominated vertices (line 3-4). Vertices in the first set are firstly arranged: the vertex is put into bucket in the queue top (line 6-7), and concerned bucket information is updated. Then the bucket is inserted back to the queue (line 9). This process is carried out repeatly for all memory-dominated vertices. 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 similarly as the above process. Finally we reorder the whole graph, making sure that vertices in the same bucket have consecutive ids (line 16-20).

This is a light-weight algorithm in that the complexity is $|V|log|B|$, in which $|B|$ is far less than $|V|$. Therefore it's nearly a linear complexity algorithm.

5.3 Parameter Determination

In fact, 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.

Apparently, in implementations with different reordering units, those parameters are different, but we can get them using similar methods. In the following text, we will only introduce how we get all parameters in the wedge-based implementation [13] as an example, which use vertex as reordering units. And similar methods adapt to implementations using edges. These parameters are task-determined, so we only

need to figure them out once for multiple datasets.

functions $\mathbf{F_c}$ and $\mathbf{F_m}$ As we have analysed, computing intensity c is directly determined by getting u, v and w of new subtasks, which we call "subtask id" for convenience. Intuitively, the frequency of a thread to fetch new subtask id is directly determined by adjacency list length: threads working on longer adjacency list have less calculations of getting a new subtask id. For example, a thread will get plenty of subtasks before it has to change its u while working on a vertex with many neighbors. Besides, we can draw the same conclusion from Figure 7. We conclude that the c of a vertex v is negtive correlated with $\widetilde{d}(v)$, so we define:

$$F_c(v) = 1/\widetilde{d}(v)^{\alpha} (\alpha > 0) \tag{18}$$

As for m, we think that concerned shared memory bandwidth BW is a good measurement of memory access intensity. By collecting vertices with same degrees and running them alone with *nvprof*, we can obtain functional image of BW in Figure 10. The memory intensity is positive correlated with BW, so

$$F_m(v) = BW(v)^{\beta} (\beta > 0) \tag{19}$$

Parameter setting α, β and λ are discussed as follows.

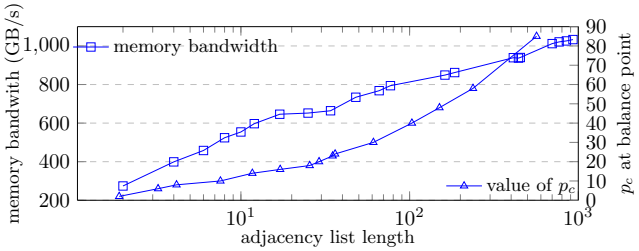


Figure 10: Shared memory usage and the value p_c at balance point concerning adjacency list length

parameter λ It is a parameter that can transform c to m, and we need to figure out λ as carefully as possible. 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 computing task. So here we try to get λ by experiments.

We have known that vertices with long adjacency lists need more memory resources than computing resources, therefore if we add a little more computing pressure to them, the overall executing time will remain unchanged. But if we keep forcing more computing pressure, i.e. exectuing codes for getting new subtask id for extra (denoted as p_c) times, bit by bit, the computing resources usage will reach the equal memory resource usage. And a bit more computing pressure, the executing time will surpass the base time. We call this situation *balance point*. At *balance point*, we have:

$$m = \lambda(p_c \times c) \tag{20}$$

By collecting vertices of same adjacency list length together and executing them alone, we can get p_c in different degree vertices. We carried out multiple experiments on several representative datasets to get more accurate p_c. And in our experiments, "overall executing time will remain unchanged" only allows 5% deviation. Figure 10 shows the functional image of computing pressure p_c. We can see that p_c grows as the adjacency list length grows. The image only covers cases of memory-dominated vertices, and for computing-dominated vertices, we can also get their memory pressure threshold p_m from experiments.

According to our analyse, Equation 20 should be satisfied at *balance point*, and it is a direct proportional function, the ratio of which is the λ. We have known the representation of the two variables c and m, and after trying verious of options, we find that if

$$F_c(v) = \sqrt{1/\widetilde{d}(v)},\ F_m(v) = \sqrt{BW(v)} \tag{21}$$

, the functional image of m related to $c \times p_c$ can be well fitted into a function of direct proportion, as shown in Figure 11. And the constant of variation is λ. In our experiment, the λ is 9.682.

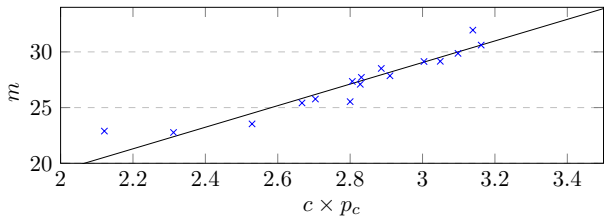


Figure 11: Fitting function

So far, we have figure out all parameters of the optimization problem. The computing intensity and memory intensity is inferred by analyse and λ is obtained by experiments.

6. RELATED WORK

In this paper, we review related work in two categories: existing analytical model for GPU computing and state-of-the-art triangle counting algorithms.

6.1 Analytical Model for GPU Computing

Existing analytical models for GPU computing are mainly about performance predictions. For example, Hong et al. [12] propose a prediction model for algorithms on GPU, which introduces two metrics for judging resource prefers of parallel programs, and estimating the number of memory requests by considering the number of running threads and memory bandwidth. Amaris et al. [2] also propose an execution time prediction model, mixing computation, memory access and cache usage. Ma et al. [17] design a performance analytic model for memory-limited kernels, focusing on tuning various configuration parameters such as *gridDim* and *blockDim*. Yang et al. [28] work on matrix multiplication, but 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 analytical models in graph data preprocessing so as to speed up a bunch of corresponding algorithms. To the best of our knowledge, no existing work study this problem.

6.2 Triangle Counting Algorithms on GPU

There are mainly three ways [25] for triangle counting on GPU: matrix multiplication [4,5], subgraph isomorphism and list intersection. Generally, the list intersection-based triangle counting algorithms are often better than other two categories. List intersection is the most significant operation in triangle counting, while binary search and merge-based are two typical list intersection algorithms. Han's

work [11] speeds up list intersections on parallel archtectures using sort merge methods and use bit-compression for higher parallelism. Some recent works [3, 13, 14] show that binary search is better than merge-based list intersection for triangle counting on GPU because of better independency and larger parallelism. Ao [3] et al. narrow range of binary search using liner regression and hash methods. List intersection-based GPU triangle counting algorithms can be divided into three types according to their basic unit of work distribution:

vertex-based Bission et al. [6] came up with an vertex-based implementation: they assign a vertex to a block and a thread is responsible for a neighbor's computation task. He uses bitmap in shared memory or global memory for fast look up of a list. Wang et al. [26] have also given a vertex-based implementation in Gunrock library. Usually vertex-based methods have better memory access performance because of high reuse of adjacency lists.

edge-based A basic implementation of edge-basic method [18] uses a thread for dealing with an edge. And in TriCore [14], each warp is dispatched to deal with an edge; what's more, they use shared memory for enhance list intersections. To better balance workload of different threads, Green et al. presents a method [9, 10] that estimates workload of each edge in advance and then assigns threads to edges accordingly, which outperforms most edge-based methods. Compared with vertex-based method, edge-based methods usually mean finer work granularity and thus better workload balance.

wedge-based Hu et al. [13] present a wedge-based method, which keeps the data arrangement of vertex-based method, but uses finer-grained work distribution. In their method, a thread just answers a simple question: given a fragment $u - v - w$, where (u, v) and (v, w) are confirmed edges, are u and w neighbors? And the thread answers this question by doing a binary search in u's adjacency list. A wedge is in fact their basic work distribution unit. Instutively, this implementation achieves better workload balance because of finer work granularity. Also, it keeps vertex-centric data structure so the memory access is better than those edge-based methods.

In fact, the three of our introduced implementations includes all these work distribution methods, which are very representative.

7. EXPERIMENTS

7.1 Settings

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

Datasets: Both real-world and synthetic datasets are used. We obtain real-word datasets from Stanford Network Analysis Project(SNAP)[1] and HPEC graph challange[2], and synthetic datas are obtained by kronecker generator[3]. Table 4 shows statistics of all real-world datasets we use in the following experiments. We also use synthetic graph generators (BA graph, Random Power Law graph) provided by SNAP to study the performance of our two models.

Comparative Methods:We use Hu's [13] and Bission's implementation [6] to verify effectiveness of intra-block BSP model, while Hu's [13] and Fox's [9] implementation for resource balance model. The two models are used in edge-directing and vertex-ordering, respectively. Comparative experiments are conducted for those implementations using our analytic model-based optimizations with the baseline strategies. The baseline strategies are degree-based ("D-direction" for short) for edge-directing and original vertex id-based ("Original" for short) for vertex ordering, respectively.

Table 4: Datasets Infos

dataset	nodes	edges	triangles
email-Euall	265K	729K	267,313
gowalla	197K	2M	2,273,138
road_central	14M	17M	228,918
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_g500-log18	25M	25M	281,814,846
kron_g500-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

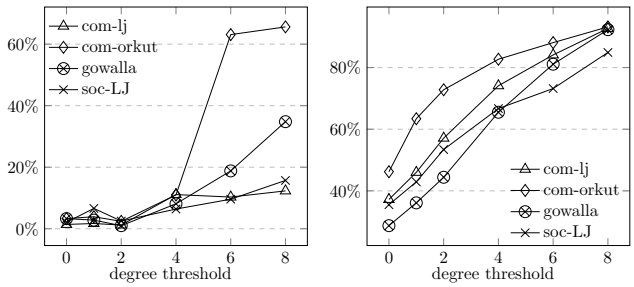
7.2 Intra-block BSP Model

Intra-block BSP model is used in edge-directing scheme to balance the workloads. 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.

Hu′s implementation. We compare the cost of the whole graph according to our cost function Equation 1 under three edge direction methods: id-based, degree-based (D-direction in Figure 12(a)) and our methods. Figure 12 shows the overall cost decline percentage of our method compared with the other two methods on four graph datasets. It's easy to conclude that vertices with large degree have especially great influence on the performance, so we pay extra attention to those vertices and *degree threshold* as k in Figure 12 means we only add up cost of vertices with degree larger than $k * \widetilde{d}_{avg}$. From Figure 12, our method achieves increasingly better effect as the *degree threshold* increases.

Then the running times of three edge direction methods on real-world datasets are shown in "bars" in Figure 13. Note the running time includes two parts: the up part of the bar is edge-direction preprocessing time and the bottom part is GPU kernel running time. It is easy to conclude that both our method and degree-based approach (D-direction) are faster than the id-based scheme greatly. 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) in "curves" in Figure 13. Our strategy works better than D-direction on almost all datasets; we achieve 9% $\sim$ 42%

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



(a) cost decline to D-direction (b) cost decline to id-based

Figure 12: Cost decline of our methods compared with other edge direction strategies

improvement on kernel time and 1% ∼ 34% improvement on total time.

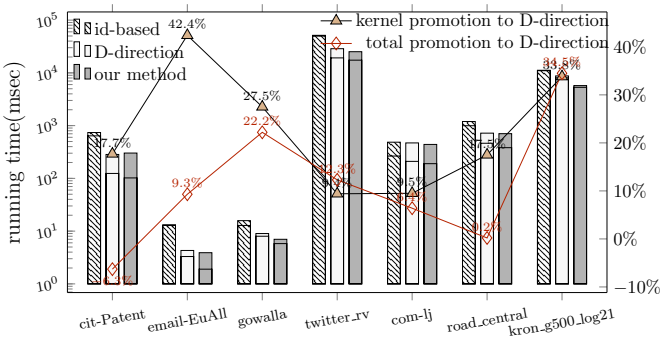


Figure 13: Running time of different edge direction methods

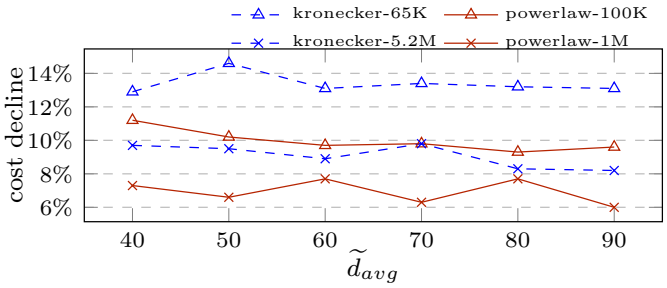
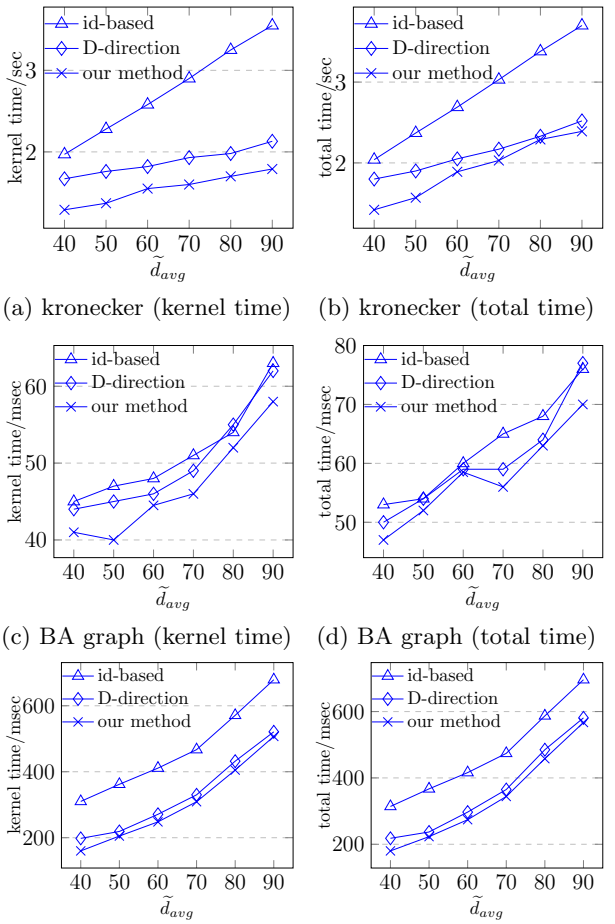


Figure 14: Cost decline compared with other edge direction strategy

Bission′s implementation. In this implementation, a vertex will get a block to finish its work only when the $\widetilde{d}_{avg}$ of the whole graph is larger than 38. Only in this condition our intra-block BSP model applies to its implementation. Few real-world graphs satisfy the degree requirments. Instead, we choose synthetic datasets, so that it's more convenient to evaluate the performance with requisite $\widetilde{d}_{avg}$.

Figure 14 shows the reduction percentage of cost (Equation 1) in two synthetic graphs with different $\widetilde{d}_{avg}$ and different vertices sizes, comparing with the degree-based (D-direction) approach. We can see that the cost reduction percentage is comparatively steady with $\widetilde{d}_{avg}$, but changes a little obviously with graph vertices size. Generally, our



(a) kronecker (kernel time) (b) kronecker (total time)

(c) BA graph (kernel time) (d) BA graph (total time)

(e) random power-law graph (kernel time) (f) random power-law graph (total time)

Figure 15: Running time comparison of Bission's vertex-based work

analytic model-based approach has about 6%∼15% cost reduction.

We choose three kinds of synthetic graphs to measure performance promotion: kronecker graph, BA graph and random power-law graph. Similarly, graphs are generated with $\widetilde{d}_{avg}$ from 40 to 90 to make the results more credible. Running time promotions of kernel and total time are shown in Figure 15. Similar as Hu's work, id-based method has the worst performance. Our method achieves better performance than D-direction strategy on kernel time, especially in kronecker graph and random powerlaw graph. And total time promotions shows similar results. In general, kernel promotions are more remarkable than total time promotions.

In conclusion, our edge direction method effectively reduces the cost raised in Equation 1 for two state-of-art implementations, and thus achieving good performance improvements on both kernel running time and total time.

7.3 Resource Balance Model

The resource balance model is used to find a good vertex ordering for better resource usage balance. To only measure the effectiveness of our vertex-reordering strategy, we adopt the D-direction method in all implementations.

Hu′s implementation We experiment on three ordering

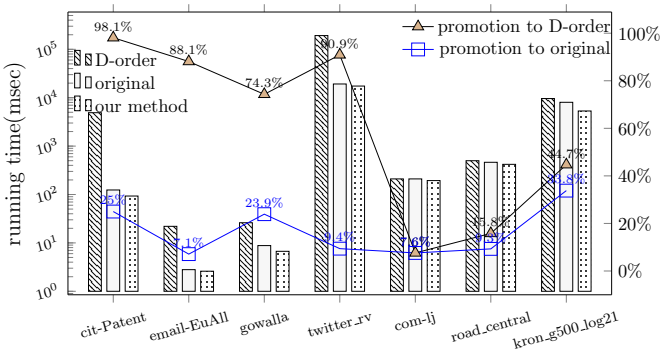


Figure 16: Experiments on Hu's work of resource balance model

strategies: original vertex order (Original), degree-based (D-order) and our method. Note that our algorithm achieves nearly linear complexity to $|V|$, consequently the preprocessing time for different vertex orderings is much smaller than GPU kernel times in experiments. Thus, we only report the GPU kernel times in Figure 16. Generally, the D-order performs the worst in most datasets, since the workloads with the similar resource prefers are assigned to one block, leading to more resource conflicts. Our method outperforms almost all datasets, achieving 7.4% $\sim$ 33.8% improvements on average than Original, and over 50% improvement on average than D-order (shown in "curves" in Figure 16).

Fox's implementation Fig 17 shows results of balanced model and original distribution in several datasets. In fact this work is a little different from Hu's work in reordering: we reorder vertices to balance memory and computing resource usage inside blocks in Hu's work; while in this implementation, we do this rearrangement to edges in that edges are basic work units for it. But we use almost same method to deal with edges as to deal with vertices. So in this implementation we can't measure the performance of D-order. Finally we achieve 10.5%$\sim$25% performance improvement in the given datasets. Specially for this implementation, our strategy performs better in large scale datasets.

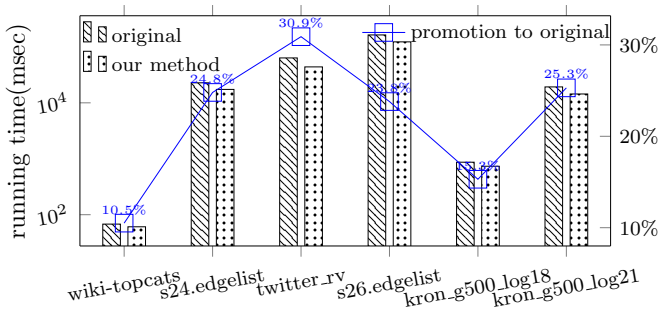


Figure 17: Results of Fox's work

7.4 Put It All Together

Note that our analytic models-based edge direction and vertex ordering strategies are orthogonal to each other. To evaluate the performance of the combined method, we implement the two strategies in Figure 18. The curves show the performance promotions with only implementing the analytic model-based edge direction or vertex ordering. Generally, the combined approach can speed the overall running

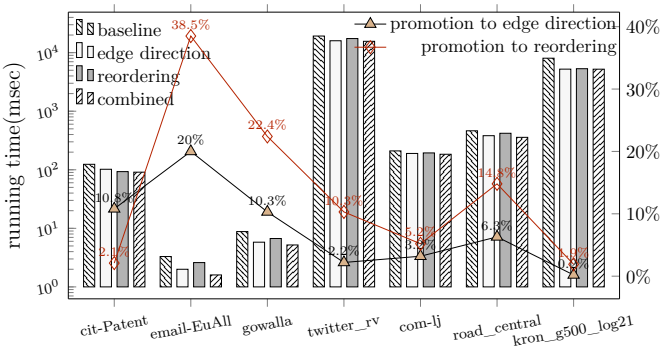


Figure 18: Combining two models on Hu's work

time by 7.6% on average comparing with only model-based edge direction, and 13.6% on average comparing with only model-based vertex reordering.

8. CONCLUSION

Triangle counting is a fundamental graph algorithm due to its wide application. Therefore, 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-art triangle counting algorithms. Based on our proposed analytic models, we propose model-guided edge-direction and vertex reordering 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.

9. REFERENCES

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

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

[3] N. Ao, F. Zhang, D. Wu, D. S. Stones, G. Wang, X. Liu, J. Liu, and S. Lin. Efficient parallel lists intersection and index compression algorithms using graphics processing units. *Proceedings of the VLDB Endowment*, 4(8):470–481, 2011.

[4] A. Azad, A. Buluç, and J. Gilbert. Parallel triangle counting and enumeration using matrix algebra. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop (IPDPSW)*, pages 804–811. IEEE, 2015.

[5] L. Becchetti, P. Boldi, C. Castillo, and A. Gionis. Efficient semi-streaming algorithms for local triangle counting in massive graphs. In *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 16–24. ACM, 2008.

[6] M. Bisson and M. Fatica. High performance exact triangle counting on gpus. *IEEE Transactions on Parallel and Distributed Systems*, 28(12):3501–3510, 2017.

[7] M. Burtscher, R. Nasre, and K. Pingali. A quantitative study of irregular programs on gpus. In *2012 IEEE International Symposium on Workload Characterization (IISWC)*, pages 141–151. IEEE, 2012.

[8] F. Busato and N. Bombieri. An efficient implementation of the bellman-ford algorithm for kepler gpu architectures. *IEEE Transactions on Parallel and Distributed Systems*, 27(8):2222–2233, 2015.

[9] J. Fox, O. Green, K. Gabert, X. An, and D. A. Bader. Fast and adaptive list intersections on the gpu. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2018.

[10] O. Green, J. Fox, A. Watkins, A. Tripathy, K. Gabert, E. Kim, X. An, K. Aatish, and D. A. Bader. Logarithmic radix binning and vectorized triangle counting. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2018.

[11] S. Han, L. Zou, and J. X. Yu. Speeding up set intersections in graph algorithms using simd instructions. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1587–1602. ACM, 2018.

[12] S. Hong and H. Kim. An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness. In *ACM SIGARCH Computer Architecture News*, volume 37, pages 152–163. ACM, 2009.

[13] L. Hu, N. Guan, and L. Zou. Triangle counting on gpu using fine-grained task distribution. In *2019 IEEE 35th International Conference on Data Engineering Workshops (ICDEW)*, pages 225–232. IEEE, 2019.

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

[15] F. Khorasani, K. Vora, R. Gupta, and L. N. Bhuyan. Cusha: vertex-centric graph processing on gpus. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*, pages 239–252. ACM, 2014.

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

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

[18] A. Polak. Counting triangles in large graphs on gpu. In *Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International*, pages 740–746. IEEE, 2016.

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

[20] J. Soman, K. Kothapalli, and P. Narayanan. Some gpu algorithms for graph connected components and spanning tree. *Parallel Processing Letters*, 20(04):325–339, 2010.

[21] H.-N. Tran, J.-j. Kim, and B. He. Fast subgraph matching on large graphs using graphics processors. In *International Conference on Database Systems for Advanced Applications*, pages 299–315. Springer, 2015.

[22] C. E. Tsourakakis, P. Drineas, E. Michelakis, I. Koutis, and C. Faloutsos. Spectral counting of triangles via element-wise sparsification and triangle-based link recommendation. *Social Network Analysis and Mining*, 1(2):75–81, 2011.

[23] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[24] J. Wang and J. Cheng. Truss decomposition in massive networks. *arXiv preprint arXiv:1205.6693*, 2012.

[25] L. Wang, Y. Wang, C. Yang, and J. D. Owens. A comparative study on exact triangle counting algorithms on the gpu. In *Proceedings of the ACM Workshop on High Performance Graph Processing*, pages 1–8. ACM, 2016.

[26] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens. Gunrock: A high-performance graph processing library on the gpu. In *ACM SIGPLAN Notices*, volume 51, page 11. ACM, 2016.

[27] D. J. Watts and S. H. Strogatz. Collective dynamics of 'small-world'networks. *nature*, 393(6684):440, 1998.

[28] X. Yang, S. Parthasarathy, and P. Sadayappan. Fast sparse matrix-vector multiplication on gpus:

implications for graph mining. *Proceedings of the VLDB Endowment*, 4(4):231–242, 2011.