Wedge-Parallel Triangle Counting for GPUs

Jeffrey Spaan[1](✉), Kuan-Hsun Chen[1], David A. Bader[2], and Ana-Lucia Varbanescu[1]

[1] University of Twente, Enschede, The Netherlands
{j.p.spaan,k.h.chen, a.l.varbanescu}@utwente.nl

[2] New Jersey Institute of Technology, Newark, USA
david.bader@njit.edu

Abstract. For fast processing of increasingly large graphs, triangle counting – a common building block of graph processing algorithms, is often performed on GPUs. However, applying massive parallelism to triangle counting is challenging due to the algorithm's inherent irregular access patterns and workload imbalance. In this work, we propose WeTriC, a novel *wedge*-parallel triangle counting algorithm for GPUs, which, using fine(r)-grained parallelism through a lightweight static mapping of wedges to threads, improves load balancing and efficiency. Our theoretical analysis compares different parallelization granularities, while optimizations enhance caching, reduce work-per-intersection, and minimize overhead. Performance experiments indicate that WeTriC yields 5.63× and 4.69× speedup over optimized vertex-parallel and edge-parallel binary search triangle counting algorithms, respectively. Furthermore, we show that WeTriC consistently outperforms the state-of-the-art (i.e., on avg. 2.86× faster than Trust and 2.32× faster than GroupTC).

Keywords: Triangle Counting · Graph Processing · Parallel Computing on GPUs · Wedge-Parallel Approaches

1 Introduction

Graphs are flexible data structures that can efficiently capture entities (vertices) and their relations (edges). Graph processing algorithms – like shortest path, page rank, or betweenness centrality – are built to analyze such interconnected data to extract non-trivial (often statistical) information.

Graph processing algorithms vary in complexity, and become challenging to solve in reasonable time for very large input graphs. In fact, graph processing performance is dependent on the graph properties, the selected algorithm(s) and their implementation, and the hardware platform of choice. Especially when using parallel systems - like multi-core CPUs and GPUs - choosing an incorrect mix of (algorithm, implementation, platform) for the given graph workload can lead to significant performance loss (i.e., 2–3 orders of magnitude) [22].

W. E. Nagel et al. (Eds.): Euro-Par 2025, LNCS 15902, pp. 3–17, 2026.
https://doi.org/10.1007/978-3-031-99872-0_1

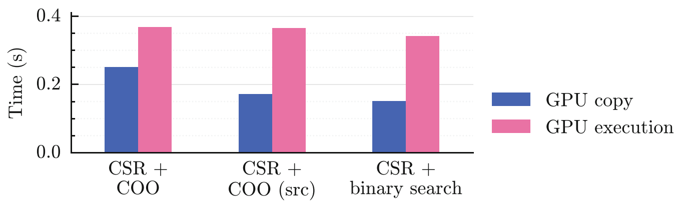


Fig. 1. Performance for different edge-retrieval strategies for the Wikipedia graph (see Table 3). Each thread finds their edge (v, w) and performs BINARYSEARCH$(N(w), v)$.

This work focuses on a specific graph processing problem – triangle counting – and a specific target platform – a GPU . Triangle counting is a common building block for algorithms such as k-truss [23], clustering coefficient [20], and link recommendation [7]. While many algorithms [1,3,6,11,14,17,18,24] have been proposed, we observe that the task size of these algorithms is often imbalanced. Commonly, a vertex or an edge is assigned for processing to every thread, warp or block. However, the work per vertex or edge is graph-dependent, and can vary wildly. Their remedies, such as dynamic scheduling (like work-stealing) and oversubscribing, not only create an overhead in managing and synchronizing the workload – they can also lead to underutilization due to many idling threads.

Our algorithm, WETRIC, reduces this waste by introducing a new, finer, *wedge-parallel* granularity, which, through a static lightweight array of indices (equal in size to the number of vertices), minimizes load imbalance. Its performance stems from two key insights: (1) allocating more threads or blocks is inexpensive, meaning that we can process one wedge per thread even for a very large number of wedges, and (2) inferring the vertices of a wedge is a relatively cheap operation. Figure 1 shows the execution time for a kernel that retrieves an edge (v, w) and computes BINARYSEARCH$(N(w), v)$. From left to right, the source vertex v is retrieved from an edge list (COO), a source-only edge list, and by binary searching the CSR's row pointer. We observe that binary searching is always favored over storing an edge list, meaning that this approach can save both time and space. Thus, in the same vein, we apply a similar strategy for wedges to overcome the space restrictions of fully materializing a *wedge list.* This paper makes the following contributions:

– We present a novel GPU algorithm WETRIC for counting triangles in static undirected sparse graphs (Sect. 3), together with an in-depth complexity analysis (Sect. 4).
– We apply and evaluate a series of algorithm optimizations, on top of our wedge-based approach, to further balance the workload, decrease overhead, improve caching, and reduce the work per intersection (Sect. 5).
– We provide a C+CUDA implementation[1] of the algorithm, and show its scalability and performance across a diverse pool of graphs. The evaluation shows that WETRIC consistently outperforms the state-of-the-art (Sect. 6).

[1] https://github.com/jeffreyspaan/wedge-parallel-triangle-counting.

Table 1. Graph terminology.

$G(V, E)$	A *graph* with vertices V and edges E.	$N(v)$	*Neighborhood* of v (i.e., $w \in N(v)$ for every edge (v, w).
v	A *vertex* or *node* with ID v.	$d(v)$	The *degree* of v (i.e., $\|N(v)\|$).
(v, w)	An *edge* from v to w (we say w *neighbors* v).	$N(v)_{w+}$	Neighborhood of v where $u > w$ for all $u \in N(v)$.
(v, w, u)	A *wedge* where v is a *base* vertex and w and u are *leaf* vertices.	$d(v)_{w+}$	$\|N(v)_{w+}\|$.
n	Number of vertices (i.e, $\|V\|$).	d_{avg}	Average degree (i.e., $\frac{m}{n}$).
m	Number of edges (i.e, $\|E\|$).	d_{max}	Maximum degree.

2 Background and Related Work

2.1 Graphs and Data Structures

Graphs are collection of vertices connected by edges. Graph topology – that is, intuitively, the 'shape' of the graph – is characterized by a graph's (statistical) properties, like average degree or diameter. We summarize all graph properties used in this work, and their notations, in Table 1.

Dense graphs can be captured in an $n \times n$ *adjacency matrix*, where every possible edge, whether present in the graph or not, is stored on disk. For large sparse graphs (i.e., $m \ll n^2$), this structure is far too wasteful since most entries are zero. Instead, sparse graphs (and matrices) are commonly stored in compressed formats, like Compressed Sparse Row (CSR) or Coordinate (COO). The COO format consists of two arrays: one for the source vertices v and one for the destination vertices w. The CSR format compresses the former into indices (the *row pointer*) which point into a destination vertices array (the *column index*).

2.2 Triangle Counting

A *clique* is a subgraph in which each vertex is directly connected to all the other vertices in the clique. Triangle counting is the operation of determining how many triangles (3-cliques) are in a graph. The main strategy for triangle counting is *intersecting*: for every vertex v, determine how much v's neighborhood overlaps with the neighborhood of each of v's neighbors w. In other words: we want to find (the size of) the intersection $N(v) \cap N(w)$ for all $w \in N(v)$.

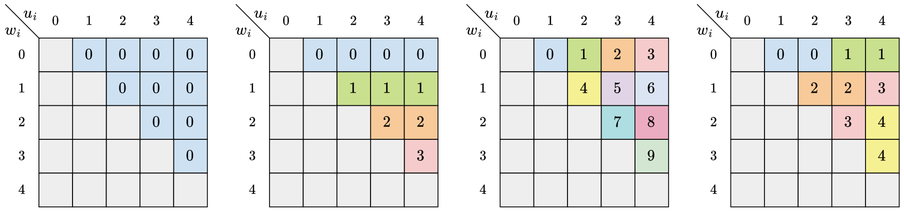
Binary search is the most common strategy for triangle counting on GPUs. It intersects by binary searching each u from $N(v)_{w+}$ in $N(w)$ for each w in $N(v)$. Binary-search-based triangle counting is attractive for many-core architectures because each wedge is independently evaluated. Furthermore, since the worst case-complexity per-wedge is logarithmic, i.e., $\mathcal{O}(\log(d(w)))$, it can reduce the workload difference between w's with diverse degrees. Examples include HTC [24], GroupTC [14], TriCore [11], and HyKernel [1].

Merge-path intersects by merging $N(v)$ and $N(w)$ at the same time. The obvious benefit of this technique is that $N(w)$ is only searched once. In turn, this means the per-edge worst-case complexity of this algorithm is $\mathcal{O}(d(v) + d(w))$, which is lower than the alternatives. Although predominantly used for sequential triangle counting algorithms, parallelized versions have been proposed by, among others, Mailthody et al. [15] and Pearson et al. [19].

Hashing replaces the binary search with a hash-table lookup. The main benefit of hashing-based triangle counting is the $\mathcal{O}(1)$ complexity for evaluating a wedge. The drawbacks are the construction and synchronization overhead, collisions – multiple vertices sharing the same hash, and increased workload imbalance. Hashing is implemented in the GraphChallenge[2] champion H-INDEX [17] and its successor TRUST [18]. On a coarser scale, Bisson et al. [6] use a bitmap (i.e., a full adjacency matrix row) while HTC [24] uses a *segmented* bitmap approach, where empty parts of the bitmap are omitted.

2.3 Parallelization Granularity

The granularity denotes the level at which the workload is divided between processors. Choosing a different granularity can change the amount of work per processor, the context, and the overlap between processors. For GPUs, the workload is often parallelized by the vertices, by the edges, or by a combination of the two. Figure 2 contrasts the processing of a vertex v with $d(v) = 5$ in vertex-parallel (1 thread), edge-parallel (4 threads) and wedge-parallel (10 threads).



(a) Vertex-parallel **(b)** Edge-parallel **(c)** Wedge-parallel, *spread* = 1 **(d)** Wedge-parallel, *spread* = 2

Fig. 2. Assignment of threads to the wedges of v, depicted as a triangular $N(v) \times N(v)$ matrix. Each cell is a wedge (v, w, u) comprised of v plus the neighbor shown in the horizontal ($u = N(v)[u_i]$) and vertical ($w = N(v)[w_i]$) axis. Cell numbers indicate the ID of the executing thread.

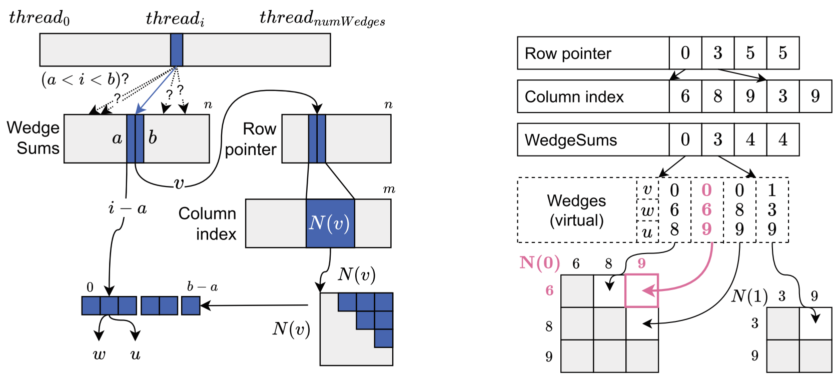
Vertex-parallel. Parallelizing by the vertices, where each thread calculates the triangles for a fixed number of vertices, allows for flexibility in the way that $N(v)$ and $N(w)$ are iterated because each thread has exclusive access to all wedges of v. Unfortunately, this is also a drawback: because the overlap between adjacent threads is low, there are few opportunities for reusing and/or coalescing data. HTC [24] uses this granularity for low-degree vertices.

[2] https://graphchallenge.mit.edu/.

Edge-parallel. Alternatively, one can assign a fixed number of edges to each thread. As shown in Fig. 2b, the maximum difference in the number of wedges per thread is now $d(v) - 1$, compared to d_{max}. HTC [24] uses this granularity for high-degree vertices. HyKernel [1] and H-INDEX [17] exclusively parallelize by the edges. Bader et al. [4] parallelize by the *edge cover*, a subset of the edges.
Vectorized. Some algorithms do not map vertices or edges to *threads* but rather to blocks, warps, or subwarps. Here, each collection of (a fixed number of) threads iterates over the wedges of one vertex or edge until all wedges are exhausted. Examples include GroupTC [14], Hu's algorithm [10], Bisson's [6], and TRUST [18]. A drawback is that this approach can under- or overshoot the number of threads required (a.k.a. oversubscribing). For instance, a warp-based approach (32 threads) has 31 idling threads for a 1- or 63-wedge vertex.

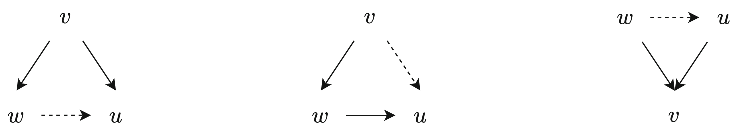
3 WeTriC

The input of our algorithm is an undirected graph, which we scan for duplicates, reorder (see Sect. 5.1), and turn into a directed graph (in CSR format). This procedure is identical to the state-of-the-art. Next, instead of storing the wedges, which would not be scalable (e.g., the Wikipedia graph in CSR format requires ~1 GiB, while storing all wedges would take up ~75 GiB), we construct the wedges dynamically. Our approach, as presented in Fig. 3, allows each thread to dynamically construct its wedge tuple (v, w, u) with only n additional memory. On the GPU, each thread follows the following three steps:
(a) Finding the Base Vertex v. First, each thread (with index i) searches an array named `wedgeSums` containing the cumulative number of wedges per vertex to find the index v where a = `wedgeSums`$[v] \leq i < b$ = `wedgeSums`$[v+1]$ is true. In other words, if we consider all wedges stored in an array (like the virtual array in Fig. 3b), a and b define the boundaries of the wedges of vertex v in that



(a) Wedge i (v, w, u) is highlighted. **(b)** Wedge 1 $(0, 6, 9)$ is highlighted.

Fig. 3. An overview of the algorithm, focusing on the process of retrieving a wedge: (a) the mechanism and (b) an example.



(a) Outgoing **(b)** Arrow **(c)** Incoming

Fig. 4. Wedge styles. Closing edges are dotted.

array, in the same way that $\texttt{rowPointer}[v] \leq e < \texttt{rowPointer}[v + 1]$ define the boundaries of all edges e in the column index for vertex v.

In practice, we calculate `wedgeSums` a priori using a prefix sum of the per-vertex wedge counts: $\binom{d(v)}{2}$. Furthermore, on the GPU, each thread performs a *binary* search to find v (and thereby a and b). This operation is relatively inexpensive because adjacent threads will share all but one level of the binary search tree, which allows for coalescing and/or caching of the lower levels.

(b) Finding the Leaf Vertices w and u. Knowing both v and a, we can find the relative index of this wedge within the wedges of v: $i - a$. If we consider all neighbor pairs w and u (i.e., the wedges) of v laid out in a $N(v) \times N(v)$ grid (like in Fig. 2), the cell with linear index $i - a$ will give us the x and y coordinates[3] (i.e., the w and u) of this wedge. In fact, this grid is an upper triangular matrix as we only consider wedges where $v < w < u$ to only count each triangle once.

(c) Intersecting. When the thread has calculated the base and leaf vertices, it binary searches $N(w)$ for u to determine whether the wedge (v, w, u) is a triangle. This operation has a worst-case complexity of $\mathcal{O}(\log(d(w)))$.

4 Theoretical Analysis

4.1 Wedge Styles

Along with the granularity, the wedge *style* is a key component of the workload distribution. Besides *outgoing* wedges, there are two other types of wedges (as depicted in Fig. 4): *arrow* wedges, and *incoming* wedges . The latter is, for our purposes, identical to the outgoing variant. Arrow wedges, however, are substantially different. Specifically, the number of consecutive closing edges (v, u) with the same target neighborhood is $d(v) \cdot d(w)$ compared to $d(v)_{w+}$ for outgoing wedges (on average, $(d_{\text{avg}})^2$ versus $\frac{d_{\text{avg}}-1}{2}$). Trust [18] leverages this property to make hashing the neighbors of v more efficient in terms of lookups per table and total number of tables (n versus m). The arrow style does come with an important drawback: the total number of wedges increases to $\sum_{v=0}^{n} \sum_{w \in N(v)} d(w)$ (assuming every degree is d_{avg}, $\sim 2\times$ more). For this reason, and because inferring an arrow wedge is costlier, this style was found to perform worse.

[3] The conversion of linear to Cartesian coordinates for a triangular matrix of a fixed size ($d(v)$), which was not discovered by us, finds its origin in triangular numbers [2].

In contrast, for the edge-parallel algorithm, we found that using arrow wedges was faster than using outgoing wedges, likely because the number of adjacent threads searching the same neighborhood is $d(v)$, compared to 0.

Lastly, we can also alternate the outgoing and arrow style dynamically depending on the current workload. We refer to this style as *mixed*. Interestingly, the vertex-parallel algorithm prefers the mixed wedge style. We believe this is because it has the lowest total amount of work and (like the outgoing style) the smallest maximum imbalance between threads (see Table 2). Furthermore, in contrast to the edge-parallel algorithm, it does not matter whether we binary search in $N(v)$ or $N(w)$ as neither is accessed by adjacent threads.

4.2 Wedge-Parallel

Compared to the vertex- and edge-parallel algorithms, the wedge-parallel algorithm excels at workload balance. As shown in Fig. 2, for the outgoing style, the number of wedges per thread ranges from 0 to $\binom{d_{\max}}{2}$ when parallelizing by vertex (Fig. 2a), 0 to $d_{\max} - 1$ when parallelizing by edge (Fig. 2b), and 1 when parallelizing per wedge (Fig. 2c). Moreover, among the outgoing variants, the wedge-parallel algorithm has the best spatial locality. Adjacent threads in the vertex- and edge-parallel algorithms binary search in different neighborhoods, and thus access unrelated (and therefore likely uncached) data. Instead, in the outgoing wedge-parallel algorithm, a thread shares its search neighborhood with, on average, $\frac{d_{\text{avg}}-1}{2}$ adjacent threads.

5 Optimizations

5.1 Graph Reordering

Like other triangle counting works [14, 18, 24], we first perform a *reordering* of the vertices by their (undirected) degree. This optimization benefits triangle counting because it (1) increases the median degree, improving locality, (2) flattens the degree curve, improving load-balancing, and (3) reduces outliers, reducing the number of wedges. We clearly observe (2) and (3) in Fig. 5. For our algorithm, removing high-degree outliers is the most important aspect because the number

Table 2. Theoretical properties of parallelization strategies.

	Vertex-parallel			Edge-parallel			Wedge-Parallel	
	Outgoing	Arrow	Mixed	Outgoing	Arrow	Mixed	Outgoing	Arrow
Total work	Medium[a]	High[b]	Low[c]	Medium[a]	High[b]	Low[c]	Medium[a]	High[b]
Extra data	×	×	×	×	×	×	n	$m+n$
Extra work to retrieve the vertex/edge/wedge	×	×	×	$\log(n)$	$\log(n)$	$\log(n)$	$\log(n)$	$\log(n)+\log(d(v))$
Maximum imbalance[d]	$\binom{d_{\max}}{2}$	${d_{\max}}^2$	$\binom{d_{\max}}{2}$	$d_{\max}$	$d_{\max}$	$d_{\max}$	0	0
Num. of adjacent threads searching in the same neighborhood	0	0	0	0	$d(v)$	0 or $d(v)$	$d(v)_{w+}$[e]	$d(v) \cdot d(w)$[e]
Num. of adjacent threads iterating over the same neighborhood	0	0	0	$d(v)$	0	$d(v)$ or 0	$\frac{\binom{d(v)}{2}}{\text{spread}}$	$\frac{d(w)}{\text{spread}}$

[a] $\sum_{v \in V} \sum_{w \in N(v)} d(v)_{w+}$
[b] $\sum_{v \in V} \sum_{w \in N(v)} d(w)$
[c] $\sum_{v \in V} \sum_{w \in N(v)} \min(d(v)_{w+}, d(w))$
[d] In the number of binary searches (i.e., wedges) between adjacent threads.
[e] If spread $= 1$, or spread > 1 (see Sect. 5.2) and threads cooperate (see Sect. 5.3).

of wedges per vertex is quadratic in its degree. For instance, for the Wikipedia graph (WKL), reordering reduces the maximum degree from approximately 10^6 to 10^3, resulting in 94% fewer wedges (overall).

Although degree-based reordering is the most popular strategy (e.g. [14]), other orderings prioritizing different metrics have been proposed. For example, Trust [18] uses a degree-based ordering combined with heuristics to minimize the number of hash-table collisions, while HTC [24] reorders using the integration of density and degree (following Han et al. [9]), aiming to improve locality.

5.2 Spreading

To amortize the extra work introduced by the binary search to find the base vertex v, we can increase the work (i.e., number of wedges) per thread. We name this amount the *spread*. Increasing the spread is beneficial because finding the next wedge $(v_{i+1}, w_{i+1}, u_{i+1})$ from a known wedge (v_i, w_i, u_i) is (compared to two binary searches) relatively cheap. We either increment u_i (in $N(v)$), increment w_i (in $N(v)$), or increment v and reset the w_i and u_i indices to 0 and 1. Figure 2d shows the iteration pattern for a spread of 2. Note that this optimization increases the workload per thread, but does not alter the workload balance; each thread still evaluates a constant number of wedges.

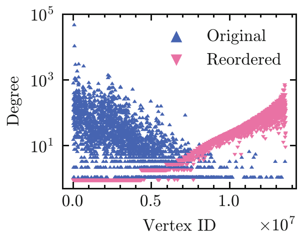


Fig. 5. (Out)degree distribution of the WKL graph.

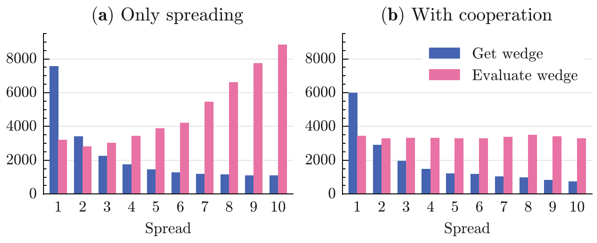


Fig. 6. Average number of cycles (y-axis) required to retrieve and evaluate a wedge for WKL.

5.3 Cooperation

An unfortunate consequence of spreading is that a thread with starting wedge (v, w, u) will share fewer vertices with adjacent threads, i.e., we trade spatial locality for temporal locality, which means we have conflicting goals when it comes to the spread: we both want to increase it (for fewer memory loads) and decrease it (for more cached and/or coalesced memory loads). Our solution is to store the wedges in shared memory and, when all wedges are stored, iterate over them as if the spread were 1, e.g., thread 0 stores its wedges at index $0, 1, ..., (spread - 1)$ but reads at $0, blockSize, ..., blockSize \cdot (spread - 1)$. This ensures that, in the same timestep, the wedges to be evaluated are in the original order $(0, 1, 2, ...)$. We refer to this optimization as *cooperation*. The benefit of cooperation can be seen in Fig. 6a versus Fig. 6b.

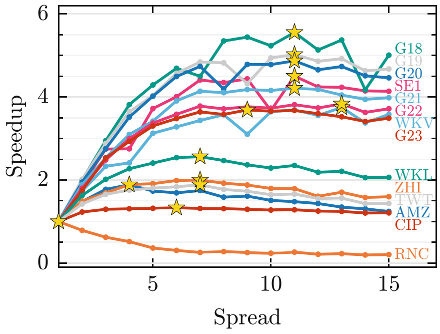


Fig. 7. Speedup for increasing spreads (with cooperation). The best speedup for each graph is marked with a golden star.

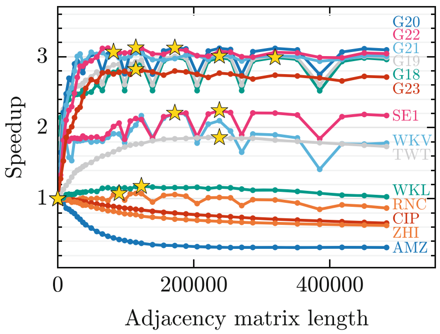


Fig. 8. Speedup for increasing adjacency matrix lengths. The best speedup for each graph is marked with a golden star.

Figure 7 shows the speedup when increasing the spread for all graphs listed in Table 3. For most graphs, we observe a steep rise in performance with the spread until ~7, after which the impact stagnates.

5.4 Adjacency Matrix

To improve the intersection performance, we store some neighborhoods in an *adjacency matrix* (bitmap). Although this is practically unfeasible for the full graph, fortunately, because we store undirected edges as directed edges, i.e., $v < w$ is true for every edge (v, w), we can exclusively store the neighborhoods of the last x vertices, i.e., the vertices from ID $n - x$ to $n - 1$. This is possible because $d(v) < x$ is guaranteed if $v \geq n - x$. Conceptually, we only store a (bottom-right) submatrix of the (triangular) adjacency matrix. Coincidentally, this is also the region where the densest vertices are. Since we sort the vertices by their ascending degree and we work on undirected graphs, $d(v)_{in} = d(v)_{out}$ holds, meaning that the first vertex to be included in the adjacency matrix (vertex $n - 1$) is by definition the most popular destination vertex.

Figure 8 shows the speedup when increasing adjacency matrix length x. The data indicates that an adjacency length between 25000 and 300000 is preferable for most graphs, although the optimal length – and its impact – depends on the degree distribution of the graph. As expected, we observe the highest speedups for graphs with power-law degree distributions, e.g., the Graph500 graphs.

Not all graphs benefit from the adjacency matrix optimization, which could be due to data duplication (of the CSR graph and A), a too balanced distribution, a low maximum degree, or a combination thereof.

6 Evaluation

In this section, we compare WeTriC against the state-of-the-art, examine the impacts of our optimizations, and investigate the end-to-end performance.

Table 3. Graphs statistics (after preprocessing).

Graph	Abbr.	n	m	#wedges	d_{avg}	d_{med}	d_{max}	#triangles
roadNet-CA	RNC	1,965,206	2,766,607	$1.2 \cdot 10^6$	1.4	2	4	$1.2 \cdot 10^5$
Amazon0302	AMZ	262,111	899,792	$1.4 \cdot 10^6$	3.4	4	11	$7.2 \cdot 10^5$
wiki-vote	WKV	7,115	100,762	$1.7 \cdot 10^6$	14.2	4	74	$6.1 \cdot 10^5$
soc-epinions1	SE1	75,879	405,740	$5.7 \cdot 10^6$	5.3	1	121	$1.6 \cdot 10^6$
cit-patents	CIP	3,774,768	16,518,947	$5.0 \cdot 10^7$	4.4	4	77	$7.5 \cdot 10^6$
graph500_scale18[a]	G18	174,147	3,800,348	$2.0 \cdot 10^8$	21.8	5	432	$8.2 \cdot 10^7$
graph500_scale19[a]	G19	335,318	7,729,675	$5.0 \cdot 10^8$	23.0	5	560	$1.9 \cdot 10^8$
zhishi	ZHI	7,825,669	62,246,014	$7.2 \cdot 10^8$	8.0	4	377	$1.1 \cdot 10^8$
graph500_scale20[a]	G20	645,820	15,680,861	$1.2 \cdot 10^9$	24.3	5	715	$4.2 \cdot 10^8$
graph500_scale21[a]	G21	1,243,072	31,731,650	$3.0 \cdot 10^9$	25.5	5	885	$9.4 \cdot 10^8$
graph500_scale22[a]	G22	2,393,285	64,097,004	$7.2 \cdot 10^9$	26.8	5	1,107	$2.1 \cdot 10^9$
graph500_scale23[a]	G23	4,606,314	129,250,705	$1.8 \cdot 10^{10}$	28.1	4	1,439	$4.5 \cdot 10^9$
wikipedia_link_en	WKL	13,593,032	334,591,525	$2.0 \cdot 10^{10}$	24.6	4	1,156	$1.4 \cdot 10^{10}$
twitter (WWW)[b]	TWT	41,652,230	1,202,513,046	$1.5 \cdot 10^{11}$	28.9	10	4,102	$3.5 \cdot 10^{10}$

[a] a=0.57,b=0.19,c=0.19,d=0.05,edge factor=16. [b] a.k.a. twitter7 in SNAP [13].

6.1 Evaluation Setup

We evaluate WeTriC on 14 graphs, ranging from 10^6 to 10^{11} wedges, as listed in Table 3. We selected 8 real graphs (RNC, AMZ, WKV, SE1, CIP from SNAP [13] and ZHI, WKL, and TWT from KONECT [12]), and 6 synthetic sparse graphs (G18-G23 are generated by the Graph500 [8] generator).

We compiled our implementation (~1000 LOC) with CUDA 12.3. We performed most experiments on NVIDIA's RTX A4000 on the DAS-6 [5]. Because this GPU does not have enough device memory to contain TWT, all experiments for this graph were instead performed using Nvidia's RTX A6000.[4] For performance comparisons, we only consider the GPU execution time. All reported results are averaged over 10 runs. Standard deviation was consistently less than 1%. We calculate average speedups using the geometric mean. Occupancy data is collected using Nvidia Nsight Compute [16].

[4] Using a modified implementation with 64-bit row pointers and multiple kernel launches to handle large number of (w)edges.

For Trust, we executed 48 blocks with 1024 threads per block, using the best-performing chunk size per graph (determined empirically). For GroupTC, we follow the default parameters: 48 blocks with 256 threads per block. Unfortunately, we were unable to compare WeTriC to HTC [24], which claims an average speedup of 1.4× over Trust, since their implementation is proprietary.

6.2 Comparison with Vertex and Edge-Parallel

Table 4 shows the speedup of our wedge-parallel algorithm over the vertex-parallel and edge-parallel binary search triangle counting algorithms[5]. We achieve an average speedup of 5.63× over the vertex-parallel algorithm and 4.69× over the edge-parallel algorithm. The highest speedups for the latter come from the Graph500 graphs. For most other graphs, the edge-parallel algorithm outperforms the vertex-parallel algorithm. When increasing the Graph500 scale – and thereby the workload imbalance, WeTriC increasingly outperforms both. We also observe similar performance for all pre-balanced graphs. RNC for instance has a maximum degree of 4, meaning that the maximum workload imbalance is $\binom{4}{2} = 6$, and, as a result, all granularities perform similarly.

Table 4. WeTriC versus the state-of-the-art. Execution times are in milliseconds. Speedups (in parentheses) are relative to WeTriC.

	WeTriC	Vertex-parallel	Edge-parallel	Trust [18]	GroupTC [14]
RNC	0.19	0.17 (0.86×)	0.22 (1.15×)	1.33 (6.81×)	0.55 (2.81×)
AMZ	0.10	0.10 (1.01×)	0.11 (1.05×)	0.51 (5.06×)	0.33 (3.24×)
WKV	0.04	0.47 (10.67×)	0.05 (1.07×)	0.30 (6.77×)	0.23 (5.34×)
SE1	0.11	1.08 (9.40×)	0.46 (4.01×)	0.51 (4.41×)	0.4 (3.45×)
CIP	5.63	10.62 (1.89×)	12.52 (2.22×)	8.18 (1.45×)	7.05 (1.25×)
G18	3.19	31.08 (9.73×)	20.24 (6.34×)	10.32 (3.23×)	7.34 (2.30×)
G19	8.76	77.34 (8.83×)	90.55 (10.34×)	27.95 (3.19×)	21.34 (2.44×)
ZHI	40.79	204.67 (5.02×)	81.34 (1.99×)	48.03 (1.18×)	44.33 (1.09×)
G20	23.14	194.32 (8.40×)	323.90 (14.00×)	74.23 (3.21×)	62.76 (2.71×)
G21	67.36	453.86 (6.74×)	934.26 (13.87×)	193.87 (2.88×)	179.10 (2.66×)
G22	173.88	1556.38 (8.95×)	2875.25 (16.54×)	513.43 (2.95×)	489.25 (2.81×)
G23	446.20	5145.86 (11.53×)	8074.26 (18.10×)	1279.12 (2.87×)	1294.06 (2.9×)
WKL	809.71	8084.13 (9.98×)	2299.04 (2.84×)	612.76 (0.76×)	834.08 (1.03×)
TWT	2611.38	20108.23 (7.70×)	25921.28 (9.93×)	5819.69 (2.23×)	4388.28 (1.68×)

[5] With: the same preprocessing, all applicable optimizations from Sect. 5, and the best-performing wedge style from Sect. 4.

Table 5. Indicators of workload balance for different granularities (average ± std).

Granularity	Vertex-parallel			Edge-parallel			Wedge-Parallel	
Style	Outgoing	Arrow	Mixed	Outgoing	Arrow	Mixed	Outgoing	Arrow
Active warps (average, %)	43 ± 18	48 ± 16	41 ± 19	80 ± 7	86 ± 3	77 ± 8	81 ± 15	83 ± 14
Active threads per warp (average)	18 ± 6	14 ± 4	13 ± 5	21 ± 3	17 ± 3	15 ± 3	29 ± 3	29 ± 2

Table 5 shows the measured average number of active *threads* in a warp (max 32) and the average number of active *warps* compared to the theoretical maximum (a.k.a. *achieved occupancy*). We observe a clear increasing trend in the average active threads with finer granularity. Thus, the amount of (absolute time- and instruction-wise) thread divergence and early termination decreases with the workload imbalance. The number of active warps also increases with the granularity, but remains the same for the edge- and wedge-parallel algorithms. Although inter-warp balance is not as important as intra-warp balance (as warps are not executed in lockstep), WeTriC is not able to consistently improve it.

6.3 Comparison with State-of-the-Art

The performance of WeTriC compared to Trust and GroupTC can be seen in Table 4. In comparison to Trust, we achieve a minimum speedup of 0.76×, a maximum speedup of 6.81×, and an average speedup of 2.86×.

Although WeTriC generally outperforms Trust, it is hard to determine why, simply because Trust, a vectorized hashing-based algorithm, is different on a fundamental level. Besides this, the performance is also highly dependent on the graph distribution. For instance, G23 and WKL have similar graph properties (Table 3) but yield wildly different speedups (2.87 vs. 0.76). This is likely because WKL's distribution is flatter, thereby lowering the potential of the adjacency matrix optimization (see Fig. 9), and of the finer granularity (observe the relatively lower speedups in Table 4 versus the edge- and vertex-parallel algorithms compared to G23). In the interest of fairness, we repeated the experiments on Nvidia's A5000, A6000, A100, and A2 GPU, yielding average speedups (excluding TWT) of 4.15×, 4.95×, 10.27×, and 3.24×, respectively.

Compared to GroupTC, WeTriC achieves a minimum speedup of 1.03×, a maximum speedup of 5.34×, and an average speedup of 2.32×. Unlike Trust, the average speedups on different GPUs remain consistent: 2.66×, 2.67×, 2.80×, 1.98×, likely because both algorithms, unlike Trust, are binary-search-based.

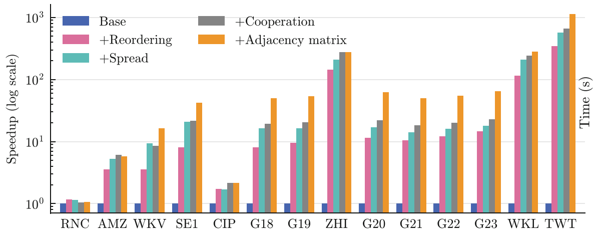


Fig. 9. Performance impact of optimizations (cumulative). Higher is better.

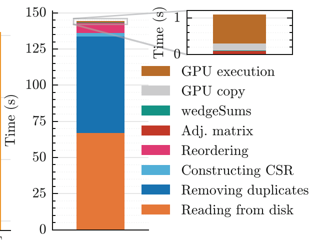


Fig. 10. End-to-end performance for WKL.

6.4 Optimizations Impact

Figure 9 shows the cumulative performance impact of our optimizations. Reordering, spreading, cooperation, and adjacency matrix yield an average speedup of 12.3×, 1.5×, 1.2×, and 1.8× over the previous version, respectively. Although all graphs massively benefit from reordering, ZHI, WKL, and TWT benefit the most with a speedup of 144×, 114×, and 346×, respectively, most likely due to the high reduction in the number of wedges (99.4%, 94%, and 99.7%).

SE1 and WKV highly benefit from the spread, but are hampered by cooperation. Most other graphs benefit from both. Overall, we observe very similar speedups for the Graph500 graphs. In addition, note that the optimal spread (see Fig. 7) and adjacency matrix length (see Fig. 8) do not increase with the scale. Thus, the speedup of our optimizations likely persists at larger scales.

6.5 End-To-End Performance

Figure 10 shows the execution time for all steps of our algorithm. Although the preprocessing time overshadows the execution time, note that this part is not our focus, and thus largely unparallized and unoptimized. Furthermore, an important observation is that some of the preprocessing time is compensated with algorithmic speedups. For example, although reordering WKL takes 7 seconds, counting the triangles of an *unordered* WKL takes 228.7 (instead of ~2) seconds.

7 Conclusion

GPU-based triangle counting algorithms are promising for processing massive sparse graphs with high performance. However, state-of-the-art algorithms suffer from large workload imbalance. To alleviate this problem, we propose WeTRiC, a novel wedge-parallel triangle counting algorithm for GPUs, that uses a lightweight data structure to resolve workload imbalance. We analyze the theoretical properties and advantages of WeTriC compared to state-of-the-art solutions, and apply several optimizations to further improve its efficiency. Our extensive empirical analysis shows WeTriC consistently outperforms the state-of-the-art – by, on average, 2.86× (Trust [18]) and 2.32× (GroupTC [14]).

Disclosure of Interests. The authors have no competing interests to declare.

Acknowledgments and Artifact Availability. David Bader is supported in part by U.S. National Science Foundation grants CCF-2109988 and OAC-2402560. The artifact is available in the Zenodo repository [21].

References

1. Almasri, M., Vasudeva, N., Nagi, R., Xiong, J., Hwu, W.M.: HyKernel: a hybrid selection of one/two-phase kernels for triangle counting on GPUs. In: IEEE HPEC (2021)
2. Angeletti, M., Bonny, J.M., Koko, J.: Parallel euclidean distance matrix computation on big datasets (2019), hal-02047514
3. Bader, D.A.: Fast triangle counting. In: IEEE HPEC (2023)
4. Bader, D.A., et al.: Triangle counting through cover-edges. In: IEEE HPEC (2023)
5. Bal, H., et al.: A medium-scale distributed system for computer science research: Infrastructure for the long term. Computer **49**(5), 54–63 (2016)
6. Bisson, M., Fatica, M.: High performance exact triangle counting on GPUs. IEEE Trans. Parallel Distrib. Syst. **28**(12), 3501–3510 (2017)
7. Dong, Y., et al.: Link prediction and recommendation across heterogeneous social networks. In: IEEE ICDM (2012)
8. Graph 500 Steering Committee: The Graph500 Benchmark. https://www.graph500.org (2010)
9. Han, S., Zou, L., Yu, J.X.: Speeding up set intersections in graph algorithms using SIMD instructions. In: ACM SIGMOD (2018)
10. Hu, L., Guan, N., Zou, L.: Triangle counting on gpu using fine-grained task distribution. In: IEEE ICDEW (2019)
11. Hu, Y., Liu, H., Huang, H.H.: TriCore: parallel triangle counting on GPUs. In: ACM SC (2018)
12. Kunegis, J.: Konect: the koblenz network collection. In: ACM WWW (2013)
13. Leskovec, J., Krevl, A.: SNAP datasets: stanford large network dataset collection. http://snap.stanford.edu/data (2014)
14. Li, J., Xu, Z., Pham, M., Tu, Y., Zhou, Q.: A comparative study of intersection-based triangle counting algorithms on GPUs. In: IEEE IPDPS (2024)
15. Mailthody, V.S., et al.: Collaborative (CPU + GPU) algorithms for triangle counting and truss decomposition. In: IEEE HPEC (2018)
16. Nvidia: Nsight Compute (2025), https://docs.nvidia.com/nsight-compute/
17. Pandey, S., Li, X.S., Buluc, A., Xu, J., Liu, H.: H-INDEX: hash-indexing for parallel triangle counting on GPUs. In: IEEE HPEC (2019)
18. Pandey, S., et al.: Trust: Triangle counting reloaded on gpus. IEEE Trans. Parallel Distrib. Syst. **32**(11), 2646–2660 (2021)
19. Pearson, C., et al.: Update on triangle counting on GPU. In: IEEE HPEC (2019)

20. Schank, T.: Algorithmic aspects of triangle-based network analysis. Ph.D. thesis, University of Karlsruhe (2007)
21. Spaan, J., Chen, K.H., Bader, D.A., Varbanescu, A.L.: Artifact of the paper: wedge-parallel triangle counting for GPUs (2025). https://doi.org/10.5281/zenodo.15611508
22. Verstraaten, M.: Analysis and prediction of GPU graph algorithm performance. Ph.D. thesis, University of Amsterdam (2022)
23. Wang, J., Cheng, J.: Truss decomposition in massive networks. Proc. VLDB Endow. **5**(9), 812–823 (2012)
24. Zeng, L., Yang, K., Cai, H., Zhou, J., Zhao, R., Chen, X.: HTC: hybrid vertex-parallel and edge-parallel triangle counting. In: IEEE HPEC (2022)