2023 Design, Automation & Test in Europe Conference (DATE 2023)

CLAP: Locality Aware and Parallel Triangle Counting with Content Addressable Memory

Tianyu Fu*†, Chiyue Wei*†, Zhenhua Zhu†, Shang Yang†, Zhongming Yu†, Guohao Dai‡, Huazhong Yang†, Yu Wang†
†Dept. of EE, BNRist, Tsinghua University ‡Shanghai Jiao Tong University
daiguohao@sjtu.edu.cn, yu-wang@tsinghua.edu.cn

Abstract—**Triangle counting (TC) is one of the most fundamental graph analysis tools with a wide range of applications. Modern triangle counting algorithms traverse the graph and perform set intersections of neighbor sets to find triangles. However, existing triangle counting approaches suffer from the heavy off-chip memory access and set intersection overhead. Thus, we propose CLAP, the first content addressable memory (CAM) based triangle counting architecture with the software and hardware co-optimizations.**

To reduce off-chip memory access and the number of set intersections, we propose the first force-based node index reorder method. It simultaneously optimizes both data locality and the computation amount. Compared with random node indices, the reorder method reduces the off-chip memory access and the set intersections by 61% and 64%, respectively, while providing 2.19× end-to-end speedup. To improve the set intersection parallelism, we propose the first CAM-based triangle counting architecture under chip area constraints. We enable the high parallel set intersection by translating it into content search on CAM with full parallelism. Thus, the time complexity of the set intersection reduces from $O(m+n)$ or $O(n \log m)$ to $O(n)$. Extensive experiments on real-world graphs show that CLAP achieves 39×, 27×, and 78× speedup over state-of-the-art CPU, GPU, and processing-in-memory baselines, respectively. The software code is available at: https://github.com/thu-nics/CLAP-triangle-counting.

Index Terms—**triangle counting, content addressable memory**

I. Introduction

Triangle counting (TC) is the problem of searching and counting triangles in a given graph. Triangle counting is one of the most fundamental tools for graph analysis. It has a wide range of applications, including social network [1], recommendation system [2], biochemistry [3] and so on. As shown in Figure 2(b), modern triangle counting algorithms use for-loops to traverse the graph, then use set intersections to find the common neighbors of two nodes to find the triangle.

However, existing triangle counting approaches suffer from the following two challenges. Firstly, triangle counting involves heavy off-chip memory access overhead. Previous work has shown that the total data transfer volume from the off-chip memory is 106× larger than the original graph size [4] and takes up 70% of the total runtime on average [5]. It is mainly caused by the massive neighbor set data requests and poor locality of the graph data. We perform triangle counting on the same graphs in [4] and find that the total neighbor set data request volume is 135× larger than the original graph size. To make matters worse, the complexity of the graph means that the topologically adjacent nodes may not have spatial locality in memory. Previous work [6] has shown that graph processing suffers from 30% to 90% cache miss ratio, which poses even more overhead on off-memory data access.

*: Both authors contributed equally to this work.

Secondly, triangle counting involves heavy set intersection overhead. Previous work [7] reports that the set intersection takes 94% of the total computation time. As shown in Figure 2(b), it is the dominant operation of the inner loop. On CPUs and GPUs, the merge-list-based [8] and binary-search-based set intersection [9] methods are commonly used. Given m-element-set A and n-element-set B, the methods take $O(m+n)$ or and $O(n \log m)$ time complexity to compute $A \cap B$, respectively. However, these two methods hardly utilize the Instruction-level Parallelism (ILP) of modern hardware because of the control dependency of the set intersection. For example, the binary-search-based set intersection on CPUs needs the result of the previous comparison to determine the search range of the next, as shown in Figure 4, thus fully sequential without ILP.

Luckily, the development of the Content Addressable Memory (CAM) sheds light on a preferable architecture for TC. CAM is a special kind of memory that can search the address of the stored content in one clock cycle. It is widely used in network routers and data mining [10]. This enlightens us to harvest the high computing parallelism of CAM to speed up the set intersections in the TC problem.

In this paper, we propose CLAP, the first CAM-based Locality Aware and Parallel architecture for the triangle counting problem. CLAP uses software–hardware co-optimizations to improve the data locality and enables high set intersection parallelism of the triangle counting problem. The main contributions are summarized as follows.

- To reduce the off-chip memory access and the number of set intersections, we propose the force-based node index reorder method. The proposed reorder method simultaneously optimizes both data locality and the number of set intersections. The reorder method reduces the off-chip memory access and the set intersection by 61% and 64%, respectively, while providing 2.19× end-to-end speedup.
- To improve the parallelism of set intersections, we propose the first CAM-based TC architecture. The CAM-based processing elements (PEs) reduce the time complexity of set intersections from $O(m+n)$ or $O(n \log m)$ to $O(n)$.
- Experimental results show that, compared with the state-of-the-art CPU, GPU, and processing-in-memory (PIM) based TC designs, CLAP can achieve 39×, 27×, and 78× speedup, respectively.

The following sections are organized as follows. Section II discusses the background and related work on graph representation, TC and CAM. Section III introduces the force-based reorder method and how it optimizes multiple objectives. Section IV introduces CLAP's high parallel CAM-based archi-

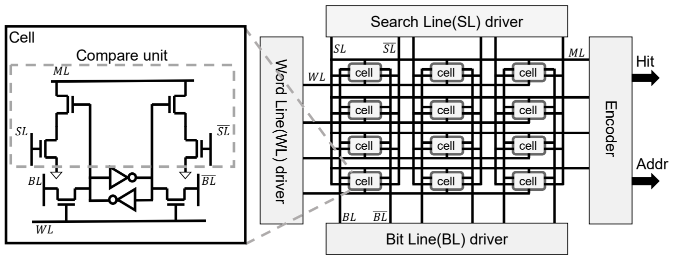
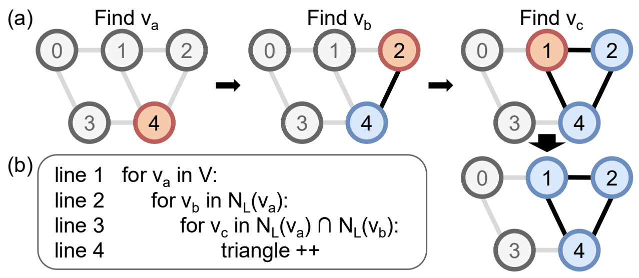


Fig. 1. The circuit structure of CAM. It performs parallel content search and outputs the matched address.



```
line 1   for $v_a$ in V:
line 2       for $v_b$ in $N_L(v_a)$:
line 3           for $v_c$ in $N_L(v_a) \cap N_L(v_b)$:
line 4               triangle ++
```

Fig. 2. (a) TC problem and (b) TC algorithm

tecture. Section V demonstrates extensive experiments and the paper concludes with Section VI.

II. Background and Related Work

A. Graph Representation

Given a graph $G = (V, E)$, V and E represent the set of all nodes and edges in the graph. $N(v)$ is the neighbor set of node v and $d(v) = |N(v)|$ is the degree of it. A unique integer $i(v) \in [0, |V|)$ is assigned to each node as its index. According to the index, $N(v)$ can be split into two subsets $N_L(v) = \{u \in N(v) | i(u) < i(v)\}$ and $N_R(v) = \{u \in N(v) | i(u) \geq i(v)\}$. $d_L(v)$ and $d_R(v)$ are defined as $|N_L(v)|$ and $|N_R(v)|$, respectively. The graph data is commonly represented as the Compressed Sparse Row (CSR) format [4], which sequentially stores the adjacent list according to the node indices.

B. Triangle Counting

Figure 2 shows the execution flow of TC. It traverses all the nodes of the graph. For each node v_a, it traverses all the v_b in $N_L(v_a)$ and performs the set intersection: each $v_c \in N_L(v_a) \cap N_L(v_b)$ corresponds to a triangle consists of v_a, v_b, and v_c. Note that $N_L(v)$ instead of $N(v)$ is used to avoid double counting [4]. There has been extensive work studying the efficient execution of the TC problem with different architectures. The CPU approach GBBS [11] reorders the node indices to reduce the number of set intersections. The GPU approach ColGPU [8] utilizes GPU to perform high parallel TC. The PIM approach TCIM [12] implements TC on MRAM to eliminate heavy data transfer. The near-memory-computing (NMC) approach DIMMining [4] designs the systolic merge array to accelerate set operations. Despite various optimizations, little work focuses on optimizing both memory access and the number of set intersections.

C. Content Addressable Memory

Content addressable memory (CAM) [13] is a specialized memory. Unlike regular memory that can only access content with address, CAM can also use content to search the data address in one cycle. The parallel data searchability of CAM enables a wide range of latency-sensitive applications [10]. The circuit structure of CAM is shown in Figure 1. CAM consists of several entries. Each entry of the CAM consists of several bit cells, which store 1-bit data as normal memory does. Each bit cell contains a compare unit that activates in parallel. When the content search request is sent to the search lines, every bit cell simultaneously compares itself to the corresponding search line in one cycle and returns the corresponding entry address if matched.

III. Force-Based Reorder

A. Motivations and Objectives of Reorder

Reorder is a commonly used preprocessing scheme for graph algorithms [9], [14]. For TC, there exist two main lines of work that target different objectives.

One objective is to use community-based reorder to optimize data locality. The graph community refers to a group of nodes that are densely connected internally. As discussed in Section II-A, the node indices determine which nodes' adjacency lists are stored together, thus influencing the data locality for graph traverse. A typical insight is to assign continuous indices to nodes from the same community, since a node is highly likely to access its adjacent nodes from the same community when traversing the graph. Typical example [14] identifies communities of different sizes and assigns continuous indices for nodes in them.

The other objective is to use degree-based reorder to optimize the number of set intersections. According to Figure 2(b), let the total length of all sets (i.e., $N_L(v_a)$ and $N_L(v_b)$) used for intersection be $|X|$, we can derive Equation 1. Note that $v_b \in N_L(v_a) \Leftrightarrow v_a \in N_R(v_b)$ by definition.

$$\begin{aligned}
|X| &= \sum_{v_a \in V} \sum_{v_b \in N_L(v_a)} d_L(v_a) + d_L(v_b) \\
&= \sum_{v_a \in V} \sum_{v_b \in N_L(v_a)} d_L(v_a) + \sum_{v_b \in V} \sum_{v_a \in N_R(v_b)} d_L(v_b) \\
&= \sum_{v_a \in V} d_L(v_a) d_L(v_a) + \sum_{v_b \in V} d_R(v_b) d_L(v_b) \\
&= \sum_{v \in V} d_L(v) d(v)
\end{aligned} \tag{1}$$

Equation 1 reveals that each neighbor set $N_L(v)$ is calculated $d(v)$ times in TC. According to the definition, given a node v, the smaller index $i(v)$ is, the smaller $d_L(v)$ of the node will be. Though $\sum_{v \in V} d_L(v) = |E|$ is constant, we can adjust the distribution of node indices so that the weighed summation in $|X|$ can be lessened. The intuitive is that nodes with higher degree d should be assigned with smaller node indices. To minimize the set intersection, previous work [9] proves that the optimal order is to index the node by the descending rank of

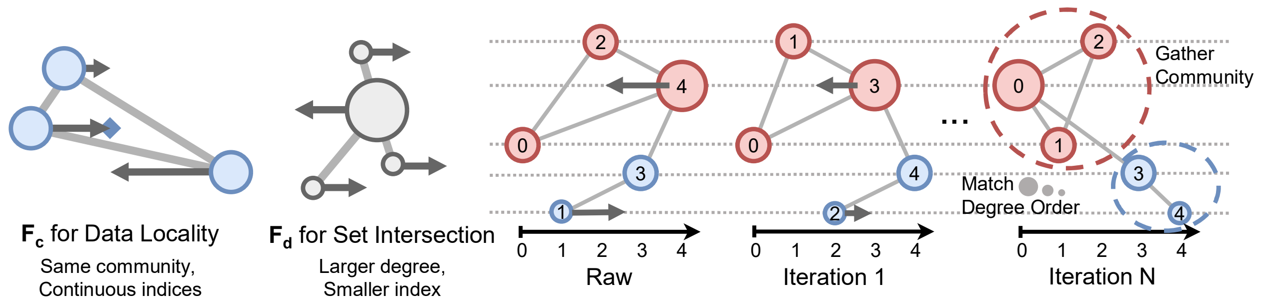


Fig. 3. The force-based reorder. The x-coordinates of nodes represent their indices. (1) Degree-based force and community-based force drag the nodes horizontally to optimize the node indices for different objectives. (2) Each node is horizontally dragged with two forces to be reordered iteratively. Forces on two typical nodes are shown. The reordered graph better confirms both objectives than the raw graph.

the node degree. Note that for the nodes with the same degree, their ranks are random. So we use $R_+(v)$ and $R_-(v)$ to denote the largest and smallest degree rank of v, which is the range of its optimal index.

However, to the best of our knowledge, no previous work simultaneously optimizes both data locality and the number of set intersections. The single-objective optimization causes the shortcoming of the other. For example, though [14] achieves 38% less cache miss ratio than [9], it requires 2.57× set intersections, resulting in 49% more off-chip memory access.

B. Force Design and Reorder Scheme

We propose the force-based reorder method. It abstracts different objectives as different forces dragging each node towards its optimal position. As shown in Figure 3, the nodes of the raw graph are shown on a plane, where the x-coordinate indicates the node's index. The community-based force and the degree-based force are defined to optimize data locality and the number of set intersections respectively. When reordering, the nodes are dragged by these horizontal forces to move and reorder iteratively.

The design of forces follows the same principles discussed in Section III-A. To optimize locality, we design the community-based force F_c. Each node of the community C has a force F_c pointing towards the center of the community, dragging nodes to gather for continuous indices. The magnitude of F_c is proportional to the distance between the node and the community center, as shown in Equation 2.

$$F_c(v, C) = i(v) - \frac{1}{|C|} \sum_{v_c \in C} i(v_c) \tag{2}$$

To optimize the number of set intersections, we design the degree-based force F_d. Each node v has the force F_d dragging it towards the margin of its optimal index range $[R_-(v), R_+(v)]$, as discussed in Section III-A.

$$F_d(v) = \begin{cases} R_+(v) - i(v), & \text{if } i(v) > R_+(v) \\ R_-(v) - i(v), & \text{if } i(v) < R_-(v) \\ 0, & \text{otherwise} \end{cases} \tag{3}$$

When reordering, all the nodes are dragged horizontally by $\Delta i = \alpha F_c + \beta F_d$ at each iteration, where hyperparameters $\alpha, \beta \in [0, 1]$. Since $i' = i + \Delta i$ may not an integer, the new indices i' is further quantized by the index rank: for $i'_0 \le i'_1 \le \dots$, let $[i'_0, i'_1, \dots] \to [0, 1, \dots]$. The F_c and F_d are updated at each iteration. Figure 3 shows how nodes are gradually reordered by two forces, so that the nodes in the same community tend to have continuous indices, while the large-degree nodes tend to have smaller indices. The force-based reorder achieves both better locality and a minimum number of set intersections.

TABLE I
THE DRAM ACCESS AND SET INTERSECTION DATA VOLUME COMPARISON

	DRAM access(B)			Number of set intersections		
	[14]	[9]	Ours	[14]	[9]	Ours
MC	2.1E+8	2.2E+8	**2.1E+8**	8.0E+7	**5.5E+7**	**5.5E+7**
PT	**1.1E+9**	2.3E+9	1.7E+9	3.4E+8	**2.0E+8**	**2.0E+8**
YT	5.1E+9	3.7E+8	**3.5E+8**	1.1E+9	**7.8E+7**	8.5E+7

Hyperparameters α and β allow the balance of both objectives. According to Equation 1, a dataset is sensitive to degree-based order when the degrees are unevenly distributed. We use the normalized standard deviation of degree σ_n to choose the hyperparameters, where $\sigma_n = \sqrt{\sum_{v_i \in V} (d(v) - \bar{d})^2 / |V|} / \bar{d}$.

The runtime complexity of the force-based reorder with I iteration is $O(I|V|\log|V| + |E|)$, where $I = 30$ is empirically good enough and used throughout this paper. Compared with the $O(E^{3/2})$ complexity of TC, the complexity of the reorder preprocessing is acceptable.

To exploit the benefit of our force-based reorder, we compare the number of set intersections, and the DRAM access volume with those of the single-objective reorder methods. As shown in Table I, the force-based reorder achieves comparable performance on the respective focus of the two methods, while making up for the shortcomings. The end-to-end performance gain is further analyzed in Section V-C.

IV. CAM-based Architecture Design

A. CAM-based Set Intersection

The set intersection is the key operation for TC. Thus, we propose the CAM-based set intersection to improve its parallelism. Figure 4 shows an example of the binary-search-based intersection used by previous work [8], and our proposed CAM-based intersection. A and B have m and n elements respectively. Both methods search each element of B in A to perform $A \cap B$. The binary search takes $O(\log_2 m)$ time complexity and requires constant data readout from memory. The CAM-based method stores all the elements of A as CAM entries and puts the element of B on CAM's search line. CAM

Algorithm 1: Workflow of CLAP's PE

```
   Input: undirected graph $G(V, E)$, $N$, $M$
   Output: num_triangle
1  $G(V, E) \leftarrow$ reorder($G(V, E)$);
2  num_triangle $\leftarrow$ 0;
3  foreach $(t(v_a), v_b) \in PE.CAM$ do
4      N($v_b$) $\leftarrow$ PE.cache.read_neigh($v_b$)
5      foreach $v_c \in N(v_b)$ do
6          s $\leftarrow$ (t($v_a$),$v_c$);
           // check whether $v_c \in N_L(v_b)$
7          if $s \in PE.CAM$ then
8              num_triangle $\leftarrow$ num_triangle + 1;
9          end
10     end
11 end
```

performs the full parallel search in all elements of A in one clock cycle. Thus, the complexity reduces from $O(n \log m)$ (or $O(m + n)$ for merge-based-intersection [4]) to $O(n)$.

B. Architecture Overview

As shown in Figure 2(b), the TC algorithm includes two main operations: the for-loop based graph traverse and the neighbor set intersection. We design the CAM-based processing element (PE) as its fundamental building block to optimize both operations. Each PE contains K CAM entries. CLAP utilizes both the inter-PE parallelism for graph traverse and the intra-PE parallelism for set intersections.

CLAP's architecture overview is shown in Figure 5. CLAP follows the NMC scheme of previous work [4]. We use M PEs to construct a processing unit (PU). The PU is placed inside each DRAM rank. The memory control unit of the PU sequentially handles the data requests of the PEs and fetches the corresponding data from the DRAM rank. N PUs and DRAM ranks form the whole system.

C. Inter-PE Parallelism for Graph Traverse

The main challenge for inter-PE parallelism is to minimize the interference between different PEs. Previous work [6] shows that using 8-job concurrent graph processing only brings 40% runtime gain. The drawback is mainly because of the interference of different jobs, e.g., cache interference. Thus, we design all the PEs to be independent and operate by themselves to enable fully parallel computation with minimum interference. The main observation behind it is that the nested for-loops starting from different v_a are completely independent, as shown in line1 in Figure 2(b).

Under such observation, the whole TC task is first evenly distributed to each PU based on v_a. To optimize the load balance between different PUs, the loop starting from v_a is assigned to the $\mod(v_a, N)$-th PU. The number of rank N is relatively small, e.g., 16, compared with thousands to millions of nodes of the graph dataset. So this simple workload distribution is empirically balance enough.

For each PU, its DRAM device contains a full copy of the graph data and handles its own v_a by assigning them to its PEs. It begins with neighbor loading. The PU assigns the task to a PE by loading $N_L(v_a)$ to PE's CAM. Each entry of CAM means a $v_b \in N_L(v_a)$. The loading is done until the CAM of this PE cannot store another $N_L(v_a)$. It benefits from continuous DRAM device access. Note that a $\log_2(K)$ bit tag is attached to every v_b to represent the corresponding v_a. PE then operate-by-itself to finish the computation between line2 to line4 in Figure 2(b) as discussed in Section IV-D. Once a PE finishes its current task, it outputs the counting result and requests for neighbor loading again. This enables the dynamic scheduling of PU's tasks. When all the v_a in the PU are assigned, the counting is done and the summation of the count is returned.

In this manner, each PE handles its own v_a without interference. Besides, due to the improvement of locality discussed in Section III, we can use a small cache for each PE while maintaining a low cache miss ratio, as shown in Section V. Thus we enable a larger amount of PEs under area constraint, further improving the inter-PE parallelism.

D. Intra-PE Parallelism for Set Intersections

The main insight for intra-PE parallelism is to translate the set intersection into a parallel search operation in CAM. Its full parallel content search ability reduces the $O(\log n)$ time binary search of v_c in $N(v_a)$ to $O(1)$ time as shown in Line 6 of Algorithm 1.

Figure 5 shows the architecture of each PE and how it works to find the triangles. Note that v_bs, or $N(v_a)$s have been loaded to CAM as discussed in Section IV-C. The PE goes through the ① to ④ step pipeline to check the existence of a potential triangle according to the loaded data. Each step corresponds to an operation in Figure 2(b). The CAM acts as both the memory and the computing unit to enable high parallel set intersections.

In step ①, the CAM acts as a regular memory to sequentially read the content of an entry: the tag of v_a ($t(v_a)$) and one neighbor of it v_b. The `control` unit keeps track of the current entry position. It starts with the first entry and moves forward by one entry for each cycle. In step ②, v_b is passed to the `Fetcher` to get $N_L(v_b)$. The `Fetcher` decodes v_b to the addresses of its neighbor and tries to fetch $N_L(v_b)$ from cache. If cache misses, a data request will be sent to the memory control unit of PU to fetch the data from the DRAM Device in the rank. In step ③, the each fetched $v_c \in N_L(v_b)$ is concatenated with the tag $t(v_a)$ from step ① to form the search word s in Algorithm 1. s is passed to the search lines of CAM

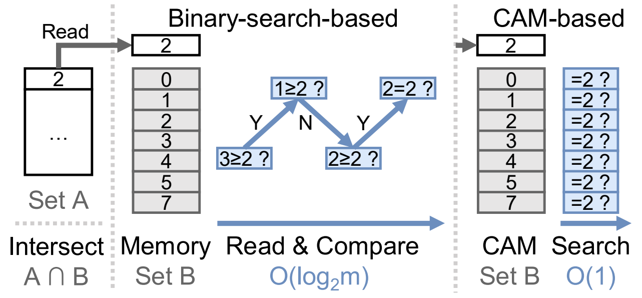


Fig. 4. The binary-search-based and CAM-based set intersection

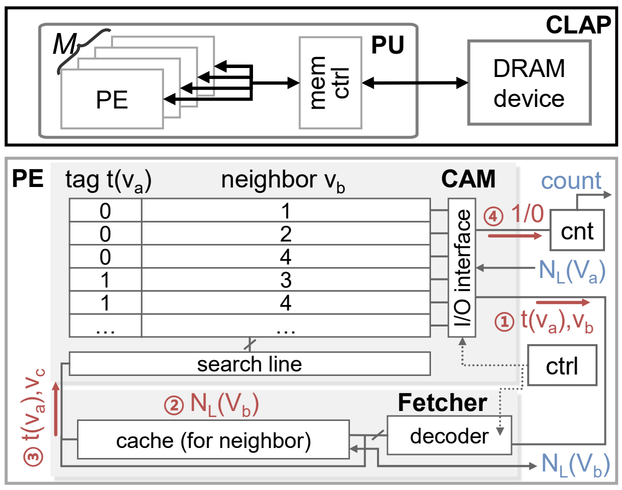


Fig. 5. The CLAP architecture Overview and the main PE pipeline. CLAP places a processing unit (PU) in each DRAM Rank. Each PU consists of M processing elements (PEs). PE goes through the pipeline of: ① get node v_b; ② fetch neighbor $N_L(v_b)$; ③ for each $v_c \in N_L(v_b)$, search v_c in $N_L(v_a)$(with tag $t(v_a)$); ④ accumulate count by 1 if matches.

and is compared with every entry in the CAM within one clock cycle. Note that CAM's parallel search ability reduces the $O(\log n)$ time binary search of v_c in $N(v_a)$ to $O(1)$ time. Finally in step ④, if v_c from step ③ matches any elements of $N(v_a)$, a valid signal is generated to add the counter by one. Step ③ and ④ continues for $n = |N_L(v_b)|$ cycles until all v_c are processed. Under such design, we complete the intersection $N_L(v_a) \cap N_L(v_b)$ and the time complexity is reduced from $O(m+n)$ (the merge-based intersection) or $O(n \log m)$ to $O(n)$. The next round continues with step ① for the next v_a.

Recall that the mentioned workflow is fully pipelined, and the CAM unit is well-utilized to act as both the memory and the parallel computing unit. This enables the processing-in-CAM scheme with high intra-PE parallelism.

V. Experiments

A. Experimental Setup

1) CLAP setup: Table II shows the configuration of CLAP. We use 15 small PUs for the most nodes, and one large PU for the few ($< 0.1\%$) nodes with $d_L > 512$. We simulate the end-to-end performance with: (1) In-house behavioral level TC simulator to emulate the computation of CLAP and generate the memory access trace. (2) The cache simulator [4] for cache behavior and CACTI [15] in 32nm technology for area and energy. (3) Ramulator [16] for DRAM latency. (4) The 28nm CAM at 400MHz [17] for CAM metrics. (5) Synopsys Design Compiler for the peripheral circuits under TSMC 65nm technology and is scaled to 32nm by [18].

2) Existing TC designs: **GBBS** [11] implements TC on CPUs. We evaluate GBBS on a machine with two Intel(R) Xeon(R) Gold 6226R CPUs with 64 threads. **ColGPU** [8] is a CPU and GPU collaborated TC system that runs on an IBM Minsky machine. **TCIM** [12] implements TC algorithm with a 16 MB MRAM computation array and a single-core CPU. **DIMMining** [4] uses 32 NMC module with 16 DRAM rank to perform TC.

TABLE II
CLAP Configuration

Software Configuration		
Value of σ_n	$\sigma_n > 1.2$	$\sigma_n \leq 1.2$
Value of α / β	0.10 / 0.90	0.99 / 0.01
Hardware Configuration		
DRAM capacity / #Ranks	64GB / 16	
DDR4 configuration	4Gb x8 2400R	
DRAM frequency	1200MHz	
PU Frequency	400MHz	
Type of PU	Small	Large
#PUs / #PEs per PU	15 / 8	1 / 4
#Entries / Tag bits / Neighbor bits	512 / 9 / 32	1024 / 10 / 32
Cache size per PE	16KB	32KB

TABLE III
Information of Graph Datasets

	Abbr.	#Nodes	#Edges	#Triangles
P2P	PP	8.11K	26.0K	2.35K
Astro	AS	18.8K	198K	1.35M
Email-Enron	EE	36.7K	184K	727K
Mico	MC	96.6K	1.08M	12.5M
RoadNet-PA	PA	1.09M	1.54M	67.2K
RoadNet-TX	TX	1.38M	1.92M	8.29K
Patents	PT	3.77M	16.5M	7.52M
Youtube	YT	1.13M	2.99M	3.06M
LiveJournal	LJ	4.00M	34.7M	178M

3) Datasets: We evaluate the performance of CLAP on different real-world datasets: P2P, Astro, LiveJournal, Email-Enron, RoadNet-PA, RoadNet-TX from [19], Mico from [20], Patents from [21], and Youtube from [22]. The datasets' abbreviations and statistics are shown in Table III.

B. End-to-end Performance

We compare CLAP with the above state-of-the-art CPU, GPU, PIM, and NMC TC designs. Table IV shows that our end-to-end performance achieves 39×, 27×, 78×, and 1.4× speedup respectively.

C. Benefit of Force-based Reorder

Our reorder technique optimizes both the off-chip memory access and the number of set intersections. The end-to-end performance of different orders tested on CPU is shown in Table V. The preprocessing time is not considered for all methods. The proposed force-based reorder achieves 1.12× speedup over the best baseline. Compared with random order,

TABLE IV
Comparison of End-to-End Runtime (Seconds)

	GBBS [11]	ColGPU [8]	TCIM [12]	DIMMining [4]	CLAP Ours
PP	4.2E-03	N/A	N/A	1.6E-05	**1.3E-05**
AS	8.0E-03	N/A	N/A	2.6E-04	**1.8E-04**
EE	N/A	2.6E-03	2.1E-02	N/A	**4.2E-04**
MC	2.4E-02	N/A	N/A	1.6E-03	**1.5E-03**
PA	N/A	1.5E-02	4.3E-02	N/A	**2.2E-04**
TX	N/A	1.3E-02	5.3E-02	N/A	**2.7E-04**
PT	3.7E-01	N/A	N/A	1.5E-02	**1.0E-02**
YT	7.5E-02	N/A	9.8E-02	5.2E-03	**2.7E-03**
LJ	7.3E-01	N/A	2.0E+00	6.9E-02	**4.9E-02**
Normed	39.10	27.07	78.27	1.40	1.00

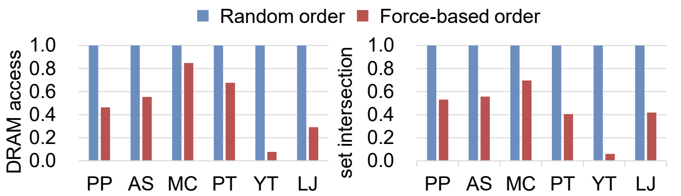


Fig. 6. DRAM access volume and the number of set intersections normalized by random order

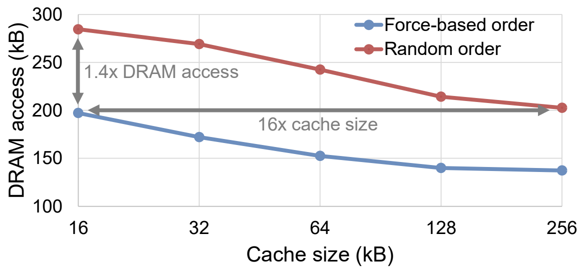


Fig. 7. DRAM access of one PE changes with cache size in different order

force-based order reduces the number of set intersection and DRAM access volume by 61% and 64%, respectively, and the result is shown in Figure 6.

Furthermore, since triangle is the most basic graph substructure, the reordered graph also benefits other subgraph counting algorithms. We use DIMMining's software to test the general benefit of the force-based reorder on CPU. Experiments on the same graphs in Table V show that the force-based reorder achieves $1.89\times$, $1.49\times$, and $1.09\times$ speedup over the random, community-based [14], and degree-based [9] order for four-clique counting, respectively. It also achieves $1.82\times$, $1.49\times$, and $1.06\times$ speedup for five-clique counting, respectively.

D. Area and Power Analysis

Thanks to the locality improvement of the reorder method, we can use a smaller cache for our CAM-based PE to reduce area and energy overhead while maintaining a low DRAM access number. Figure 7 shows that through the force-based reorder, 16kB cache requires less DRAM access than 256kB cache in random order when counting triangles on dataset Astro. The area and power of CLAP are shown in Table VI. CLAP uses 32% less area and 2% less energy than DIMMining.

VI. Conclusion

In this work, we propose CLAP, a Content addressable memory based Locality Aware and Parallel triangle counting architecture. For the software, we propose the force-based reorder method to optimize data locality and the number of set intersections simultaneously. Force-based reorder reduces the amount of off-chip memory access and the number of set intersections by 61% and 64% respectively. For the hardware, we propose the CAM-based architecture to improve the parallelism of the set intersection. It utilizes both the intra-PE parallel search ability of CAM and the inter-PE interference-free parallel design. It achieves O(n) set intersection time complexity and 128 fully parallel PE under area constraint. Experiments show that CLAP's end-to-end performance exceeds state-of-the-art CPU, GPU, and PIM baselines by $39\times$, $27\times$, and $78\times$ respectively.

TABLE V
End-to-end runtime (seconds) comparison of reorder strategies on different datasets.

	Random	Comm-based [14]	Degree-based [9]	Ours
PP	2.22E-3	2.30E-3	1.57E-3	**1.56E-3**
AS	4.44E-2	4.01E-2	2.86E-2	**2.81E-2**
MC	3.47E-1	2.37E-1	2.24E-1	**2.24E-1**
PT	4.14E+0	2.18E+0	3.30E+0	**1.92E+0**
YT	2.82E+0	2.54E+0	3.84E-1	**3.75E-1**
LJ	1.94E+1	1.32E+1	1.11E+1	**9.84E+0**
Normed	2.19	1.69	1.12	1.00

TABLE VI
Area and power analysis of CLAP

	CAM	Fetcher	Control	Total
Area (μm^2)	408421	7339418	556046	8303884
Energy (mW)	440.66	2543.83	319.22	3303.72

VII. Acknowledgement

This work was supported by National Natural Science Foundation of China (No. 62104128, 61832007, U19B2019, U21B2031), Tsinghua University Initiative Scientific Research Program, Beijing National Research Center for Information Science and Technology (BNRist), and Tsinghua EE Xilinx AI Research Fund.

References

[1] C Seshadhri et al. Wedge sampling for computing clustering coefficients and triangle counts on large graphs. *SADM*, 2014.
[2] Charalampos E Tsourakakis et al. Spectral counting of triangles via element-wise sparsification and triangle-based link recommendation. *SNAM*, 2011.
[3] Luca Becchetti et al. Efficient semi-streaming algorithms for local triangle counting in massive graphs. In *SIGKDD*, 2008.
[4] Guohao Dai et al. Dimmining: pruning-efficient and parallel graph mining on near-memory-computing. In *ISCA*, 2022.
[5] Hao Wei et al. Speedup graph processing by graph ordering. In *ICMD*, 2016.
[6] Yu Zhang et al. Cgraph: A correlations-aware approach for efficient concurrent iterative graph processing. In *USENIX ATC*, 2018.
[7] Shuo Han et al. Speeding up set intersections in graph algorithms using simd instructions. In *ICMD*, 2018.
[8] Ketan Date et al. Collaborative (cpu+ gpu) algorithms for triangle counting and truss decomposition on the minsky architecture: Static graph challenge: Subgraph isomorphism. In *HPEC*, 2017.
[9] Lin Hu et al. Triangle counting on gpu using fine-grained task distribution. In *ICDEW*, 2019.
[10] Kostas Pagiamtzis et al. Content-addressable memory (cam) circuits and architectures: A tutorial and survey. *JSSC*, 2006.
[11] Laxman Dhulipala et al. Theoretically efficient parallel graph algorithms can be fast and scalable. *TOPC*, 2021.
[12] Xueyan Wang et al. Triangle counting accelerations: From algorithm to in-memory computing architecture. *IEEE TC*, 2021.
[13] Robert Karam et al. Emerging trends in design and applications of memory-based computing and content-addressable memories. *Proceedings of the IEEE*, 2015.
[14] Junya Arai et al. Rabbit order: Just-in-time parallel reordering for fast graph analysis. *IPDPS*, 2016.
[15] Naveen Muralimanohar et al. Cacti 6.0: A tool to model large caches. *HP laboratories*, 2009.
[16] Yoongu Kim et al. Ramulator: A fast and extensible dram simulator. *IEEE Computer architecture letters*, 2015.
[17] Supreet Jeloka et al. A 28 nm configurable memory (tcam/bcam/sram) using push-rule 6t bit cell enabling logic-in-memory. *JSSC*, 2016.
[18] Aaron Stillmaker et al. Scaling equations for the accurate prediction of cmos device performance from 180 nm to 7 nm. *Integration*, 2017.
[19] Jure Leskovec et al. Snap datasets: Stanford large network dataset collection, 2014.
[20] Mohammed Elseidy et al. Grami: Frequent subgraph and pattern mining in a single large graph. *VLDB*, 2014.
[21] Bronwyn H Hall et al. The nber patent citation data file: Lessons, insights and methodological tools, 2001.
[22] Xu Cheng et al. Statistics and social network of youtube videos. In *IWQoS*, 2008.