HiTC: High-Performance Triangle Counting on HBM-Equipped FPGAs using HLS

Junzhe Liang, Manoj B. Rajashekar, Xingyu Tian, Zhenman Fang
HiAccel Lab, Simon Fraser University, Burnaby, Canada
{jla813, mba151, xingyut, zhenman}@sfu.ca

***Abstract*—Triangle counting (TC) is one of the fundamental computing patterns in graph computing and social networks. Due to its high memory-to-computation ratio and random memory access patterns, it is nontrivial to accelerate TC's performance. In this work, we propose a high-performance TC (HiTC) accelerator to speed up triangle counting on high-bandwidth memory (HBM)-equipped FPGAs via software/hardware codesign. First, we propose hardware-friendly reordering, tiling, and encoding techniques to address the random access issue and optimize bandwidth utilization. Based on that, we design a streaming-based FPGA accelerator that leverages HBM to achieve higher bandwidth and customize the computation pipeline for better computing throughput. Experiments using the SuiteSparse dataset show that our HiTC achieves a geomean speedup of 8.6x (up to 24.1x) over the Vitis TC FPGA library on the AMD/Xilinx HBM-based Alveo U280 FPGA. Compared to the software implementation on two 12-core Intel Xeon Silver 4214 CPUs, HiTC achieves a geomean speedup of 18.6x (up to 669.8x).**

I. Introduction

Graph theory is developed to understand and analyze many real-world scenarios, from social networks to logistics systems. Triangle counting (TC) is a fundamental task in graph theory and social network analysis, which determines the number of triangles passing through each node within a graph. A triangle consists of three nodes, each connected to the other two. The importance of TC in graph analysis is to reveal important structural properties and local connectivity patterns within networks [1]. TC is commonly used for community detection [1], clustering coefficients [2], analyzing social networks [3], and enhancing recommendation systems [3].

Accelerating the TC problem is beneficial for large-scale analysis tasks, especially for large graphs containing millions of vertices and edges that is common in graph processing nowadays [4]. For example, social media companies, such as Facebook [5] and LinkedIn [3], require analysis of social network structure. In the network graph, each node presents one user and connections between users are represented by edges. In this case, triangles demonstrate a closed loop of connections such as mutual friends. Nowadays, social network size can be massive (e.g., LinkedIn has 990 million members [4]), and analyzing the network using traditional computing methods can be very time-consuming and power-consuming.

There are serval challenges to implementing an efficient TC algorithm, since TC has a high memory-to-computation ratio, which requires massive random memory access compared to the amount of computation. The sparsity of the large-scale adjacent matrix leads to irregular memory access, imbalanced workloads, and degraded data locality. In addition, bitwise operation on the binary elements requires a comprehensive approach to enhance both parallelism and bandwidth efficiency.

In this work, we propose HiTC to accelerate TC on HBM-based FPGAs via software/hardware codesign with three major contributions. To the best of our knowledge, this is the first TC accelerator on FPGA using binary matrix multiplication.

1. We propose hardware-friendly reordering, tiling, and encoding techniques to handle random access and optimize bandwidth utilization.
2. We design a streaming-based FPGA accelerator, which exploits high bandwidth memory (HBM) and customized computation pipelines to improve the overall performance.
3. Experimental results show that, running on the AMD/Xilinx HBM-based Alveo U280 FPGA, HiTC outperforms the 24-core CPU implementation with an 18.6x geomean speedup (up to 669.8x), and exceeds the AMD/Xilinx Vitis TC library on the same FPGA, with a geomean speedup of 8.6x (up to 24.1x).

The rest of the paper is organized as follows: Sec. II covers the background and related work. Sec. III presents our hardware/software codesign of HiTC. Sec. IV illustrates experimental results and Sec. V concludes the paper.

II. Background and Related Work

A. TC with Bitwise Operations

Numerous methods have been proposed to count triangles, which can be divided into three categories: subgraph matching approach, intersection approach, and (binary) matrix multiplication method [6]. In this paper, we use the matrix multiplication-based approach for TC, utilizing the efficiency of bitwise operations and its high parallelism and scalability for processing large graphs. Moreover, there is no study on FPGA acceleration of the matrix multiplication method for TC yet, leaving ample room for development in this area.

The matrix multiplication-based approach for TC is a class of algorithms that uses an adjacency matrix to perform TC. Let A be the symmetric adjacency matrix representation of an undirected graph $G(V, E)$. $A[i][j] \in \{0, 1\}$ indicates whether there is an edge between vertices i and j. $A^2[i][j]$ and $A^3[i][j]$ shows the number of paths that start from i to j using two steps and three steps, respectively. In this case, the value on the diagonal value, $A^3[i][i]$, indicates the number of triangles starting from V_i. After excluding repeated triangles, the number of unique triangles of graph G can be calculated:

$$TC(G) = (\sum diag^{-1}(A \times A \times A))/6 \quad (1)$$

Azad, et al. [7] further improves this method with a masking procedure to reduce computation complexity:

$$TC(G) = nnz(A \cap (L \times U))/2 \quad (2)$$

where L is the lower triangular part of A, and U is the upper triangular part of A, with all zeros on the diagonal. nnz stands for number of nonzeros.

In this algorithm, $L \times U$ produces wedges of the graph, and $A \cap (L \times U)$ filters out wedges that are not connected by a third edge. A further improvement is proposed by Sandia [6] which replaces L and A by U. Using the U instead of A to do the element-wise multiplication counts each triangle exactly once instead of twice. Thus Sandia [6] reduces both input data size and the number of required operations:

$$TC(G) = nnz(U \cap U^2) \quad (3)$$

As U is a sparse binary matrix, multiplication in this formula can be replaced with AND operation in hardware. Accumulation can be done with a bit counter (BitCount):

$$TC(G) = \text{BitCount}(\text{AND}(U[i][*], U[*][j]^T)),\ \forall U[i][j] = 1 \quad (4)$$

B. Related Work

Recently, the TC problem has been accelerated across different platforms, including CPUs, GPUs, and FPGAs.

Several techniques have been proposed to accelerate TC on CPUs, including multi-threading and vectorization. For example, TCM [8] employs a matrix multiplication-based approach with row-wise partitions, enabling parallel processing of sub-matrices. However, various sub-matrices might access the entire graph concurrently, necessitating shared memory space among multiple processor cores. TC-SMID [9] utilizes fast vector instruction implementations of set operation-based algorithms to directly compute the exact triangle count.

For GPU implementations, bbTC [10] uses a blocked-based matrix multiplication with rectilinear partitioning. However, this way of partition requires complex preprocessing to find the suitable cutting position. HPETC [11] takes advantage of a set intersection operation implemented relying on bitmaps and on atomic operations. Tom et al. [12] implement the map-based algorithm using GraphMat, a parallel and distributed graph processing framework. In general, GPUs are power-hungry and are not friendly for bitwise operations.

The work by Huang et al. [13] is the only existing paper for TC on FPGA. It uses the intersection-based method for triangle counting, which iterates over each edge and finds common elements from two adjacency lists of head and tail nodes. However, we could only synthesize by not being able to build the design based on their source code. In addition, AMD/Xilinx Vitis TC library [14] also uses an edge-based set intersection way for TC. In Sec. IV-B, we will present a quantitative comparison to the multicore CPU implementation and Vitis TC library FPGA implementation.

C. Challenges for Accelerating TC on FPGAs

FPGAs possess great flexibility and power efficiency. However, the TC accelerator design on FPGAs introduces more challenges in addition to the ones explained in Sec. I.

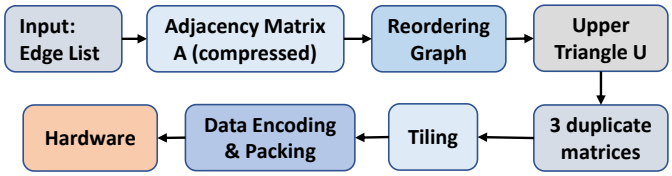

Fig. 1: The overview of the proposed HiTC software workflow.

1. **Limited On-chip Memory:** Limited on-chip memory resources on FPGA are insufficient to buffer large graph matrices, requiring an efficient tiling scheme.
2. **Data Format and Hardware Codesign:** Widely used compressed formats, such as CSR (compressed sparse row), are not friendly for bursting memory accesses and computation parallelism. Customized data format and corresponding accelerator pipeline design are necessary to improve the computing throughput.
3. **HBM Bandwidth Utilization:** HBM-equipped FPGA provides a high bandwidth potential, along with the challenge to fully utilize it. An efficient accelerator design should optimize bandwidth across all HBM channels.

III. HiTC Design

To address the aforementioned challenges of TC accelerator design, HiTC takes a hardware/software codesign approach. The software part preprocesses the data stored in CSR and transforms it into a customized hardware-friendly format through graph reordering, tiling, and encoding. While in the hardware part, we design a streaming accelerator on the FPGA, which exploits the off-chip bandwidth and customizes the computing pipeline for the bitwise computation.

A. Hardware-friendly Software Preprocessing

Fig. 1 shows the software preprocessing workflow of HiTC codesign, which includes graph reordering, tiling, and encoding, to enhance the computation efficiency on hardware.

Graph Reordering: The distribution of elements inside an adjacency matrix is usually diverse with poor data locality. We use the minimum degree order (MDO) algorithm [15] to sort the nodes in the ascending order of their degree value and rename the node ID of the graph. As depicted in Fig. 2, compared to the original adjacency matrix, MDO helps gather most of the nonzero elements into the right corner.

Data Tiling and Encoding: As will be explained later in Fig. 5, the reordered adjacent matrix is duplicated into A, B, and C. While A is streamed in, we need to buffer B and C. Buffering sparse matrices on-chip is nontrivial, as traditional 2D tiling for dense matrices uses a fixed tile shape and buffers many zeros. To address the limited on-chip memory issue, we propose a dynamic tiling technique to tile as many nonzero elements as it can until it reaches the hardware buffer size limit. Fig. 3 shows an example with a hardware buffer size limit of 2×2. So, in a tile, the distribution of nonzero elements in any row and any column will not exceed 2. In our actual design, the buffer size is 512 nonzero rows $\times$ 31 nonzero columns (due to data encoding).

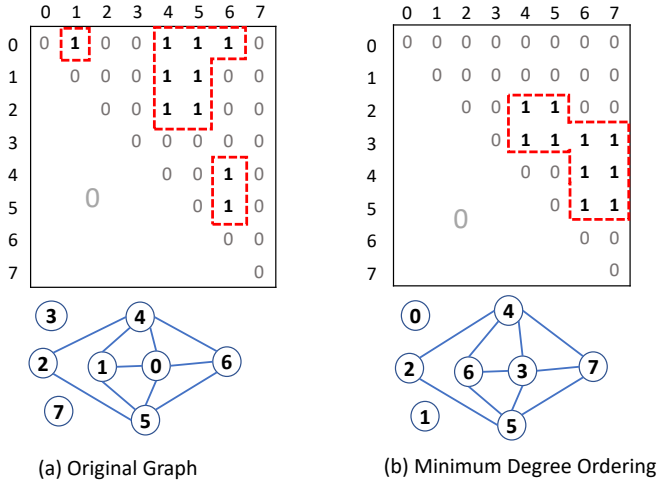

Fig. 2: Graph reordering: minimum degree order (MDO).

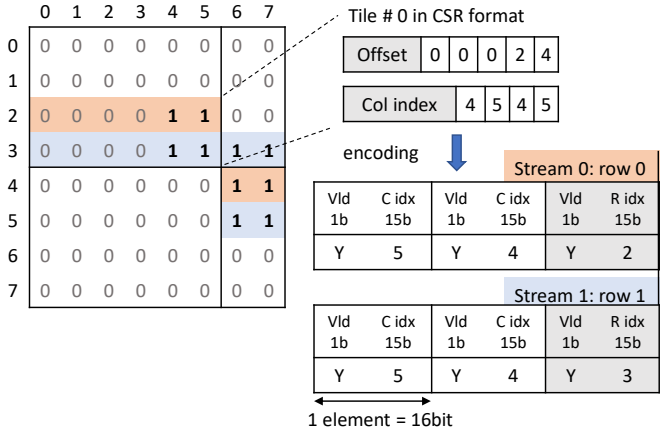

Fig. 3: Customized sparse matrix tiling and encoding example.

Moreover, to improve the off-chip bandwidth utilization, each row will be encoded into a fixed-width packet including a row index and multiple column indices. Both row index and column indices are 15 bits and 1 valid bit is appended to each index. In this example in Fig. 3, each tile will be encoded into 2 packets where each packet has 2 column indices. In our actual design, the width of each packet is 512 bits (i.e., 31 columns per row) to fully utilize the HBM bandwidth.

B. Hardware Design

1) Overview Architecture: Fig. 4 illustrates the overall architecture of our streaming-based accelerator. The hardware accelerator incorporates multiple pairs of load units and fine-tuned processing element groups (PEGs) to enable efficient parallel processing. Each load unit reads the preprocessed data from HBM banks in a streaming fashion, which fully utilizes the bandwidth of each HBM bank. By scaling up the number of pairs, HiTC can maximize bandwidth utilization across more HBM banks. In each PEG, the data is processed tile by tile. Multiple PEGs can process different tiles in parallel, generating partial results, which are then accumulated by an adder tree to get the total number of triangles.

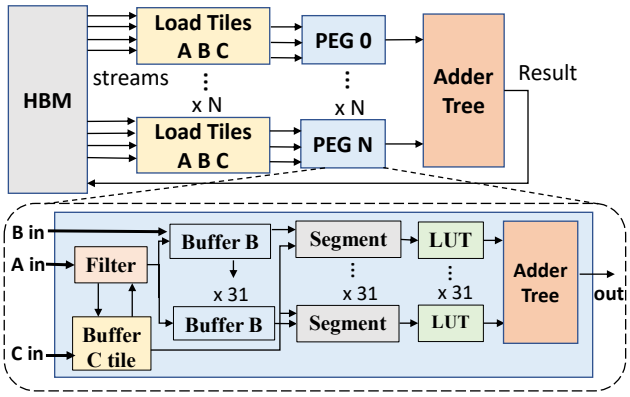

Fig. 4: Overall streaming architecture of HiTC.

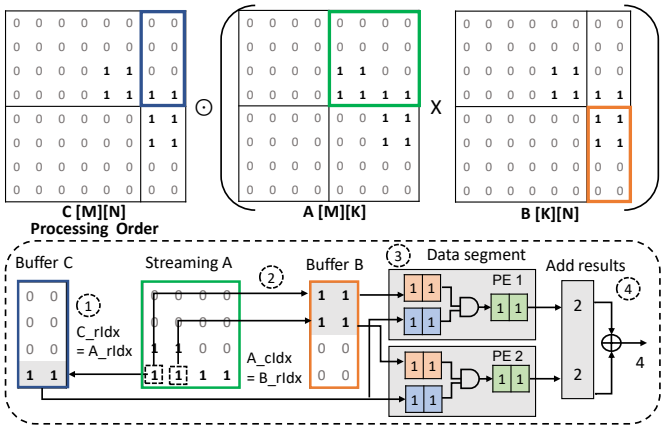

Fig. 5: Processing order inside each PE group (PEG).

2) Processing Order of Individual PEG: The detailed processing order in each PEG is shown in Fig. 4 and Fig. 5.

First, one B tile and one C tile are buffered on-chip in parallel. The B tile is duplicated to 31 buffers by chain-based broadcasting, which is more efficient than one-to-all broadcasting due to its better timing closure.

After that, tiles of A are read in a streaming fashion by the filter module. This module checks A row indices against the C buffer row indices: When a row match is found, the corresponding row of buffer C is forwarded to all segment modules. The column indices of in the current A row are used to access rows of B buffer, where each row of B is forwarded to one segment module. Since each packet of A has up to 31 columns, each PEG has 31 PEs (i.e., 31 buffers for B, 31 segments, and 31 LUT modules) running in parallel.

Next, we need to check the number of common bits in the pair B and C rows. To check multiple elements concurrently, we decode the B and C rows into a dense representation. To reduce resource usage and avoid unnecessary checking for zero elements, we perform data slicing on each pair of B row and C row into multiple segments, and skip those segments with all zeros. To check the number of common bits in each pair of B and C nonzero segments (with the same segment ID), we use a lookup table (LUT) to perform this computation in one cycle (parallelism per LUT equals segment size).

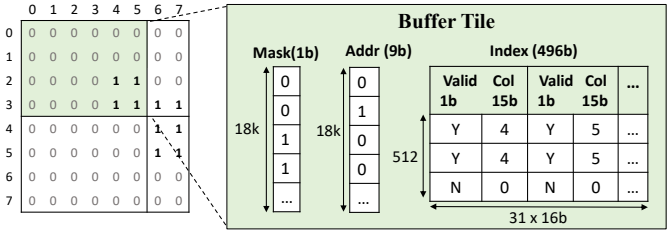

Fig. 6: Hardware component: buffer module design.

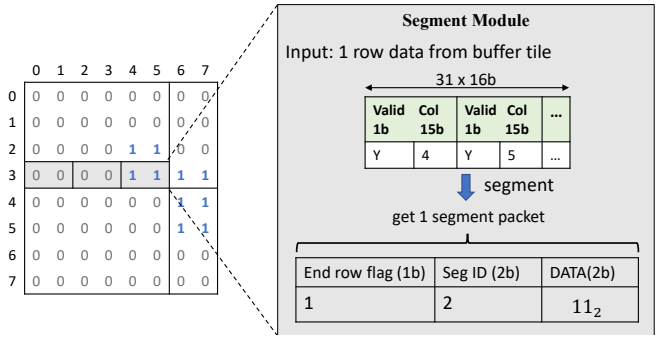

Fig. 7: Hardware component: data segment module design.

Finally, results produced by all LUTs within a PEG are accumulated by an adder tree. In the example in Fig. 5, each LUT counts 2 triangles and the final result of this PEG is 4.

3) Buffer Scheme: To maximize the data reuse and guarantee efficient parallel accesses on buffered tiles, HiTC incorporates a buffering scheme addressing this challenge. Each buffer module has three components: an index array, an address array, and a mask array to store tiles B and C as shown in Fig. 6.

Each tile is buffered packet by packet and the index array stores the actual preprocessed packets. Due to the sparsity, the index array may not store consecutive rows. To address this problem, we propose a combination of mask array and address array to simplify the process of finding a corresponding row.

For each preprocessed packet read from HBM, it first checks the address array to get the corresponding row ID in the index array. Then it sets the corresponding bit of the mask array to 1. The size of the mask array (and the index array) is set as 18K to fully utilize an 18Kb BRAM bank. In other words, each tile has no more than 18k rows.

Then the column indices inside this packet are stored in one row of the index array. Due to limited on-chip memory size, we set the depth of the index array to 512, i.e., it can store a maximum of 512 packets. The row ID of the index array can be represented by 9 bits and this row ID is stored in the address array. When loading a new tile, we only need to reset the mask array and there is no need to reset the whole buffer.

4) Data Segment Design: Column indices in each row of the index array could be discontinuous, making it hard to compare nonzero elements of B row and C row, and find the common bits in parallel. To address this challenge, we decode each pair of B row and C row into multiple comparable (nonzero) dense segments through a segment module.

Each row in a buffer can be decoded and divided into multiple fixed-size nonzero dense segments. As an example shown in Fig. 7, each row of the tile is divided into 3 segments with size of 2. When we check the column indices in the index array, only the last segment has nonzero elements. Then, this segment is packed with 1) a 1-bit end row flag indicating whether it is the last segment on this row of tile, 2) a 2-bit segment ID storing the segment index, and 3) a 2-bit DATA representing all elements inside this segment in a dense format (i.e., including all 0s and 1s). In the actual design, we choose the segment size to be 16 elements, i.e., the length of DATA is 16-bit. Since we use 15 bits to represent a column index in a tile (Sec. III-A), the length of segment ID is 11-bit $(15 - log_2 16)$. The segment ID can be calculated as:

$$\text{segment\_ID} = \text{column\_index}/\text{segment\_size} \tag{5}$$

After the extraction of nonzero segments of B row and C row, we use LUT to compare the segments with the same segment ID and count the number of common bits in one cycle.

Algorithm 1 Software to hardware scheduling in HiTC.

1: **Input:** (1) 3 copies of matrix U: A[M][K], B[K][N], C[M][N]; (2) matrix hyper-parameters: m_{cut}, n_{cut}, k_{cut}; (3) Q pointer
2: **Output:** number of triangles: tc (initialized to 0)
3: **for** k = 0 to k_{cut} **do** ▷ go through each cut in k dimension
4: **for** n = 0 to n_{cut} **do**
5: buffer tile B_{kn} ▷ maximize data reuse of B_{kn}
6: **for** m = 0 to m_{cut} **do**
7: buffer tile C_{mn} ▷ data reuse for C_{mn} as well
8: **for** p = 0 to $\#PEG$ **do** ▷ run in parallel
9: **for** $i = Q_{mk}$ to Q_{mk+1} **do** ▷ nonzero rows of A_{mk}
10: stream in $A_{mk}[i]$
11: **if** $C_{mn}[i]$ is not empty **then**
12: slice $C_{mn}[i]$ into a set of segments $Segs_C$
13: **for** j = 0 to 31 **do** ▷ 31 PEs run in parallel
14: search corresponding row in B_{kn}
15: slice the B row into a set of segments $Segs_B$
16: $tc \leftarrow tc$ + segment_LUT($Segs_B$, $Segs_C$)

C. Software to Hardware Scheduling

Algorithm 1 outlines the pseudo-code for the software to hardware scheduling in HiTC. Initially, the input graph is represented as the upper triangle part of an adjacency matrix format, with three copies named matrix $A[M][K]$, $B[K][N]$, and $C[M][N]$, shown in Fig. 5. To handle large-scale graphs, the three input matrices are tiled into sub-matrices. To optimize the on-chip buffering of B tiles, we aim to reuse the current B tile as much as possible, which dictates the computation order across the tile level (lines 3-5).

After buffering B (line 5) and C (line 7) tiles on-chip, nonzeros within a tile of A are streamed into PEGs from multiple HBM channels (lines 8-10). Each PEG is cyclically assigned a distinct set of A rows, and a pointer list Q (line 9) tracks the tile position inside matrix A.

Inside each PEG, it filters the current A row by checking if the corresponding C row is not empty (line 11); if it is not empty, the corresponding C row is decoded and sliced into segments $Segs_C$ (line 12), as explained in Sec. III-B4. After filtering, we use the column indices of the nonzeros in

TABLE I: Selected graph dataset from SuiteSparse.

Dataset	# Vertices	# Edges	Density
kron_g500-logn17	131,070	5,113,985	5.95E-04
TEM181302	77,360	3,828,854	1.28E-03
raefsky6	6,316	134,443	6.74E-03
bundle1	10,581	380,160	6.79E-03
facebook	4,039	88,234	1.08E-02
mouse_gene	45,101	14,461,095	1.42E-02
mycielskian15	24,575	5,555,555	1.84E-02
mycielskian14	12,287	1,847,756	2.45E-02
mycielskian13	6,143	613,871	3.25E-02
human_gene1	22,283	12,323,680	4.96E-02
raefsky1	36,417	291,034	4.39E-04
human_gene2	14,340	9,027,024	8.78E-02

the current A row to search the corresponding rows in the B buffer. Since there are 31 columns in an A packet, we use 31 PEs to do the search and processing in parallel (lines 13-16). Inside each PE, the found B row is also decoded and sliced into segments $Segs_B$. After that, the segment_LUT function (line 16), receives the two sets of segments for B and C in a streaming fashion and outputs the number of common nonzero bits, as explained in Sec. III-B4. Finally, these results are accumulated to get the final number of triangles.

IV. Experimental Results

A. Experimental Setup

We evaluate HiTC on the AMD/Xilinx HBM-based Alveo U280 FPGA using Vitis HLS and Vitis 2021.2. We compare the HiTC performance with the open-source Vitis TC FPGA library [14] on the same FPGA and an optimized 48-thread CPU implementation running on two 12-core Intel Xeon Silver 4214 CPUs. As shown in Table I, our evaluation includes 12 real-world graphs from the widely used SuiteSparse matrix collection [16], ranging from 4K to 131K vertices and 88K to 14M edges, with densities from 4.39E-04 to 8.78E-02. We measure the HiTC performance (with 6 PEGs) on the actual FPGA board and exclude the preprocessing time. And we use the post place-and-route report for the resource utilization.

B. Comparison to Multicore CPU and Prior FPGA Design

Fig. 8 compares our HiTC performance against a 24-core CPU implementation and the Vitis TC FPGA library [14] design on real-world graphs. It presents the relative speedup with the single-core CPU implementation as the baseline, where higher values indicate better performance. We evaluate both HiTC and Vitis TC library on the Alveo U280 FPGA. The other FPGA implementation proposed by Huang [13] is omitted since it only provides synthesis results.

First, the 24-core multi-core CPU implementation shows a geometric mean speedup of approximately 8x across 12 datasets over the single-core CPU, ranging from 2x to 20x. The speedup of the multi-core CPU is limited by irregular

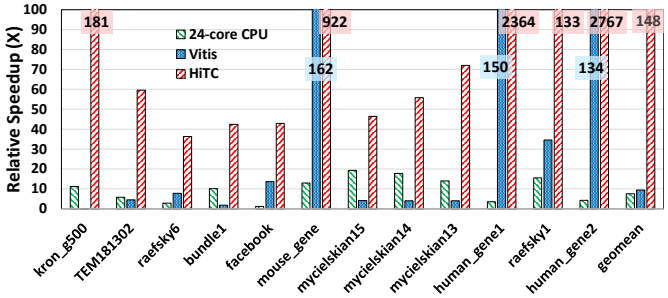

Fig. 8: Relative performance speedup comparison of 24-core CPU, Vitis FPGA library design, and our HiTC on real-world graphs. The baseline is single-core CPU performance.

data dependency, memory bandwidth, and resource contention. These factors can reduce the overall throughput.

Second, the Vitis TC library design outperforms the 24-core CPU implementation, achieving speedups ranging from 2x to 162x over the single-core CPU.

Third, our HiTC design achieves remarkable speedups ranging from 36x to 2,767x over the single-core baseline. Three datasets, *mouse_gene*, *human_gene1*, and *human_gene2*, exhibit speedups exceeding 900x compared to the baseline, with densities of 1.42E-02, 4.96E-02, and 8.78E-02, respectively. This is because those datasets have a more balanced distribution of nonzero elements after tiling, and various PEGs can have similar (balanced) workloads. The speedup on other datasets such as the Facebook dataset shows less improvement. For example, it achieves about 42x speedup on the Facebook dataset. This dataset is a friend list on Facebook, where most edges are dominated by a minority of vertices, leading to an extremely imbalanced distribution of edges. Such distribution results in imbalanced workloads and limits the performance.

Finally, on average (geomean), HiTC achieves an 18.6x speedup over the 24-core CPU implementation, and an 8.6x speedup over the Vitis TC library design on the same FPGA.

C. Resource Utilization and Design Frequency

TABLE II: HiTC resource utilization

Resource Utilization					Freq. (MHz)
LUT	FF	BRAM	URAM	DSP	
57.6%	34.8%	65.6%	69.6%	34.9%	211

Table II presents the resource utilization of our HiTC design, in terms of Look-Up Tables (LUTs), Flip-Flops (FFs), Block RAMs (BRAMs), Ultra RAMs (URAMs), and Digital Signal Processors (DSPs) used in the Alveo U280 FPGA. The high utilization of BRAM and URAM in HiTC mainly comes from buffer modules: it has 6 PEGs and each PEG uses 32 buffer modules, resulting in a total of 192 buffer modules in the final design. LUTs are extensively utilized for bit-wise operations in the PEGs.

To fully utilize the bandwidth of the 512-bit HBM channel running at 450MHz, a hardware accelerator should operate at more than 225MHz. HiTC achieves a very close frequency at 211 MHz due to timing closure and routing congestion issues.

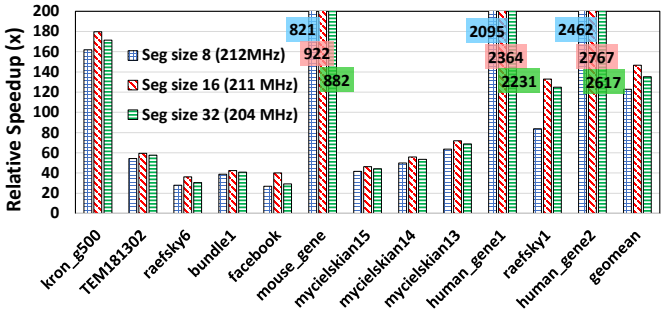

Fig. 9: Performance comparison for different segment sizes.

D. Ablation Study on Segment Size

Among HiTC configurations, the segment size primarily determines the trade-off between performance and resource usage. PEGs with larger segment sizes can process more nonzero elements in parallel, but require more resources. In the prior subsections, we showed the results for the default segment size of 16 and now we do an ablation study.

Fig. 9 compares the relative speedup of HiTC over the single-core CPU baseline, with three data segment sizes ranging from 8 to 32. It confirms that HiTC with a segment size of 16 reaches the best performance, which is our default choice.

Also note that, as the segment size increases, the resource utilization increases, and thus the hardware frequency degrades due to placement and routing congestion.

V. Conclusion

In conclusion, we have presented HiTC, the first matrix-multiplication-based triangle counting accelerator on FPGAs. To address the random access issues and optimize the locality and bandwidth utilization, we have proposed the hardware friendly graph reordering, sparsity-aware tiling, and encoding techniques. Building on top of that, we have designed a streaming-based accelerator architecture on HBM-based FPGAs. Inside this streaming architecture, we have also proposed efficient buffering techniques to accommodate random distribution of nonzero elements within fixed on-chip buffers. Leveraging the characteristics of binary sparse matrix multiplication, we have customized the computation pipeline to decode and segment sparse data into compact dense representations, and leverage lookup tables to compute the number of triangles.

Experiments with the widely used SuiteSparse dataset show that, HiTC achieves a geometric mean speedup of 8.6x (up to 24.1x) over the Vitis TC FPGA library on the same AMD/Xilinx Alveo U280 FPGA. Compared to the software implementations on two 12-core Intel Xeon Silver 4214 CPUs, HiTC achieves a geometric mean speedup of 18.6x (up to 669.8x). We plan to open source our design in future work.

VI. Acknowledgement

This work was supported in part by NSERC Discovery Grant RGPIN-2019-04613, DGECR-2019-00120, Alliance Grant ALLRP-552042-2020; CFI John R. Evans Leaders Fund and BC Knowledge Development Fund; Huawei Canada and AMD-Xilinx.

References

[1] K. Sotiropoulos and C. E. Tsourakakis, "Triangle-aware spectral sparsifiers and community detection," in *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, ser. KDD '21, 2021, p. 1501–1509.

[2] T. G. Kolda, A. Pinar, T. Plantenga, C. Seshadhri, and C. Task, "Counting triangles in massive graphs with mapreduce," *SIAM Journal on Scientific Computing*, vol. 36, no. 5, pp. S48–S77, 2014.

[3] LinkedIn, "People you may know feature," https://www.linkedin.com/help/linkedin/answer/a544682/people-you-may-know-feature-, accessed: [2024-03-21].

[4] Hannah Macready, "51 linkedin statistics you need to know in 2024," https://blog.hootsuite.com/linkedin-statistics-business/, accessed: [2024-03-21].

[5] Facebook, "Facebook," https://www.facebook.com/, accessed: [2024-03-21].

[6] M. M. Wolf, M. Deveci, J. W. Berry, S. D. Hammond, and S. Rajamanickam, "Fast linear algebra-based triangle counting with kokkoskernels," in *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, 2017, pp. 1–7.

[7] 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*, 2015, pp. 804–811.

[8] J. Shun and K. Tangwongsan, "Multicore triangle computations without tuning," in *2015 IEEE 31st International Conference on Data Engineering*, 2015, pp. 149–160.

[9] K. Ravichandran, A. Subramaniasivam, P. Aishwarya, and N. Kumar, "Chapter eight - fast exact triangle counting in large graphs using simd acceleration," in *Principles of Big Graph: In-depth Insight*, ser. Advances in Computers, R. Patgiri, G. C. Deka, and A. Biswas, Eds. Elsevier, 2023, vol. 128, pp. 233–250.

[10] A. Yasar, S. Rajamanickam, J. W. Berry, and U. V. Catalyurek, "A block-based triangle counting algorithm on heterogeneous environments," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 2, pp. 444–458, 2022.

[11] M. Bisson and M. Fatica, "High performance exact triangle counting on gpus," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 12, pp. 3501–3510, 2017.

[12] A. S. Tom, N. Sundaram, N. K. Ahmed, S. Smith, S. Eyerman, M. Kodiyath, I. Hur, F. Petrini, and G. Karypis, "Exploring optimizations on shared-memory platforms for parallel triangle counting algorithms," in *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, 2017, pp. 1–7.

[13] S. Huang, M. El-Hadedy, C. Hao, Q. Li, V. S. Mailthody, K. Date, J. Xiong, D. Chen, R. Nagi, and W.-m. Hwu, "Triangle counting and truss decomposition using fpga," in *2018 IEEE High Performance extreme Computing Conference (HPEC)*, 2018, pp. 1–7.

[14] AMD/Xilinx, *Vitis Libraries - Triangle Count*, 2022, accessed: 2024-02-17. [Online]. Available: https://xilinx.github.io/Vitis_Libraries/graph/2022.1/guide_L2/manual/triangleCount.html

[15] H. M. Markowitz, "The elimination form of the inverse and its application to linear programming," *Management Science*, vol. 3, no. 3, pp. 255–269, 1957.

[16] University of Florida Sparse Matrix Collection. The suitesparse matrix collection. [Online]. Available: https://sparse.tamu.edu/