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

by

Junzhe Liang

B.Sc., The University of Melbourne, 2021

Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of
Master of Applied Science

in the
School of Engineering Science
Faculty of Applied Sciences

SIMON FRASER UNIVERSITY
Spring 2024

Declaration of Committee

Name:	**Junzhe Liang**
Degree:	**Master of Applied Science**
Thesis title:	**HiTC: High-Performance Triangle Counting on HBM-Equipped FPGAs Using HLS**
Committee:	**Chair:** Majid Shokoufi Lecturer, Engineering Science
	Zhenman Fang Supervisor Assistant Professor, Engineering Science
	Tianzheng Wang Committee Member Assistant Professor, Computing Science
	Jiangchuan Liu Examiner Professor, Computing Science

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 streaming-based hardware accelerators on FPGAs which leverage 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).

Keywords: Triangle Counting; Binary Matrix Multiplication; High-Level Synthesis; Hardware Acceleration; HBM-Equipped FPGA

Acknowledgements

Firstly, I would like to thank my supervisor Dr. Zhenman Fang for his constant support in this project throughout the Master's program. Coming from a mechatronic engineering background with limited experience in hardware, Dr. Fang patiently guided me, helping me acquire the necessary hardware expertise and finish my thesis. This work would not have been possible without the countless discussions with him where he provided insightful ideas and feedback.

Secondly, I would like to thank the people who have had a direct involvement in this work: My colleague Manoj provided invaluable inspiration during the design stages, while Xingyu offered crucial technical support. This work would not be possible without my colleague's constant support and motivation. I extend my appreciation to all the members of our HiAccel lab, including Abdul, Dilshan, Whilliam, Kenny, Haisheng, Alec, Moazin, Akhil, Qilin, Kartik, Weihua, and Ahmad. Their countless technical discussions have greatly enriched my experience and facilitated my understanding of this research field. I am honored to be part of such a supportive and friendly team. Furthermore, I would like to acknowledge the funding support from our sponsors NSERC, CFI, Huawei, and Xilinx, without which this research would not have been possible.

Furthermore, I am grateful for the friendships I have gained outside the lab, including Yuanyuan, Yonghao, Yuning, Angela, Yueyang, Lynchee, Xu Xiang, and others I have met in Vancouver. Their constant encouragement and companionship have been a source of joy during challenging times. Finally, I would like to thank my parents, and my uncle's family for their unwavering support and encouragement while I studied overseas. These three years in Canada were the most challenging period of my life, and I am thankful to all the people who helped and supported me.

Table of Contents

Declaration of Committee ii

Abstract iii

Acknowledgements iv

Table of Contents v

List of Tables vii

List of Figures viii

1 Introduction 1

1.1 Contributions of HiTC . . . 3

2 Background and Related Work 4

2.1 Three Different Approaches for TC . . . 4

2.1.1 Subgraph Matching to a Triangle Pattern . . . 4

2.1.2 Set-Intersection Approach . . . 5

2.1.3 Matrix Multiplication-based Approach . . . 5

2.1.4 Four Different Ways of Computing Sparse-sparse Matrix Multiplication 7

2.1.5 Challenges to Accelerate TC . . . 9

2.2 Compression Format for Sparse Matrix . . . 10

2.3 Related CPU and GPU Works . . . 10

2.3.1 Previous TC Acceleration Works on CPU . . . 11

2.3.2 Previous TC Acceleration Works on GPU . . . 12

2.4 Challenges for Accelerating TC on FPGAs . . . 13

2.4.1 Previous TC Acceleration Works on FPGAs . . . 13

3 HiTC Design 14

3.1 Overall End-to-end TC Accelerator Workflow . . . 14

3.2 Hardware-friendly Software Preprocessing . . . 14

3.2.1 Graph Reordering Algorithm . . . 16

3.2.2 Hardware-friendly Tiling Technique . . . 18
3.2.3 Sparse Matrix Format . . . 21
3.3 Comparator-based Hardware Design . . . 23
3.3.1 Accelerator Architecture Overview . . . 24
3.3.2 Detailed Processing . . . 25
3.3.3 Buffering Scheme . . . 26
3.3.4 Comparator Design . . . 27
3.3.5 Algorithm . . . 28
3.3.6 Performance Analysis and Modeling . . . 30
3.3.7 On-chip Memory Resource Analysis and Modeling . . . 32
3.4 Lookup Table Based Hardware Design . . . 32
3.4.1 Accelerator Architecture Overview . . . 33
3.4.2 Data Segment Design . . . 35
3.4.3 LUT-based Intersect Module . . . 36
3.4.4 Algorithm . . . 37
3.4.5 Performance and On-chip Memory Analysis and Modeling . . . 39

4 Experimental Results **41**
4.1 Evaluation Setup . . . 41
4.2 Performance of Comparator-based Hardware Design . . . 42
4.2.1 Design Configuration . . . 42
4.2.2 Performance Scaling Analysis . . . 44
4.2.3 Execution Time Breakdown . . . 44
4.3 Performance of Lookup Table-based Hardware Design . . . 45
4.3.1 Design Configuration . . . 46
4.3.2 Segment Size Analysis . . . 46
4.3.3 Performance Scaling Analysis . . . 47
4.3.4 Execution Time Breakdown . . . 48
4.4 Overall Performance Comparison between Design 1 vs Design 2 . . . 49
4.5 Resource Utilization and Design Frequency . . . 50
4.6 Comparison with CPU and Other FPGA Design . . . 51

5 Conclusion and Future Work **55**

Bibliography **57**

List of Tables

Table 2.1 Notation used in this thesis 6
Table 2.2 Related Works in CPU, GPU and FPGA 11

Table 4.1 Selected graph dataset. 42
Table 4.2 Comparator-based hardware design parameters 43
Table 4.3 LUT-based hardware design parameters 46
Table 4.4 HiTC resource utilization and frequency 51

List of Figures

Figure 2.1 Four different ways of computing SpGEMM. 8

Figure 3.1 The overview of the proposed HiTC workflow. 15
Figure 3.2 Graph reordering techniques: Reverse Cuthill-McKee (RCM) vs. Minimum Degree Order (MDO). 16
Figure 3.3 HiTC computation order 19
Figure 3.4 An example of tiling a sparse matrix 20
Figure 3.5 Comparison of non-empty tasks after tiling sparse matrix using RCM and MDO algorithms. A non-empty task (tile_A, tile_B, tile_C) means all three tiles are non-empty. 21
Figure 3.6 An example of the customized sparse matrix format. 22
Figure 3.7 Comparator-based hardware design architecture. 24
Figure 3.8 Processing order for comparator-based hardware design. 25
Figure 3.9 Hardware component: buffer module design. 26
Figure 3.10 The proposed LUT-based hardware design architecture. 33
Figure 3.11 Processing order for LUT-based hardware design. 34
Figure 3.12 Hardware component: data segment module design. 35

Figure 4.1 Comparison of the relative speedup for various scales of the comparator-based hardware design against the baseline. 43
Figure 4.2 Comparator-based design execution time breakdown 45
Figure 4.3 Performance comparison for LUT-based hardware design with variant segment size. 47
Figure 4.4 Performance comparison when scaled-up LUT-based hardware design. 48
Figure 4.5 LUT-based design execution time breakdown 49
Figure 4.6 Accuracy of analytical module in LUT-based design 50
Figure 4.7 The speedup of LUT-based design over comparator-based design . . 51
Figure 4.8 HiTC performance comparison of our design against 1-core CPU (baseline), multi-core CPU, and Vitis FPGA design on real-world graphs 52
Figure 4.9 Throughput comparison with multi-core CPU and Vitis FPGA design and HiTC on real-world graphs. 54

Chapter 1

Introduction

Graph theory is an effective tool to understand and analyze many real-world scenarios, from social networks to logistics systems. Graphs can efficiently model relationships between entities, and clearly visualize connections, dependencies, and interactions. Triangle counting (TC) is a fundamental task in graph theory and network analysis used to determine the number of triangles passing through each node within a given graph. A triangle is a set of three nodes, where each node is connected with both of the other two nodes. The importance of triangle counting in graph analysis is to reveal important structural properties and local connectivity patterns within networks [27]. Triangle counting is commonly used for community detection [28], clustering coefficients [19], analyzing social networks, enhancing recommendation systems, etc. For example, if users A, B, and C form a triangle in the recommendation system case, they would have similar preferences. Recommending items liked by one user to the others can be more effective. Also, triangle counting is closely related to community detection in network analysis. In general nodes within the same community tend to be strongly connected, forming closed triangles. By counting triangles, users can easily identify those network regions that potentially form communities.

Accelerating the triangle counting problem is beneficial for large-scale analysis tasks, especially when dealing with massive graphs containing millions of vertices and edges that become common things in graph processing. For example, social media companies such as Facebook [10] and LinkedIn [20], need to analyze the connections between users to understand the network's structure. Each user is represented by a node and connections between users are represented by edges. Triangles in this case demonstrate a closed loop of connections such as mutual friends. Nowadays, social network size can be massive, thus analyzing the network using traditional computing methods can be extremely time-consuming and power-consuming, which is unrealistic. For example, in LinkedIn [20], a popular technique for friend recommendation, also known as "People You May Know", needs triangle counting algorithms for graph analysis. Having a real-time or near-real-time performance of the triangle counting algorithm can improve users' experience and stay ahead of competitors. Thus, accelerating the triangle counting problem is important for many real-world scenarios.

As an alternative hardware acceleration platform, Field Programmable Gate Arrays (FPGAs) have received strong interests from academic researchers and the industry. Compared to other hardware acceleration platforms like Graph Processing Units (GPU), users can customize their hardware design, and FPGAs are more energy efficient. There are various approaches to solving TC problem, and selecting a method that is more suitable for FPGAs acceleration becomes the first question.

The current TC algorithms can be classified into three approaches, namely matrix multiplication-based, set-intersection-based, and subgraph matching-based methods. For this project, we choose matrix multiplication-based TC as it is a more suitable approach for FPGAs acceleration, and a detailed explanation will be presented in Chapter 2.

TC problem has a high memory-to-computation ratio, meaning that the algorithm requires a significant amount of memory access relative to the amount of computation performed. This characteristic poses five critical challenges when accelerating a matrix multiplication-based TC on FPGA. Firstly, the sparse nature of the graphs used in TC results in irregular memory access patterns and imbalanced workload distribution. Secondly, the size of large-scale sparse matrices often exceeds the capacity of limited on-chip buffers. Thirdly, the traditional compressed sparse row (CSR) format is not HBM friendly for continuously burst reading data as it needs to interact between the offset array and index array. Fourthly, the challenge of optimizing hardware design to leverage bit-wise operations for binary sparse matrices necessitates a customized architectural approach. Lastly, timing closure problems can arise when scaling up hardware design on multi-die FPGAs. Addressing these challenges requires a comprehensive approach to both software and hardware design and optimization, tailored specifically to the unique requirements of matrix multiplication-based TC on FPGAs.

In this thesis, we propose a novel matrix multiplication-based TC accelerator, HiTC, to overcome those challenges. In the hardware-friendly software prepossessing step, we first utilize a graph reordering strategy, called Minimum Degree Order (MDO) [21] to rearrange the vertices of the graph in a way that reduces the distance between adjacent vertices in memory. As a result, it can improve data locality, and lead to faster memory access times. When triangle counting algorithms access neighboring vertices or their adjacency lists, there is a higher likelihood that these vertices are located close to each other in memory. After graph reordering, we introduce a sparsity-aware tiling technique designed for large-scale datasets. This method tiles the large sparse matrix into small tiles, allowing only a small tile to be loaded onto the FPGA at a time. Conventional DDR memory offers limited memory bandwidth [26]. To overcome this limitation, we utilize HBM to support higher memory bandwidth. Nevertheless, fully exploiting the bandwidth benefit of HBM is non-trivial. To address this, we compress the sparse matrix in each tile and apply data encoding and packing to enable streaming access. Additionally, we employ memory coalescing and bursting techniques to optimize off-chip memory access. In the hardware part, we propose

an efficient buffer technique to fit random distribution nonzero elements on the fixed on-chip buffer. To leverage the characterise of binary sparse matrix multiplication, we propose two different ways of computation: comparator-based hardware design and lookup table based hardware design. These two hardware designs are used to support different distributions of the input matrix. Overall HiTC addresses the aforementioned challenges and is the first work to implement matrix-multiplication-based triangle counting on FPGA.

We compare the hardware performance of HiTC with 24-core CPU and other FPGA implementations. There are only two existing FPGA implementations, one is from Huang et al. [18] and another is from Vitis benchmark [37]. As Huang's paper only shows the synthesis results and both Vitis benchmark and Huang's design use similar algorithms, we only compare our implements with Vitis benchmark [37] on the same Xilinx FPGA platforms.

Evaluation on SuiteSparse Matrix Collection [37] dataset shows that our HiTC achieve a geomean speedup of 8.63x (up to 24.1x), compared to Vitis TC benchmark on the AMD-Xilinx Alveo U280 HBM-based FPGA. Compared to Intel MKL running on two 12-core Xeon Silver 4214 CPUs, HiTC achieves a geomean speedup of 18.6x (up to 669.8x).

1.1 Contributions of HiTC

In summary, we propose HiTC to accelerate triangle counting on HBM-equipped FPGAs via software/hardware codesign and make three major contributions. It is the first work that uses the matrix multiplication way for TC acceleration on FPGA.

1. We propose hardware-friendly reordering, tiling, and encoding techniques to address the random access issue and optimize bandwidth utilization.

2. We design streaming-based hardware accelerators on FPGAs that leverage HBM to achieve higher bandwidth and customize the computation pipeline for better computing throughput.

3. Experimental results show HiTC outperforms multi-core CPUs with an 18.6x geomean speedup (up to 669.8x), and exceeds previous FPGA accelerators in the AMD/Xilinx Vitis library, with a geomean speedup of 8.6x (up to 24.1x).

Chapter 2

Background and Related Work

This chapter covers the relevant background information needed to understand this work. Firstly we introduce the existing three different methodologies for TC in graphs and then discuss why we choose matrix multiplication-based TC as the baseline to accelerate on FPGA. Next, we illustrate the mathematical derivation of how to use bitwise operations based on sparse matrix-matrix multiplications. We will discuss the previous CPU and GPU works for accelerating TC on CPU and GPU. Lastly, we discuss previous TC acceleration works on FPGAs.

2.1 Three Different Approaches for TC

Numerous methods have been proposed to count triangles, which can be divided into three categories: subgraph matching approach, set-intersection approach, and matrix multiplication methods. All of these algorithms have their advantages and are suitable for different graph structures and hardware platforms, making them important for accelerating triangle counting tasks in various applications.

2.1.1 Subgraph Matching to a Triangle Pattern

Subgraph matching involves finding all instances of a specific pattern, represented by a query graph, within a larger target graph [40]. This problem is utilized in various applications, with triangle counting being one example where the pattern corresponds to a triangle. The complexity of computing subgraph matching is classified as NP-complete [4]. The primary method to solve this problem is through a brute-force search, which exhaustively examines all subgraphs. Ullmann et al. [31] initially introduced a backtracking technique for subgraph matching, which explores different search sequences, pruning strategies, and neighborhood indices. Subsequent research efforts have focused on enhancing Ullmann's method, with approaches such as SPath [41] and GraphQL [15] offering further refinements.

Using the subgraph matching approach for triangle counting comes with several limitations. In this context, the computational complexity can be substantial due to the necessity

of evaluating all possible node and edge combinations in the graph. Wang et al. [32] examine the optimization effectiveness of subgraph matching for triangle counting, focusing on the filtering-and-joining procedure. This approach involves enumerating all triangles, which demands significant memory during the joining phase. Although optimizations have enhanced performance compared to earlier algorithms, subgraph matching tends to be slower than intersection-based methods, except for specific datasets resembling meshes with numerous leaf nodes that can be filtered out in the initial phase. In this next subsection, we will introduce the set-intersection approach.

2.1.2 Set-Intersection Approach

The set-intersection approach for triangle counting is a method used to identify triangles within a graph without self-edges and duplicate edges. This method involves performing set operations on the adjacency lists of vertices. Assuming the graph is stored in an edge list file, the first step is to generate the adjacency list structure, where each vertex in the graph has a corresponding list containing its neighbors. Once the adjacency list is generated, the algorithm iterates through each vertex v in the graph and considers its neighboring vertices. If u are neighbors of v, then the algorithm computes the intersection of their adjacency lists of u and v. The total number of triangles for each node is counted by summing the number of intersections across all vertices. For instance, consider nodes A and B as neighbors. The set of neighbors of node A is $\{B, D, E, F\}$, and the set of neighbors of node B is $\{A, C, E, F, G\}$. The intersection of these sets is $\{E, F\}$, which are the common neighbors of nodes A and B. Therefore, there are two triangles $\{A, B, E\}$ and $\{A, B, F\}$.

The set-intersection approach tends to be more efficient for very sparse graphs where the average vertex degree is relatively low so that the computation cost of set intersection operations is lower compared with more dense graphs. The set intersection approach for triangle counting is well-suited for parallel access on both GPUs and CPUs, because it involves performing set intersection operations independently on each vertex's adjacent list. However, this approach is challenging for FPGA implementation due to several factors: Firstly, the memory access pattern on adjacent lists is very random and not continuous, which needs to randomly jump from one vertex to another vertex to find the intersection of two adjacent lists. If the adjacent list is stored in the off-chip memory of FPGA, it causes inefficient burst reading from off-chip memory. Moreover, FPGA has limited on-chip buffers which cannot store the entire adjacent list on-chip.

2.1.3 Matrix Multiplication-based Approach

The matrix multiplication-based approach for TC is a class of algorithms that uses the adjacency matrix of the graph to perform matrix operations for TC. Many solutions employed a linear algebraic computation model. Table 2.1 lists the notation symbols used in this thesis. Let A be the symmetric adjacency matrix representation of an undirected graph $G(V, E)$.

Table 2.1: Notation used in this thesis

Symbol	Description
$G(V, E)$	An undirected graph G with vertex V and edge sets E
NZ	Non-zero
NE	Non-empty
nnz	Number of non-zero elements
A	Adjacency matrix of graph G
U	Upper triangle part of adjacency matrix A
L	Lower triangle part of adjacency matrix A
$\|set\|$	Length of *set* array
n_{cuts}	total number of cuts in n dimension of a matrix
$\odot$	Element-wise multiplication

The naive approach is:

$$TC(G) = \frac{1}{6} \sum diag^{-1}(A \times A \times A) \tag{2.1}$$

Equation 2.1 does matrix multiplication three times with the same A matrix. The total number of triangles equals to the sum of all the nnz elements in the diagonal region of the A^3 matrix. Azad, et al. [1] further improve this method by utilizing the mask to reduce the complexity of computation, as shown in Equation 2.2. L is the lower triangular part of A, and U is the upper triangular part of A.

$$TC(G) = \frac{1}{2}\text{nnz}(A \cap (L \times U)) = \frac{1}{2} \sum_{A[i][j]=1} L \times U \tag{2.2}$$

In the Azad algorithm, the element-wise matrix multiplication operation filters out all the wedges that are not connected by edges in the graph, thus removing non-triangular formations. A further improvement can be proposed by Sandia [36] which replaces U by L in the sparse matrix matrix multiplication. This modification introduces a condition where the resulting matrix from the multiplication of L with itself is nonzero only if v1 > v2. This constraint reduces the number of wedges stored in C. As a result, there is typically a decrease in the number of operations and runtime. Therefore, in this thesis, we will mainly introduce Sandia algorithm [36].

$A[i][j] \in \{0, 1\}$ indicates whether there is an edge between vertices i and j. $A^2[i][j]$ shows the number of paths that start from i to j using two steps. If there exists an edge between vertex i and vertex j, and there is also a path of length two from i to j passing through an intermediate vertex k, then vertices i, j, and k together form a triangle, and the symbol $\bigcap$ defines element-wise multiplication.

Let matrix U be the upper triangle part of the adjacency matrix A, with all zeros on the diagonal. The TC of the graph G is given by

$$TC(G) = \text{nnz}(U \cap U^2) = \sum_{U[i][j]=1} U^2[i][j] \tag{2.3}$$

where nnz stands for the number of nonzero elements.

It is noted in [33] that since matrix A is a sparse binary matrix, the multiplication operation in U^2 can be implemented by AND operation in hardware, and the summation operation becomes a bit counter (BitCount), as shown below.

$$\begin{aligned} U^2[i][j] &= \sum_{k=0}^{n} \text{AND}(U[i][k], U[k][j]) \\ &= \text{BitCount}(\text{AND}(U[i][*], U[*][j]^T)) \end{aligned} \tag{2.4}$$

$$\begin{aligned} TC(G) &= \text{BitCount}(\text{AND}(U[i][*], U[*][j]^T)) \\ &\quad \text{in which } U[i][j] = 1 \end{aligned} \tag{2.5}$$

After considering the three approaches discussed above, we have decided to use the matrix multiplication method to solve the TC problem for several reasons. Firstly, this approach can effectively utilize the bitwise operations inherent in the TC algorithm, making it well-suited for FPGA acceleration. Secondly, matrix multiplication inherently allows for a high degree of parallelism on FPGAs, further enhancing computational speed. Thirdly, this approach is scalable for large graphs by tilling the matrix and processing it in blocks. It is worth noting that the FPGA acceleration of the matrix multiplication method for TC has not been extensively studied, leaving ample room for development in this area.

2.1.4 Four Different Ways of Computing Sparse-sparse Matrix Multiplication

The above section shows that matrix multiplication-based TC is a variant of sparse matrix-matrix multiplication (SpGEMM) with extra elementwise multiplication applied after calculating the SpGEMM. The sparse matrix are binary matrix and asymmetric through the diagonal. If we focus on the SpGEMM part, there are four different ways of performing the computation [29], including the inner product approach, outer product approach, row-wise product approach, and column-wise product approach, as shown in Figure 2.1. This figure illustrates four different possible ways for matrix A (orange) x matrix B (green) = matrix C (blue). All the nonzero elements of each matrix are indicated by different colors.

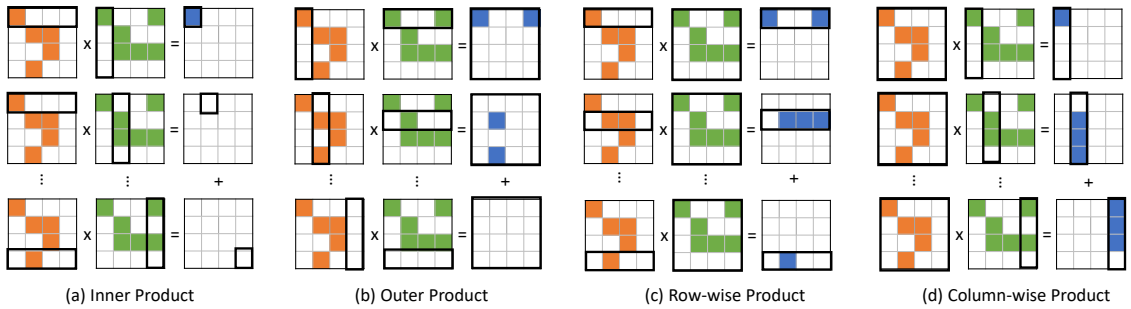

Figure 2.1: Four different ways of computing SpGEMM.

The inner product-based matrix multiplication is defined by the following equation.

$$C[i,j] = \sum_{k=0}^{N} A[i,k] \times B[k,j] \tag{2.6}$$

It is a method used to compute the elements of the resulting matrix C by taking the inner product of corresponding rows and columns of matrices A and B. Figure 2.1 shows the step-by-step computation order, where in this example $N = 3$ (based 0). Each element $C[i,j]$ of the result matrix C is the inner product of the i^{th} row of the matrix A and the j^{th} column of matrix B, which is the sum of the products of corresponding elements of the row and column. The operations contain index matching and multiply-accumulate operations to compute each element of the resulting matrix.

To achieve better memory performance, the input matrices must be stored in different formats, requiring A to be in row-major and B to be in column-major order. It computes each element of the output matrix individually, resulting in significant wasted computation when the output matrix is sparse. For example in Figure 2.1(a), when multiplying the second row of matrix A with the second column of matrix B, there is no matched index; hence the output is zero. When the input matrix is sparse, the inner product is inefficient due to the need for unnecessary index matching.

The outer product of matrix multiplication is represented by Equation 2.7.

$$C[:,:] = \sum_{k=0}^{N} A[:,k] \times B[k,:] \tag{2.7}$$

Each time, one column of matrix A multiplies with one row of matrix B, which produces partial results for the entire output matrix C. This is repeated for all elements of matrix C. A simple example can be found in Figure 2.1(b). However, the input matrices A and B have to be stored in different formats. This time A is stored in column-major, and B is stored in row-major. Also, this approach needs to buffer the entire matrix C to accumulate the partial sum of the matrix C. It might cause a read-after-write conflict in parallel computing

where different processing elements want to update the same address of the matrix C, which brings challenges to efficient parallelism.

The row-wise product approach for matrix multiplication is shown by Equation 2.8.

$$C[i,:] = \sum_{k=0}^{N} A[i,k] \times B[k,:] \tag{2.8}$$

where C[i,:] represents i^{th} row of matrix C. Each element of the i^{th} row of A multiplies with the entire B matrix to get the partial result of the i^{th} row of matrix C. After going through all the elements inside the i^{th} row of A, we need to accumulate all the partial results of i^{th} row of C. Repeat this process for all rows of matrix A, calculating the inner product for each row and each row of matrix B.

The row-wise product approach is also recognized as Gustavson's algorithm [14], involving the multiplication of non-zero elements in a row of matrix A with the corresponding non-zero entries in matrix B. Here, the row index in matrix B is determined by the column index of the non-zero value in matrix A.

There are several advantages to using the row-wise approach for TC, firstly all three matrices A, B, and C can be stored in a consistent format. According to the TC Equation 2.3, the input three matrices U can all be stored in CSR format, saving time for format conversion. Secondly, multiple processing elements (PEs) can compute different rows of the sparse output matrix individually, eliminating the need to synchronize reads and writes to the output memory. Finally, the computation pattern of the row-wise product is friendly to modify into TC design. The detailed computation process will be shown in Chapter 3.

The column-wise product approach for matrix multiplication is similar to row-wise, but in this case, all three matrices are accessed in column-major order. This is defined by Equation 2.9.

$$C[:,j] = \sum_{k=0}^{N} A[:,k] \times B[k,j] \tag{2.9}$$

Each column C[:,j] of the resulting matrix C calculates the inner product of the j^{th} column of matrix B and each column of matrix A. Comparing row-wise product with column-wise product, both of them support consistent input matrix format, one reuses B matrix and the other reuses A matrix. There is not too much difference, so in this thesis, we choose a row-wise product for sparse-sparse matrix multiplication.

2.1.5 Challenges to Accelerate TC

Accelerating triangle counting presents several challenges. Firstly, the triangle counting problem has a high memory-to-computation ratio because storing the graph structure con-

sumes a significant amount of memory. It is critical to devise an efficient compression format for storing the sparse matrix that allows for parallel access. Secondly, leveraging parallel processing capabilities is essential for accelerating triangle counting. Finally, ensuring scalability to handle graphs that exceed the capacity of a single machine is a significant challenge.

2.2 Compression Format for Sparse Matrix

This thesis focuses on the matrix multiplication method for TC. Storing matrices in a dense pattern leads to many unnecessary calculations and redundant storage [11]. This issue is particularly noticeable when dealing with the upper triangle part of a sparse matrix, where about half of the matrix is empty. To address this problem, a common approach is to use compression formats to store sparse matrices. There are several sparse matrix formats, with the compressed sparse row (CSR) [12] and compressed sparse column (CSC) formats being widely used. Both formats aim to efficiently represent sparse matrices by storing only the nonzero elements and their respective row and column indices. The main difference between CSR and CSC lies in how they store the column or row indices: CSR stores the nonzeros in row-major order, while CSC stores them in column-major order. Another format, the coordinate format (COO) [23], is a straightforward approach that pairs each nonzero element with its row and column index. Other compression formats, such as compressed diagonal storage (CDS) [8], skyline storage (SKS) [34], and diagonal (DIAG) formats, are efficient for specific distribution patterns. For example, the DIAG format is designed for sparse matrices where the majority of nonzero elements are in the diagonal region.

However, using the CSR format for HBM-equipped FPGA accelerations presents several challenges. Firstly, the CSR format is inefficient in terms of HBM bandwidth utilization. This format stores a binary sparse matrix using two arrays: a row pointer array and a column index array. Accessing non-zero elements requires the accelerator to first access the row pointer array. Additionally, the continuous storage of non-zero elements hinders cross-row vectorized accesses. The interaction between these arrays complicates continuous memory access and prevents full streaming access to the non-zero elements. Secondly, fully utilizing the port width becomes a challenge. The data bit-width in CSR format is usually 32-bit, whereas HBM can support larger port widths. Finally, how to tile the sparse matrix in the CSR format becomes a problem.

2.3 Related CPU and GPU Works

Table 2.2 lists several representative accelerator implementations on different accelerator platforms including CPU, GPU and FPGA.

Table 2.2: Related Works in CPU, GPU and FPGA

	CPU			GPU			FPGA
	TCM [DAC'18]	kkTri [HPEC'17]	TC-Cilk [HPEC'17]	HPETC [TPDS'17]	bbTC [TPDS'22]	Wang's [HPGP'18]	Huang's [HPEC'18]
Algorithmic properties							
Intersect method	✓			✓			✓
Matrix multiplication		✓	✓	✓	✓		
Subgraph matching						✓	
Optimizations							
Vertex Ordering	✓	✓		✓	✓		
Compression	✓	✓				✓	✓
Parallelization strategy							
1D	✓	✓	✓	✓			
2D					✓		

2.3.1 Previous TC Acceleration Works on CPU

Several techniques have been proposed to accelerate triangle counting on CPUs, aiming to improve performance and efficiency. These techniques include multi-threading, SIMD (Single Instruction, Multiple Data) vectorization, and task parallelism. Table 2.2 lists three representative CPU implementations, including TCM, kkTri, and TC-Cilk. TCM [7], uses a matrix multiplication-based approach for triangle counting. It employs row-wise partitions on the input sparse matrix to enable parallel processing of sub-matrices, achieving coarse-grained parallelization. However, a drawback of this approach is that different sub-matrices may access the entire graph during execution, requiring multiple processor cores to share the same memory space.

Julian [25] develops algorithms that leverage a multicore system's parallel capabilities through dynamic multithreading and are fine-tuned for memory hierarchy efficiency by employing cache-oblivious techniques.

Another approach, described in [22], utilizes fast vector instruction implementations of set operation-based algorithms to directly compute the exact triangle count. This method leverages SIMD vectorization to improve performance. kkTri [36] focuses on triangle counting on a single compute node, using linear algebra techniques. kkTri [36] uses sparse hashmap based accumulators, which can significantly improve the cache locality. In this case, the hashmap is used to quickly determine if three vertices form a triangle by checking for connecting edges between them. This approach evolved from their previous work on the miniTri application [35], which specifically targets triangles in graphs. kkTri leverages Kokkos Kernels, a C++ performance portability programming ecosystem, to adapt the implementation on multicore architectures. It offers advantages over TCM [7], particularly in its use of compression. TC-cilk [39] is another implementation that uses matrix multiplication based approach.

By efficiently utilizing the multiple cores available in modern CPUs, these techniques aim to distribute the workload of triangle counting across multiple processing units, thereby reducing computation time.

2.3.2 Previous TC Acceleration Works on GPU

TC, a foundational task in graph processing, has received significant attention regarding GPU acceleration. Various techniques are used to optimize performance in this field. Previous works can be classified into four classes based on their algorithmic properties: list intersection, map intersection, search intersection, and matrix multiplication approach. Table 2.2 lists three representative GPU implementations that use those three different approaches.

In the list intersection approach, several optimization implementations have been proposed, such as those in TCM [7], HPETC [3]. These implementations utilize set intersections between adjacent lists, which makes them cache-friendly and can utilize parallel processing efficiently. [7] is a multicore triangle counting algorithm implementation that demonstrates improved scalability in parallel environments by employing a list-based intersection approach. The computation of the list-based intersection algorithm can be illustrated with a simple example: given two adjacent lists A and B, each element in A needs to be compared with all elements in B, resulting in a time complexity of $O(n^2)$. This method contrasts with the use of dense hashmaps, which can lead to inefficient memory utilization. HPETC [3] takes advantage of a set intersection operation along with matrix multiplication implemented relying on bitmaps and on atomic operations.

The map intersection approach uses hash maps to represent the adjacent lists of vertices. Compared with list intersection approach, it allows for faster lookup of the neighbor list but it has a high memory requirement. bbTC [38] leverage the characteristics of both list intersection approach and map intersect approach by employing a hybrid approach to solve triangle counting. It uses a list-intersect algorithm for low-degree vertices and a hash map algorithm for high-degree vertices. Besides that Tom et al. [30] implement the map-based algorithm using GraphMat, a parallel and distributed graph processing framework.

In the search intersection approach, a common technique is to use binary search to achieve efficient parallelism on GPUs. Existed works include TriX [16] for multiple GPUs and TC-stream for a single GPU [17]. These implementations are designed for large-scale graphs. TC-stream proposes a parallel vertex approach with 1D partitioning and vertex reordering to address the workload balance problem. On the other hand, TriX utilizes a 2D partitioning strategy to evenly distribute the workload among multiple GPUs.

The matrix multiplication approach utilizes an adjacency matrix along with matrix multiplication operations to count triangles. In this case, the adjacency matrix is binary, simplifying the multiplication operation to the intersection operation. Thus intersection can be implemented using the previous three approaches.

2.4 Challenges for Accelerating TC on FPGAs

FPGAs possess greater flexibility than GPUs, while consuming less power, making them well-suited for high-performance applications that require energy efficiency [18]. Moreover, FPGAs possess greater flexibility than GPUs, allowing them to be repurposed for other applications if necessary. However, implementing a TC accelerator on an FPGA is non-trivial and poses several challenges. Firstly, the TC problem has a high memory-to-computation ratio, making it essential to efficiently utilize the HBM bandwidth. Secondly, the nature of sparse graphs leads to irregular memory access. Thirdly, FPGAs have limited on-chip buffer, so dividing the workload effectively is critical. Lastly, the CSR format, as a commonly used compression format, is inefficient with HBM.

2.4.1 Previous TC Acceleration Works on FPGAs

In [18], the authors present an edge-based set intersection way for acceleration TC on FPGAs. 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. The main computation of this method is to count the number of matching elements between two adjacent lists, where the adjacent lists can be seen as one row of the adjacency matrix. However, to compare the column indices between two rows requires a 32-bit input MUX in hardware, which is too expensive to represent an one-bit nonzero element in the adjacency matrix. Vitis libraries benchmark also using edge-based set intersection way for TC. Both of them are more efficient for ultra-sparse datasets (density $< 0.00001\%$) but not efficient for datasets with a higher density. In this thesis, we propose a design of streaming-based hardware accelerators on FPGAs that leverage HBM to achieve higher bandwidth and customize the computation pipeline for better computing throughput.

Chapter 3

HiTC Design

3.1 Overall End-to-end TC Accelerator Workflow

Figure 3.1 shows the overview of our HiTC accelerator design on CPU+FPGA computing system, which consists of two parts: the software part and the hardware part.

In the software part, the user needs to input the edge list file of a graph, the graph can be an undirected graph. We first convert the edge list into a sparse binary adjacency matrix stored in a compressed sparse row (CSR) format. After that, a graph reordering algorithm, called minimum degree order (MDO), is performed. The MDO algorithm originated from a technique initially proposed by Markowitz in 1959 for non-symmetric linear programming problems [21]. MDO helps to gather most of the nonzero elements into the right corner of the sparse matrix, as shown in Fig. 3.2. Based on the TC algorithm that we discussed in Section 2.1.2, only the data in the upper triangle part of the matrix needs to be used for TC.

Next, three duplicated matrices A, B, and C, are created as the input to the FPGA kernel. To save the limited on-chip buffer, we apply a sparsity-aware tiling technique on three input matrices based on the parameters defined by the user including maximum tile depth, maximum tile width, hardware buffer depth, and hardware buffer width. The next step is to do data encoding of the CSR matrix so that data is stored in a streaming fashion. To fully utilize the bandwidth (512 bits), we pack 32 elements of 16-bit data into one packet and transfer them through the HBM channels.

In the hardware part, we develop two designs: a simple comparator-based hardware design, which will be explained in Sec. 3.3, and a more efficient lookup-table based hardware design, explained in Sec. 3.4.

3.2 Hardware-friendly Software Preprocessing

In this section, we will introduce all the techniques used in software preprocessing, including graph reordering, data compressing, data encoding, and data packing. There are several

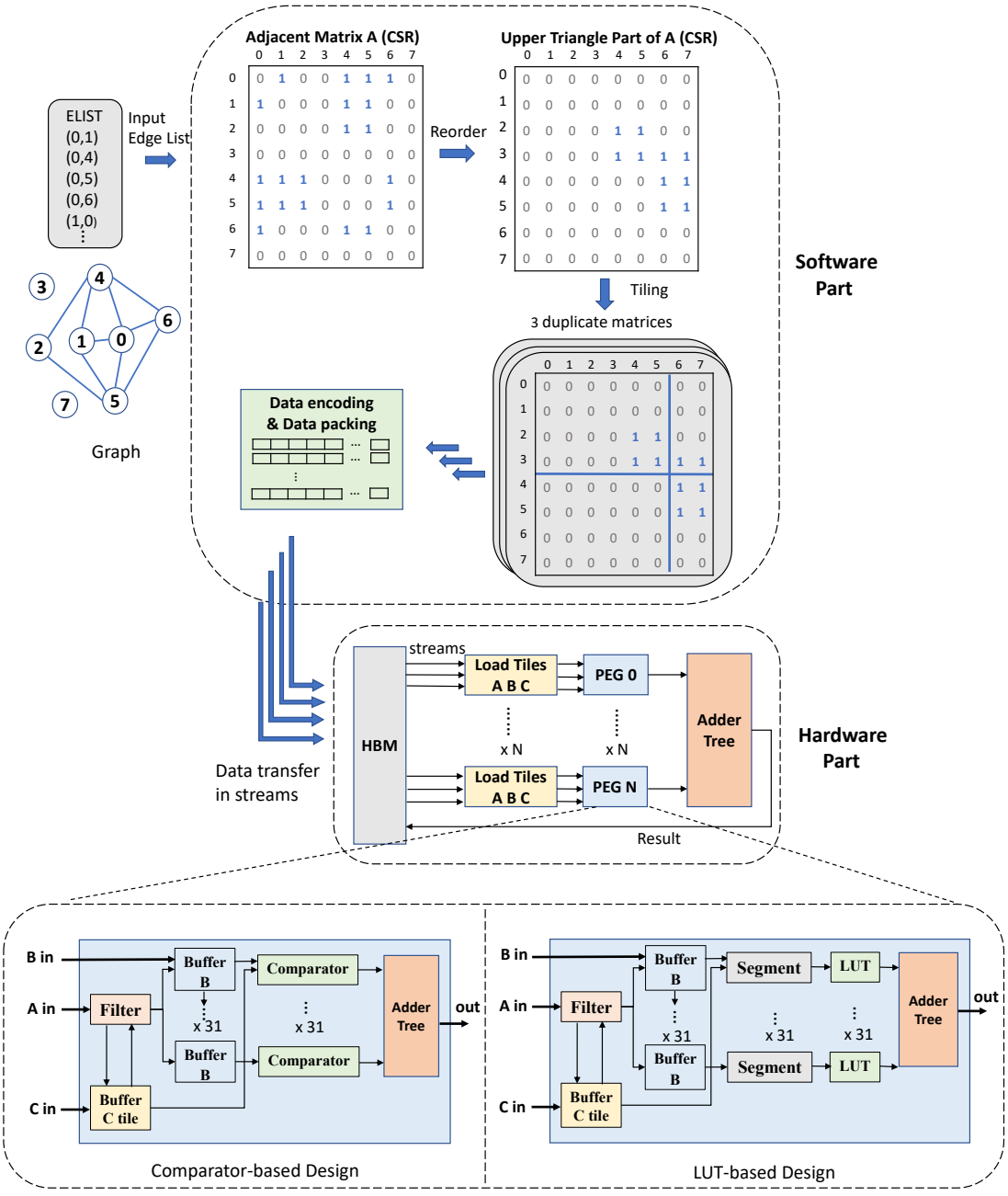

Figure 3.1: The overview of the proposed HiTC workflow.

reasons why we need software preprocessing. Firstly, the adjacency matrix generated by the original graph is too sparse with poor data locality. We need a graph reordering technique to rearrange the vertices of the graph in a way that improves the data locality of the adjacency matrix. Secondly, the conventional CSR format is not HBM-friendly. We need a customized compressed format based on the CSR to make it more suitable for hardware design, for example, by minimizing data communication overhead between streaming modules and external interfaces. Meanwhile, we propose data encoding and packing techniques to improve bandwidth optimization.

3.2.1 Graph Reordering Algorithm

Graph reordering, also known as graph permutation or graph ordering, is the process of rearranging the vertices of a graph systematically to improve data locality and minimize the edge transformation during graph processing applications [2]. The goal of graph reordering is to rename the node order without changing the graph structure. One of the major challenges for accelerating matrix-multiplication-based TC is that the adjacency matrix with an irregular memory access pattern exhibits poor data locality. After analyzing the nonzero distribution of the input graphs used for TC problem, we notice most of the nonzero elements are scattered among the adjacency matrix, as shown in Figure 3.2(a). However, the dispersed distribution is inefficient for data locality and will cause many edge transverse. After investigating the existing graph processing algorithms, we focus on two reordering

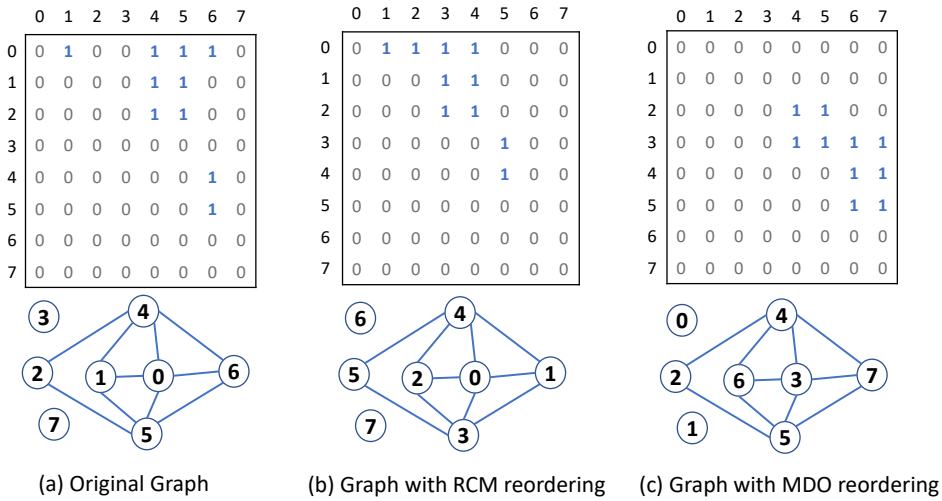

Figure 3.2: Graph reordering techniques: Reverse Cuthill-McKee (RCM) vs. Minimum Degree Order (MDO).

Reverse Cuthill-McKee (RCM) Ordering

The Reverse Cuthill-McKee (RCM) algorithm, introduced by George [9], is a variant of the Cuthill-McKee (CM) ordering [5], which aims at reducing the bandwidth of a sparse symmetric matrix A. The reason for us to choose RCM ordering instead of CM ordering is that it was observed in [24] that by reversing the CM ordering, we can reduce the amount of fill-in for many graphs.

RCM ordering is commonly used to reduce the bandwidth of sparse matrices. Note bandwidth in linear algebra means different in hardware. In the context of graph theory, the bandwidth of a matrix is defined as the maximum distance between any non-zero element and the main diagonal. We want the bandwidth to be as small as possible, which means the graph has a more clustered structure. For example, the original adjacency matrix in 3.2(a) has a large bandwidth. After the RCM reordering, all the non-zero elements of the new adjacency matrix are close to the diagonal region, as shown in Figure 3.2 (b). As the adjacency matrix is symmetric, we only consider the upper triangle part of the sparse matrix.

Algorithm 1: Cuthill-McKee Ordering

Input: Undirected graph $G = (V, E)$

Output: Ordered level sets S_i

```
1:  Choose starting node s //for each connected component;
2:  S_1 ← {s}, mark s;
3:  i = 1;
4:  while S_i ≠ ∅ do
5:      S_{i+1} ← ∅;
6:      //in order of increasing degree;
7:      for each u ∈ S_i do
8:          //in order of increasing degree;
9:          for each unmarked v adjacent to u do
10:             S_{i+1} ← S_{i+1} ∪ {v};
11:             mark v;
12:         end
13:     end
14:     i = i + 1 //move on to the next level set;
15: end
```

Algorithm 1 shows the pseudo code for the Cuthill-McKee (CM) algorithm. Once we get the net node ID for CM order, the RCM order can be obtained by reversing the CM order, as shown in Equation 3.1.

$$node_i^{RCM} = node_{n-i+1}^{CM} \quad \text{for} \quad i = 1, \ldots, n \tag{3.1}$$

In general, the RCM has the same envelope as CM but better-observed behavior in practice.

Minimum Degree Ordering (MDO)

The MDO algorithm[21], as shown in Algorithm 2 is another graph ordering algorithm commonly used in the context of sparse matrix computations. As shown in Figure 3.2, MDO reordering helps to gather most of the non-zero elements into the right corner of the adjacency matrix by renaming the node ID of the graph. First, it reorders the nodes based on their degrees, where the degree of a node represents the number of edges incident to that node. Line 4 shows how to calculate the corresponding degree of a node v. Once the degrees of all nodes are stored in a set $D(node, degree)$, the subsequent step involves sorting the degree set in ascending order of degree values. The final step is to rename the node ID based on the ascending order of degree value. With the above steps, the MDO algorithm effectively clusters the NZ elements into the right corner of the adjacency matrix.

Algorithm 2: Minimum Degree Ordering

Input: Undirected graph $G = (V, E)$ in CSR format

Output: Ordered level sets S_i

1: Create sets of node degrees D(node, degree);
2: Calculate the degree of each node in the original matrix;
3: **for** *each* $v \in V$ **do**
4: | degree = G.offset[v+1] - G.offset[v];
5: | $D_v \leftarrow (v, degree)$;
6: **end**
7: sort(D) //sort degrees in increasing order ;
8: **for** *each* $v \in V$ **do**
9: | Get the node ID from the degree sets;
10: | $S_v \leftarrow S_v \cup \{D[v].first\}$;
11: **end**

3.2.2 Hardware-friendly Tiling Technique

According to Figure 3.1, the inputs of the FPGA kernel contain three sparse matrices. Figure 3.3 shows an example of three input matrices A, B, and C. All three matrices have the same content, which represents the upper triangle part of an adjacency matrix. However, the three matrices play different roles. Mathematically, the computation order is: First, matrix A is multiplied by matrix B to get the intermediate matrix, called AB[M][N]. Then element-wise multiplication between matrix AB and matrix C is performed (C[M][N] $\odot$ AB[M][N] = result[M][N]). Finally, the number of TC is the sum of all non-empty (NE) elements of the matrix result[M][N].

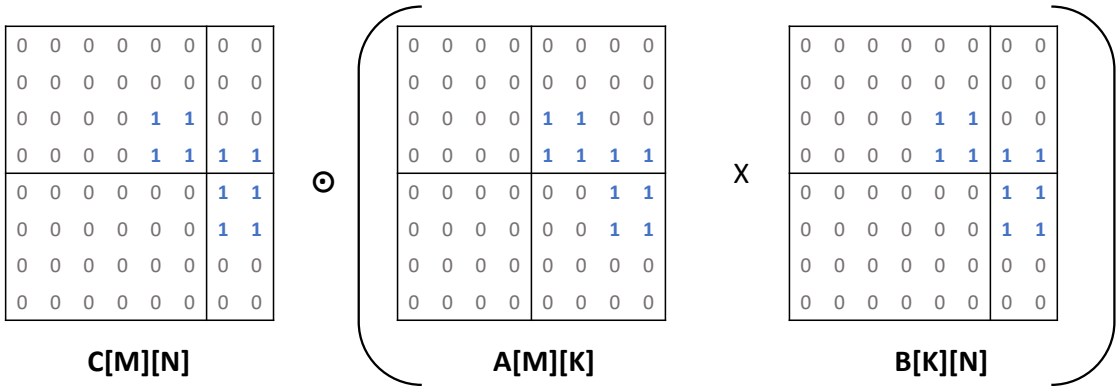

Figure 3.3: HiTC computation order

Due to the limited FPGA resource, we apply a hardware-friendly tiling technique to divide all the 3 input matrices into sub-matrices. Each time the kernel processes data tile by tile and accumulates the partial results of TC. The detailed computation order cross tiles and within tile-level order will be discussed later in this chapter.

Due to the limited on-chip buffer resource and regular hardware requirements in HiTC design, we propose a hardware-friendly tiling scheme to address those problems. The relationship between graphs and sparse matrices means that vertex-level tiling corresponds to row-based (1D) tiling, while edge-level tiling corresponds to nonzero-based (2D) partitioning [38]. However, the existing partition technique used for TC is not friendly for hardware. The matrix multiplication-based TC can be seen as a variant of SpGEMM, which contains data reused from one input sparse matrix.

According to Figure 3.3, we need to reuse matrices B and C during the computation. However, buffering sparse matrix on-chip is nontrivial. Traditional 2D tiling for matrices uses fixed tile shape, which is used for dense matrices. For sparse matrices, we want only to store the NE elements on-chip and skip the zeros. Since we cannot predict how many NE elements are inside each tile, tiling with fixed tile size cannot be used for a sparse matrix.

Thus, we propose a hardware-friendly tiling technique for sparse matrices. The new tiling scheme depends on four constraints:

1. Hardware buffer width
2. Hardware buffer depth
3. Maximum tile width
4. Maximum tile depth.

The values of these four parameters depend on the FPGA resources and can be changed by the user.

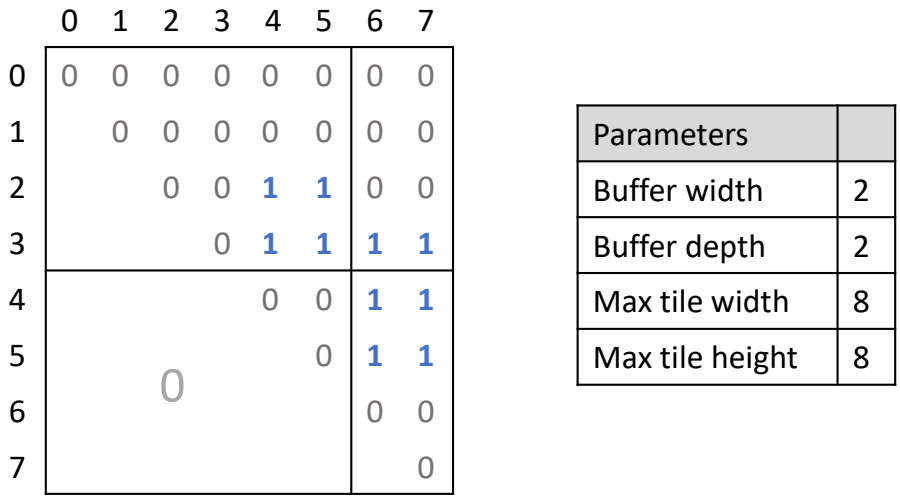

Figure 3.4: An example of tiling a sparse matrix

Figure 3.4 illustrates a simple example. In this example, we set the buffer size as 2×2, which can store at most 4 non-zero elements. Thus, we need to ensure that for each tile after tiling, the non-zero elements in each sub-matrix should not exceed 4. Therefore, finding suitable cutting positions becomes the key problem that needs to be addressed.

Firstly, in the horizontal direction of the sparse matrix, we go through the matrix row by row and count the number of non-empty rows starting from row 0. Then we split rows into tiles when the number of NE rows meets the depth limit. We repeat this process until it reaches the last row of the sparse matrix.

Conceptually, to cut in the vertical direction, we need to transpose the CSR format into CSC format first to go through data in column-major. As the original adjacency matrix is symmetric to the diagonal, the content of CSR and CSC are the same in this case. While iterating column by column, we use a dictionary to store the pair (row index: accumulated # columns). If the current column contains any row with accumulated # columns $>$ buffer width, we cut the tile from the current column and reset the dictionary.

An example can be seen in Figure 3.6. Suppose we iterate the matrix from the 0^{th} to the 4^{th} column, the dictionary store in pairs (row index: accumulated #columns), $\{(2:1),(3:1)\}$, meaning there exist two non-zeros in 4^{th} column located in 2^{th} and 3^{th} row respectively. Next, in the 5^{th} column, we update the dictionary to $\{(2:2),(3:2)\}$. However, in the 6^{th} column, the updated dictionary becomes $\{(2:2),(3:3),\ldots\}$, where $(3:3)$ means the 3^{rd} row contains 3 non-zero elements, which is greater than the buffer width. Thus, we need to cut between 5^{th} column and 6^{th} column.

Besides buffer depth and buffer width, we have other two constraints (max tile width, max tile height) to make sure the shape of sub-matrices after tiling are within a certain

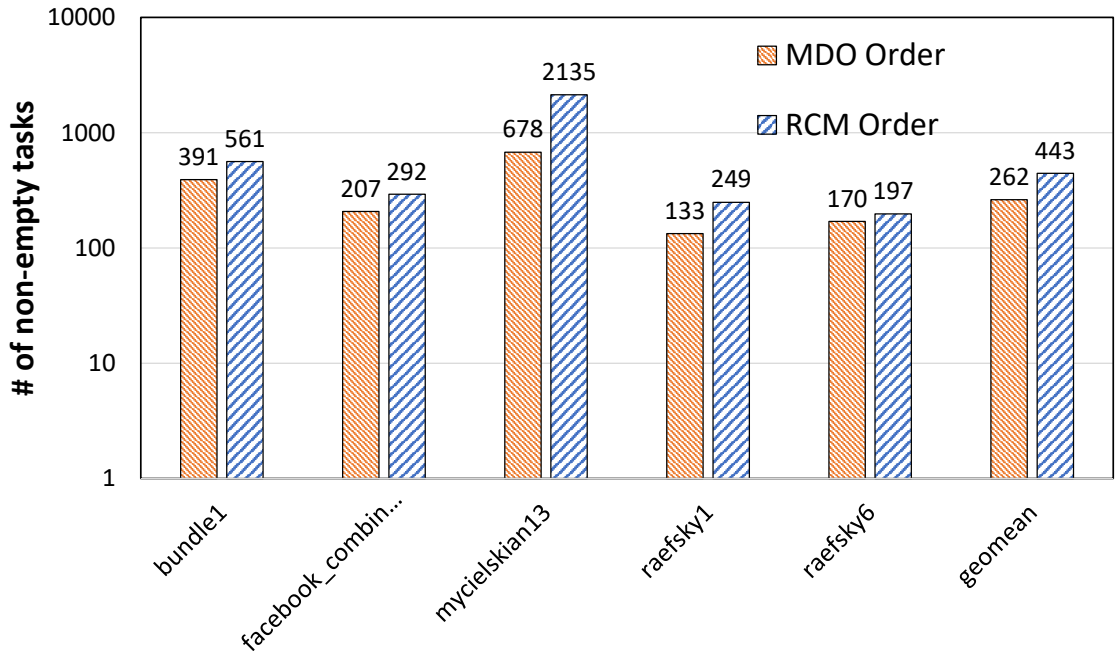

Figure 3.5: Comparison of non-empty tasks after tiling sparse matrix using RCM and MDO algorithms. A non-empty task (tile_A, tile_B, tile_C) means all three tiles are non-empty.

Figure 3.5 compares the number of non-empty tasks resulting from the two reordering algorithms, RCM and MDO, applied to the tiling of a sparse matrix. A task can be defined as a tuple (tile_A, tile_B, tile_C). According to Figure 3.3, the FPGA kernel processes tasks sequentially, and the overall kernel execution time is directly proportional to the total number of tasks. Thus, we want the number of non-empty tasks to be as low as possible. The estimated geometric mean reveals that the MDO algorithm outperforms RCM in terms of identifying and skipping empty tasks. In conclusion, the MDO reordering method is more effective in optimizing the tiling process for sparse matrices, resulting in a more compact representation with fewer non-empty tasks.

3.2.3 Sparse Matrix Format

In this thesis, we customize the traditional compressed sparse row (CSR) format to facilitate vectorized and streaming access to individual HBM channels while enabling simultaneous access to multiple channels. CSR format is a popular method for representing sparse matrices in computer science and numerical computing.

In the CSR format, the binary sparse matrix is stored in two arrays: 1. The column index array stores the column indices of the NE elements in the same order as the corresponding values in the value array. Note the input adjacency matrices are binary matrices, so it is

not necessary to store the values for the NE elements as it is always equal to 1. 2. An offset array that stores the locations in the column index array that start a row. It contains n+1 elements, where n is the number of rows in the matrix. To access the NE elements of a specific i^{th} row of the sparse matrix, we need to first locate the starting index of i^{th} row in the offset array, which indicates the position in column index arrays where the NE elements of i^{th} row start. Once we have the starting index for i^{th} row, we can iterate over the corresponding elements in the column index array to access the NE elements and their column indices for that row.

The process above contains a lot of nonconsecutive data accesses for both offset array and column index array, which prevent fully streaming accesses to the NE elements. Meanwhile, the nonconsecutive data access pattern is inefficient for off-chip memory access and will greatly reduce the performance.

To address this problem, we propose a customized compress format that supports efficient data transfer through off-chip memory in a streaming fashion.

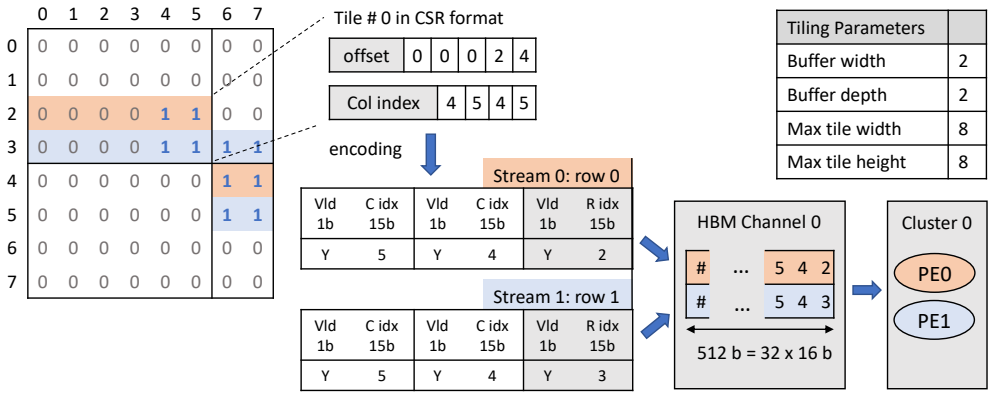

Figure 3.6: An example of the customized sparse matrix format.

Figure 3.6 illustrates an example of our sparse matrix format on one of the tiles in an 8×8 matrix. Tiling is required to handle large matrices that exceed the buffer size. Meanwhile, we can customize the data bit width used to represent the index data. This can help to reduce to memory overhead of storing the sparse matrix. The original CSR format used 32 bits to represent the index data, but we use 15 bits to represent the column index, which reduce the memory storage overhead. After tiling, the data in each tile will be stored in CSR format. To construct the column index array for a CSR matrix, iterate over each row of the original matrix. For each non-zero element in the row, record the column index of that element in the JA array. To construct the offset array, start with the value 0 at the beginning of the array. Then, for each row of the matrix, count the nnz encountered so far

and add this count to the offset array. For example, row 2 in tile 0^{th} tile contains 2 NZ, then offset[2+1] - offset[2] = 2. Then we record the column indices of those 2 NZ in the column index array, which are 4 and 5 respectively.

To minimize the data width used to represent index data, we calculate the relative position of each NE element inside the tile. The next step is to combine the offset array with the column index array into one so that there is no data communication between two separate arrays. For each row of the tile, we store the row index followed by the column indices. Each index data consists of the flag (1 bit) and index value (15 bits). If a column index has $flag == 1$, it means the column index is valid. If $flag == 0$, it means that the current column index is used for zero padding. If a row index has $flag == 1$, it means it starts a new row. If $flag == 0$, it means the previous row has not finished.

After decoding, the final step is packing the streams of elements into streams of packets. We fully utilize the HBM bandwidth that transfers 512-bit data per access to off-chip memory. Each 512-bit packet contains 1 row index (16 bits) and 31 column indices (16 bits each). In our design, the number of processing elements (PEs) per processing element group (PEG) relies on this data packet structure. Each PEG contains 31 PEs. The detailed architecture will be shown in the next section.

Figure 3.6 shows one stream of data stored in one HBM channel and is assessed by clusters of 2 PEs. The PE is scheduled in cyclic order within one tile.

In summary, the proposed customized sparse matrix format is hardware-friendly and fully utilizes the bandwidth of HBM.

3.3 Comparator-based Hardware Design

In this chapter, we introduce two streaming designs aimed at accelerating TC. As the TC application is memory-bound, we use the streaming design to save more on-chip memory. Compared with non-streaming design, the implementation effort for streaming design is more challenging. There are three input matrices for the FPGA kernel, as shown in Figure 3.3. We buffer both B tile and C tile on-chip for data reuse and access A tile in a streaming fashion.

The first design is quite straightforward, called comparator-based hardware design, which is explained in this section. The second design is an optimized design based on the first one, called lookup table-based hardware design, which will be explained in Sec. 3.4.

This section begins with an exposition on the overview architecture of the accelerator tailored to the comparator-based hardware design. Subsequently, we present the corresponding algorithm for HiTC, followed by a detailed explanation of each constituent component. This includes a buffer module and comparator design. Finally, we present an analytical model to evaluate the performance characteristics of the proposed design.

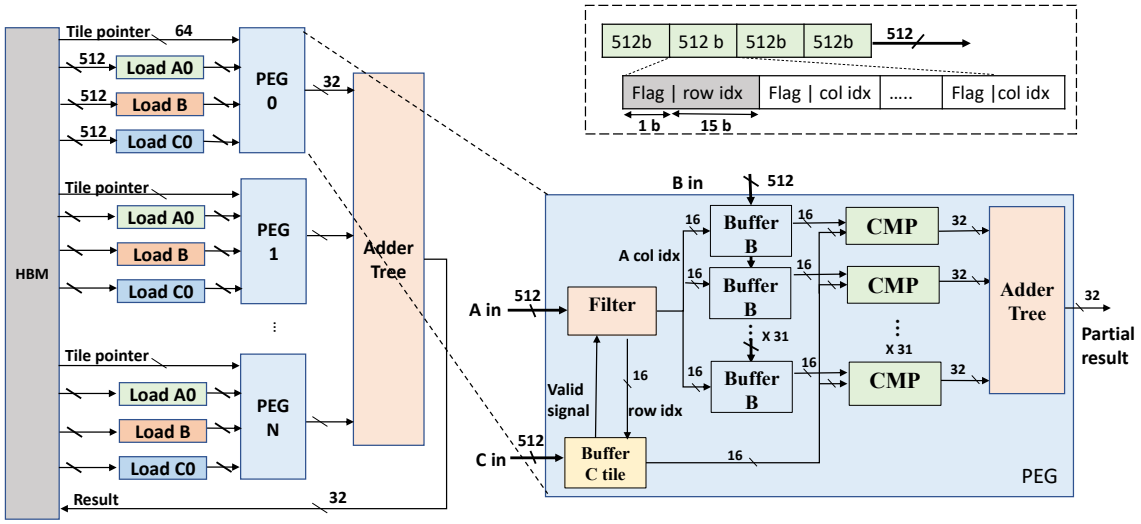

Figure 3.7: Comparator-based hardware design architecture.

HiTC makes efficient use of limited on-chip memory to exploit data reuse on the B tile and utilizes the C tile as a mask to eliminate unnecessary computation from general sparse-sparse matrix multiplication.

Figure 3.7 depicts the overview data flow architecture for comparator-based hardware design. The arrows indicate the data transfer direction, and the bit width for each stream is shown beside the arrow. Initially, the *loading* module is used to load the 512-bit input streams from HBM and stream them into *processing engine group* (PEG) through a FIFO channel. We deploy N PEG to compute block-based TC and each PEG contains 31 PEs. Since each PEG has 31 PEs, we can process i^{th} row of A $\times$ B tile in one cycle. The reason to use 31 PEs per PEG is because each element inside the input stream contains 32 $\times$ 16-bit data. One 512-bit pack consists of 1-row index with 31-column indices. Based on the computation way for the row-wise product, to calculate one row of A $\times$ B tile at one time, we need 31 PEs. Each PE contains one buffer module and one comparator which is represented as *CMP* in the architecture figure. There are 31 CMP modules run in parallel and each of them calculates the partial result. At the end, an adder tree is used to sum all the partial results generated by the 31 CMP modules and output one 32-bit value. Once all the PEGs finish processing data, we need to sum all the 6 results generated by all 6 PEGs and get the total number of triangles.

Inside one PEG, the processing includes the following. Firstly, we buffer one B tile with 31 copies and one C tile on-chip. Both the B tile and C tile use the same buffering strategy. There are 31 buffer B modules, each acting as a relay node to construct a chain-based broadcasting network. In chain-based broadcasting, data are transmitted sequentially from

one node to the next in a linear chain-like fashion. Each node in the chain receives data from the preceding node and forwards it to the subsequent node. Since all the components run in a pipelined fashion, the time overhead for transmitting data from one buffer module to another can be overlapped. After buffering both B tile and C tile, the next step is to access the *A in* data in a streaming fashion to the filter module. The filter module will output A column indices used to access B tile. Meanwhile, the filter module will output A row indices used to access B tile. After that, all the buffer modules will output column indices in a streaming fashion to the comparator module. Each comparator module is used to compare the B-column index with C column index and count the number of common numbers. The adder tree is used to add all the results generated by 31 comparator modules and output a 32-bit value as the partial result for TC.

3.3.2 Detailed Processing

The overall processing sequence can be described by the example in Figure 3.8, where only one processing engine group (PEG) exists in the architecture. Upon tiling, the FPGA kernel processes data tile by tile in a sequential manner. In this particular instance, we have three input matrices A, B, and C. Suppose the B tile and C tile are buffered on-chip already, then we input the A stream from the HBM memory.

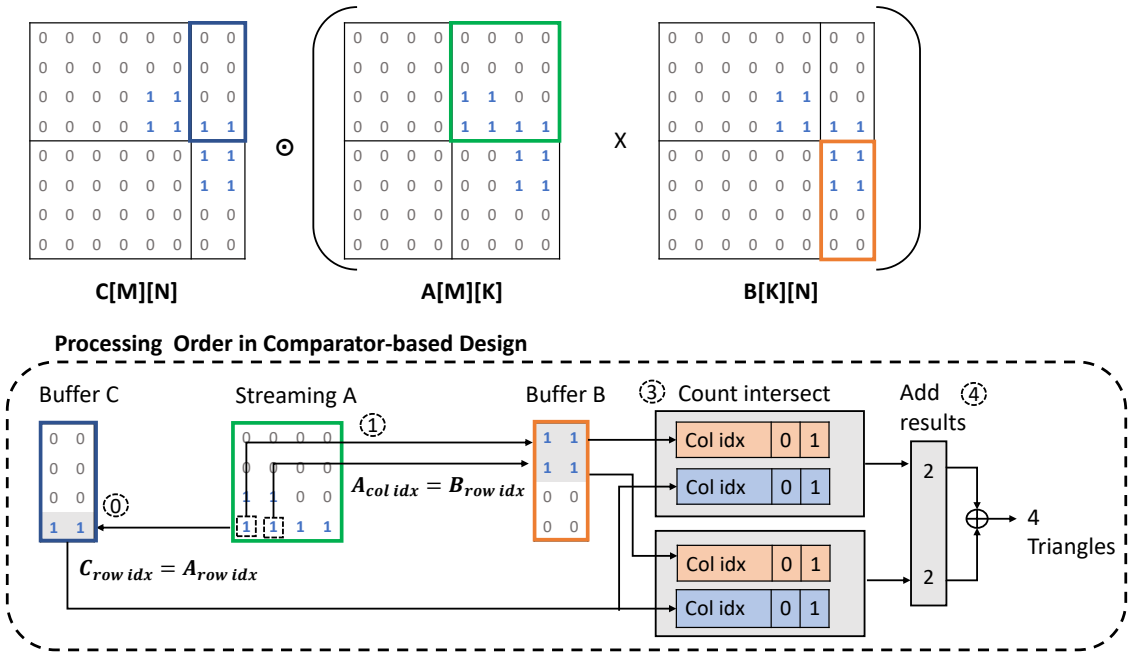

Figure 3.8: Processing order for comparator-based hardware design.

The initial step involves decoding the 1st-row packet of A and getting the current row index. Next, we check whether the corresponding row of C is non-empty. For instance, we first stream in row 2 of A, but row 2 of C is empty. We then proceed to read row 3 of A and

unpack its data. Upon accessing the C tile, we ascertain that row 3 of C contains data. We proceed to stream the column indices from the row of C into the subsequent comparator module.

After filtering the A data by the C tile, the filter module outputs all the A column indices into different buffer B modules to access the B tile. In this example, $a_{3,0}$ is utilized to locate the 0^{th} row of the B tile and retrieve its column indices (0 and 1). Concurrently, $a_{3,1}$ is employed to access the 1^{st} row of B, yielding 2 column indices (0 and 1). Regarding $a_{3,2}$, it results in an empty row in the 2^{nd} row of B, prompting us to skip it.

Upon identifying the corresponding column indices in the B buffer, the subsequent step involves comparing them with the column indices in the C tile and counting the common values. The comparator module outputs the count of common values. Finally, we sum all the output values from different comparator modules to obtain the partial result of TC.

3.3.3 Buffering Scheme

HiTC makes efficient use of limited on-chip memory to exploit data reuse. In order to solve the random distribution issue, we propose a specific tiling scheme aiming to limit the number of NE elements inside each tile. The tiling scheme is used to help with the buffer module. In this subsection, we will take a detailed look at the buffer module. In our design, both B tile and C tile use the same buffer strategy.

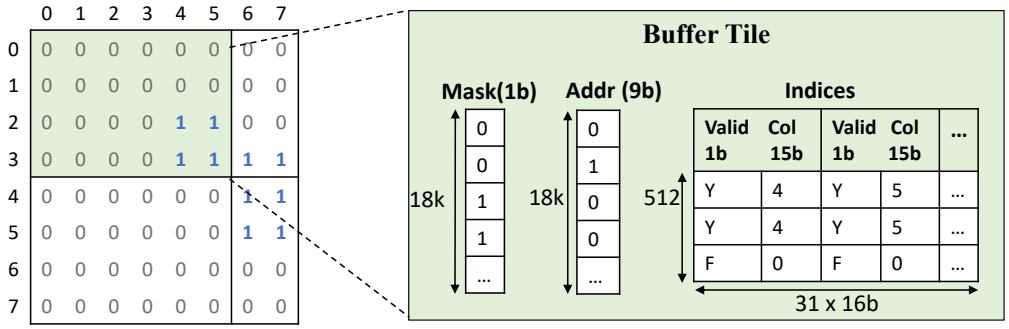

Figure 3.9: Hardware component: buffer module design.

Figure 3.9 illustrates the data structure for one buffer module, which consists of three parts: a mask table, an address array, and an index array.

To reduce the memory overhead, we only store the NE elements inside one tile. The size of the index array is determined by the HBM port width and also the BRAM bank size. As we show in the sparse matrix format, each 512-bit packet transferred through HBM channel contains 1 row index and 31 column indices. The depth of the index array relies on the 18K memory bank size of BRAM. The size of one 18K BRAM is 512×36 bits. To fully utilize the BRAM bank, we set the size of indices to be 512×496 bits, which use seven 18K BRAM banks. Each row of the index array contains 31×16 bits of data, or 496 bits in total. When

buffering data on an index array, the column indices in one row of sparse matrix can be stored on-chip in one cycle. Also, the column indices in each row of the matrix have to be stored in increasing order for the convenience of computation later.

According to Algorithm 4, we need to randomly access one row of tiles and quickly check whether this row is empty or not. Thus how to efficiently map the index becomes a crucial part of the buffer module.

To map the actual row index in the tile with the relative position in the index array, we need an address array with a shape of 18K×9 bits to record the relative address stored in the index array. The height of the address array is defined as the maximum tile height. Since we use 15 bits to represent the row and column index inside a tile, theoretically, the maximum tile height equals to 2^{15}, which is 32K. Considering the tradeoff between tile size and resource usage, we set the maximum tile height as 18K to save on-chip resources. The value inside the address array is within a range of 0 to 511, which can be represented by 9-bit data bitwidth. Since the input sparse matrix can be very sparse, each time only a few rows of the index array need to be updated. However, resetting the address array takes 18K cycles. To efficiently reset the address array, an extra mask array is used to record the validation of elements in the address array. Instead of resetting the entire address array, we only reset the mask table which only contains 0 and 1. The original size of the mask table is 18K with 1-bit data inside, where 18K is the maximum tile height. Considering the lookup table (LUT) usage, we cannot reset the 18K bits of data in one cycle. As a result, we store mask tables in the BRAM bank. To fit the shape on one BRAM bank, we reshape it to 512 with 32-bit data inside, so that it takes 512 cycles to reset the mask table, which is 32 times faster than the original mask array reset time.

3.3.4 Comparator Design

The comparator module in our architecture compares two sets of column indices from B tile and C tile, then returns the count for the number of common elements between the two sets. When analyzing different methods for counting the number of common elements between two sets, several approaches can be considered. The trivial method is called the Brute Force Method, which compares each element of one array with every element of the other array. Nevertheless, this method is not efficient, which has a time complexity of $O(n^2)$.

One of the optimized methods is using set intersection, which is friendly for streaming design, as shown in Algorithm 3. Two input sets are streaming in the comparator. It initializes two pointers, 'a' and 'b', to track the indices of sets A and B respectively and initializes a counter 'count' to zero. $|A|$ means the length of stream A. In the hardware implementation, data in the set are transferred through FIFO in a streaming fashion. The algorithm then enters a loop that continues as long as the pointers 'a' and 'b' are within the bounds of sets A and B respectively. This loop will run in pipeline fashion with initial interval (II) = 1. If both streams A and B are not empty, the algorithm compares the elements

Algorithm 3: Comparator module: Set Intersect(A, B)

Input: sets A and B
Output: Number of intersection count: *count*

```
1: //a and b use to track the index of sets A and B;
2: a ← 0; b ← 0; count ← 0;
3: while do
4:     if a < |A| and b < |B| then
5:         if A[a] = B[b] then
6:             count++; a++; b++ ;
7:         end
8:         else if A[a] < B[b] then
9:             a++;
10:        end
11:        else
12:            b++;
13:        end
14:    end
15:    else if a = |A| and b < |B| then
16:        b++;
17:    end
18:    else if a < |A| and b = |B| then
19:        a++;
20:    end
21:    else
22:        break;
23:    end
24: end
```

at indices 'a' and 'b' in sets A and B respectively. If the elements are equal, indicating a common element, the counter 'count' is incremented by one, and both pointers 'a' and 'b' are incremented to move to the next elements in sets A and B. If the element at index 'a' in set A is less than the element at index 'b' in set B, the pointer 'a' is incremented to move to the next element in set A. Similarly, if the element at index 'b' in set B is less than the element at index 'a' in set A, the pointer 'b' is incremented to move to the next element in set B. In the hardware, if one of the sets is exhausted but another set is not, it will keep reading data from the stream until both streams are empty. This while loop will stop only when both A and B reach the end of streams.

3.3.5 Algorithm

Algorithm 4 demonstrates the pseudo-code for TC accelerations in the proposed comparator-based hardware architecture.

Algorithm 4: Block-based TC with Set Intersect Approach

Input: (1) three copies of matrix U, named matrix A, B and C (2) matrix hyper-parameter m_{cuts}, n_{cuts}, k_{cuts} (3) Q pointer list

Output: number of triangles: tc

```
1:  tc ← 0;
2:  for k = 0 to k_cuts − 1 do
3:      for n = 0 to n_cuts − 1 do
4:          Buffer tiles B_kn;
5:          for m = 0 to m_cuts − 1 do
6:              Buffer tile C_mn;
7:              for p = 0 to P do
8:                  // run in parallel;
9:                  for r = Q_mk to Q_mk+1 do
10:                     stream in A_mk[i];
11:                     if C_mn[i] is not empty then
12:                         decode(C_mn[i]) //get column indices of C_mn[i];
13:                         for j = 0 to 31 do
14:                             //run in parallel;
15:                             decode(B_kn[j])//get column indices of B_kn[j];
16:                             tc ← tc + SetIntersect(C_mn[i], B_kn[j]);
17:                         end
18:                     end
19:                 end
20:             end
21:         end
22:     end
23: end
```

First, an input graph is represented as the upper triangle part of an adjacency matrix format with 3 copies named matrix A, B, and C. To support a large-scale graph, we first tile the three input matrices into sub-matrices. Since the B tiles need to buffer on-chip with multiple copies, we want to reuse the current B tile as much as possible. The computation order across the tile level can be shown in line 2-6.

After buffering B tiles and C tile on-chip, NE elements within a tile of A are streamed into P PEGs from multiple HBM channels. A distinct set of rows is usually cyclically assigned to each PE (and each HBM channel). A pointer list Q is used to track the tile position inside the stream A. Each cycle streams in 1 packet consisting of 1 row of data in A tile, as shown in Algorithm 4, line 10. The current row index i can be obtained after unpacking data.

Line 11 filters the current A row index by C tile. After filtering, we use the nnz of A tile as the index to access corresponding rows in B tile and C tile.

The last step (line 16) is to count the number of common values between $C_{mn}[i]$ and $B_{kn}[j]$ and accumulate those results. The detailed computation logic is shown in Algorithm 3. The inputs of the set intersect function are two sets of column indices from $C_{mn}[i]$ and $B_{kn}[j]$. Here we suppose column indices within 1 row of matrix are stored in increasing order. In this case, we only compare the column index for NE elements.

3.3.6 Performance Analysis and Modeling

We now analyze the performance of Algorithm 4. For better distinction, three copies of input matrices are named as matrices A, B, and C. The dimensions of the three matrices A, B, and C are $M \times K$, $K \times N$, and $M \times N$ respectively. Due to our specific tiling scheme as shown in Figure 3.3, we use m_{cuts}, n_{cuts}, and k_{cuts} to denote the number of cuts in the M, N, and K dimensions respectively. The example in Figure 3.3 contains $m_{cuts} = n_{cuts} = k_{cuts} = 2$. Meanwhile, we use m, n, and k to represent the tile indices. For example, $B_{1,2}$ means a sub-matrix of B at position $(1, 2)$.

1. In line 2-5 of Algorithm 4, we process the tiles one by one, and we only process the data when all tiles A, B and C are non-empty. In line 4 we buffer B tile on-chip. The cycle count for buffer tile B is:

$$\begin{aligned} T_{\text{bufferB}} &= T_{\text{reset row mask}} + T_{\text{buffer index array}} \\ &= [(\text{row mask height} - 1) \times \text{II} + \text{iteration latency}] \\ &+ [(\text{avg \#rows in B tile} - 1) \times \text{II} + \text{iteration latency}] \end{aligned} \tag{3.2}$$

   The buffer module consists of two separate for loops with pipeline II=1, including resetting the row mask and updating the indices array. The time to fill in data in the address array is covered inside the time to update the indices array. Since each B tile has random NE elements inside, we consider the average number of rows inside each B tile.

2. The cycle count for buffer tile C is:

$$\begin{aligned} T_{\text{bufferC}} &= T_{\text{reset row mask}} + T_{\text{buffer index array}} \\ &= [(\text{row mask height} - 1) \times \text{II} + \text{iteration latency}] \\ &+ \left[(\frac{\text{avg \#rows in C tile}}{\text{PEG}} - 1) \times \text{II} + \text{iteration latency}\right] \end{aligned} \tag{3.3}$$

   Both C tile and B tile use the same way of the buffering scheme, but the processed data are different. Unlike buffer entire tiled data like B, each C buffer only contains $\frac{1}{PEGs}$ part of C tiles.

3. The PEG region (lines 10-18) contains three parts: decoding B, decoding C, and comparing set intersection. Before computation, rows in A tile have been filtered by C tile. We use $\#rows_{A\bigcap C}\ perPEG$ to indicate the number of common rows in A tile generated after filtering:

   The cycle count for decode rows in C tile is:

$$\begin{aligned} T_{\text{decode C}} &= \#rows_{A\bigcap C}\ perPEG \times T_{\text{get c rows}} \\ &= \#rows_{A\bigcap C}\ perPEG \times [(31-1) \times \text{II} + \text{iteration latency}] \end{aligned} \tag{3.4}$$

   We use $\bigcap_{i\in R_{A\cap C}} A_{i,*} \cap R_B$ to represent the intersection of the column indices inside the rows of $\#rows_{A\bigcap C}\ perPEG$ with the rows in B tile in matrix notation. The cycle count for decode rows in B tile is:

$$\begin{aligned} T_{\text{decode B}} &= \bigcap_{i\in R_{A\cap C}} A_{i,*} \cap R_B \times T_{\text{get b rows}} \\ &= \bigcap_{i\in R_{A\cap C}} A_{i,*} \cap R_B \times [(31-1) \times \text{II} + \text{iteration latency}] \end{aligned} \tag{3.5}$$

   The cycle count for set-Intersect way of comparator is within a range. min_{cc} shows the best case, if both B stream and C steam are consumed at the same time. max_{cc} shows the worst case, if one of the streams stalls for a long time while the other stream is consuming. So the cycle count is:

$$\begin{aligned} T_{\text{comparator per tile}} &= \#rows_{A\bigcap C}\ perPEG \times \text{range}(min_{cc}, max_{cc}) \\ min_{cc} &= \max(\text{avg \#col indices per B row}, \text{avg \#col indices per C row}) \\ max_{cc} &= \text{avg \#col indices per B row} + \text{avg \#col indices per C row} \end{aligned} \tag{3.6}$$

4. The total cycle count for block-based TC-Naive Set Intersect is:

$$\begin{aligned} T_{\text{cc}} &= \text{Number of valid}(tileA, tileB, tileC) \\ &\quad \times [\max(T_{\text{bufferB}}, T_{\text{bufferC}}) + \max(T_{\text{decodeB}}, T_{\text{decodeC}}, T_{\text{comparator}})] \end{aligned} \tag{3.7}$$

   Overall, the FPGA kernel processes tasks sequentially, where a valid task contains non-empty tile A, non-empty tile B, and non-empty tile C. Inside each task, both B tile and C tile are buffered currently, so we consider the maximum time of them. After buffering, the computation part contains three processes running in parallel.

3.3.7 On-chip Memory Resource Analysis and Modeling

One buffer module contains a mask array, address array, and index array. After applying array reshaping, the mask array with 18432 bits is reshaped into 512×36 bits, which can fit into one memory bank of 18K BRAM. The address array with a shape of 18432×9 bits is reshaped to 2304×72 bits. An ultra Random-access memory (URAM) block size is 4096×76 bits, and the reshaped address array uses one memory bank of URAM. The size of the index array is 512×496 bits, which can either take 14 banks of 18K BRAM or 7 URAM blocks. We use URAMs as much as possible. Therefore, the entire buffer module can consume either ($15 \times$ 18K BRAM $+ 1$ URAM) or ($1 \times$ 18K BRAM $+ 8$ URAM).

There are three buffer modules used on-chip memory, we named bufferB, bufferB_U, and bufferC_U. The resources used are ($15 \times$ 18K BRAM $+ 1$ URAM), ($1 \times$ 18K BRAM $+$ 8URAM), and ($1 \times$ 18K BRAM $+ 8$ URAM) respectively.

IN comparator-based hardware design we use 6 PEGs. The on-chip buffer usage for all the buffer modules can be estimated as follows:

$$\begin{aligned}
\text{On-chip resource usage for 6 PEGs} &= 2 \times (31 \times \text{bufferB_U} + \text{bufferC_U}) \\
&\quad + 4 \times (31 \times \text{bufferB} + \text{bufferC_U}) \\
&= 124 \times \text{bufferB} + 62 \times \text{bufferB_U} + 6 \times \text{bufferC_U} \\
&= 124 \times (15 \times \text{ 18K BRAM} + 1 \text{ URAM}) \\
&\quad + 62 \times (1 \times \text{ 18K BRAM} + 8 \text{ URAM}) \\
&\quad + 6 \times (1 \times \text{ 18K BRAM} + 8 \text{ URAM}) \\
&= 1928 \times \text{ 18K BRAM} + 668 \text{ URAM} \qquad (3.8)
\end{aligned}$$

3.4 Lookup Table Based Hardware Design

In the previous section, we discuss a comparator-based hardware design that leverages the set intersection to count triangles. The major computation in the design is to count the number of common column indices between the B-column index array and the C-column index array. In the previous design, we iterate through the 16-bit column indices in one array and check if they exist in another array. This can be seen as a 16-bit comparator. However, this way of comparison does not utilize the characteristics of a binary sparse matrix, where each row of the matrix can be seen as a binary string [33].

Instead of comparing the indices of NE elements one by one, we could directly use the AND operation between one binary string and another. We can count the number of set 1 in the output binary string using a lookup table. This way of comparison can process multiple NE elements at one time, which is more efficient for a more dense sparse matrix.

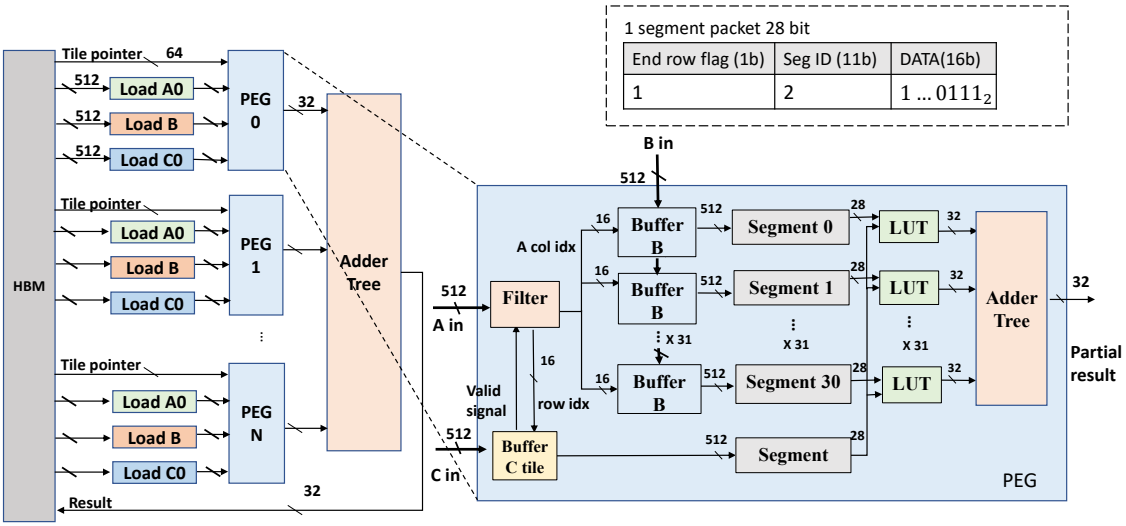

Figure 3.10: The proposed LUT-based hardware design architecture.

3.4.1 Accelerator Architecture Overview

Figure 3.10 depicts the overview data flow architecture for the proposed lookup table-based hardware design. The left part of the figure is the same as the previous comparator-based design architecture in Figure 3.7.

Inside one PEG, the processing order includes the following. Firstly, we buffer one B tile with 31 copies and one C tile on-chip. Both the B tile and C tile use the same buffering strategy. After buffering both the B tile and C tile, the next step is to access the *A in* data in a streaming fashion to the filter module. The filter module will output A column indices used to access B tile. Meanwhile, the filter module will output the A row index used to access the C tile. After that, all the B buffer modules will output column indices to the segment module doing the data slice. The main limitation of the comparator-based design is that each time only one column index can be compared with another column index in the comparator module. This leads to low parallelization. By using the segment to store multiple NZ elements inside one segment. Helping us to process multiple NZ elements at one time.

Each segment module will receive column indices stream B and it is a fixed-size data segment pack. The segment pack is used to decompress NE column indices to binary data segments with 1 and 0 inside. When input column indices stream in sequentially, the same segment pack will be updated. We then stream out the segment packet to the LUT module once the next column indices are out of range of the current data segment.

The LUT module has two inputs, including a B-segment pack stream and a C-segment pack stream. Similar to the set Intersect way of comparator in the previous design, this time it will compare the data segments of two input streams. The LUT module aims to use

a pre-defined LUT to count the bit set 1 in a given binary segment. In each cycle, the LUT module outputs the 32-bit results. In the end, an adder tree is used to add all the results generated by 31 LUT modules and output a 32-bit value as the partial result for TC.

Detailed Processing

The overall processing sequence can be elucidated using the example in Figure 3.11. We use the same input example as in the previous design section. Consider a scenario where only one PEG is used in the architecture. Upon tiling, the FPGA kernel processes data tile by tile in a sequential manner. In this particular instance, we have three input matrices A, B, and C. Suppose the B tile and C tile are buffered on-chip already, then we input the A stream from the HBM.

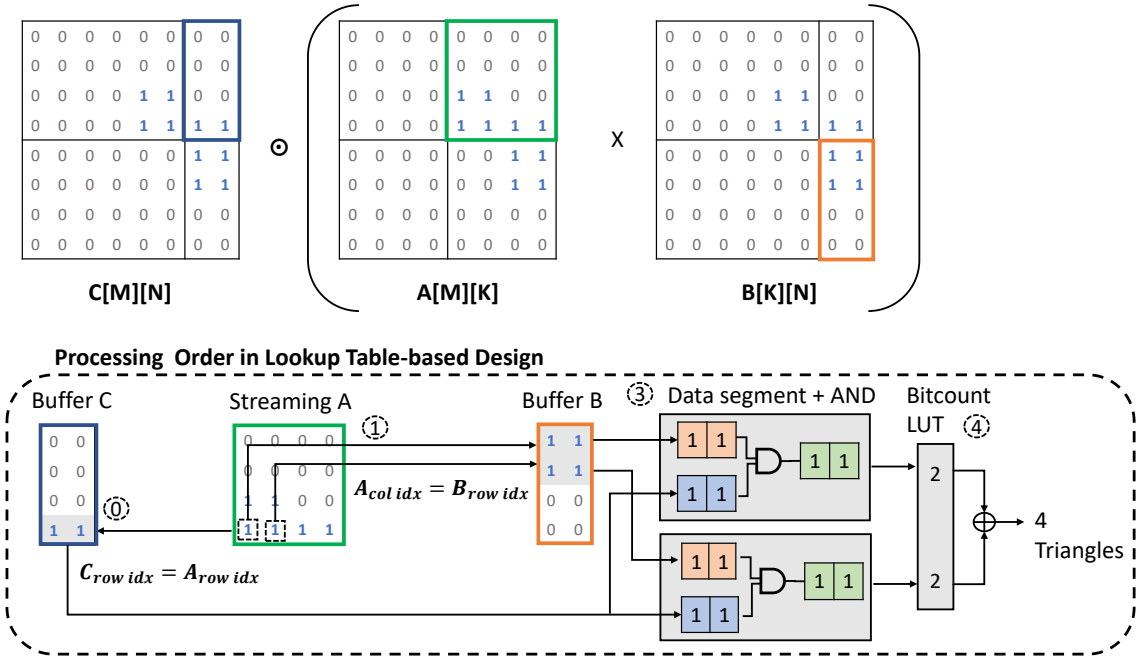

Figure 3.11: Processing order for LUT-based hardware design.

Simpliar as Figure 3.8, it first buffers B and C tiles. Then filtering out some of the data in stream A. Then the column indices to find the corresponding row of B row indices. Then, we identify the corresponding column indices in the B buffer, the subsequent step involves decompressing the column indices of C tile into a dense format and applying the data-slicing technique to each row of C tile. Then we update the data segment and stream out to LUT module. The same data segment processing needs to apply for B tile data. In the given example, we set the segment size equal to 2, meaning each segment value is 2 bits. If we want to compare 0^{th} row of the B with 3^{rd} row of C, we slice the 0^{th} row of the B with segment width equal to 2, and do the same thing to the 3^{rd} row of C.

The next step is for segment comparison. After transferring 1 B segment and 1 C segment into 1 LUT module, inside the LUT module, we first compare the segment indices between

two input segments. We can do an AND operation to find the intersection in one cycle. In reality, the slice indices between B segment and C segments might be different. Only when both segments have the same segment indices, we can do the following intersection operation and computation. We then use a LUT to count the set 1 inside the intersect segment in one cycle. The LUT in this case is a fixed array that includes all the possibilities for the count bit set. In this example, the LUT contains 4 elements (0, 1, 1, 2), corresponding to 4 different input conditions ("00", "01", "10", "11"). In addition, there are two LUT segments in this example to count the bit set in parallel.

The last step is to sum the results from 2 LUT modules and get the final result equal to 4. Note this example only indices the computation process order, but does not show all hardware components.

3.4.2 Data Segment Design

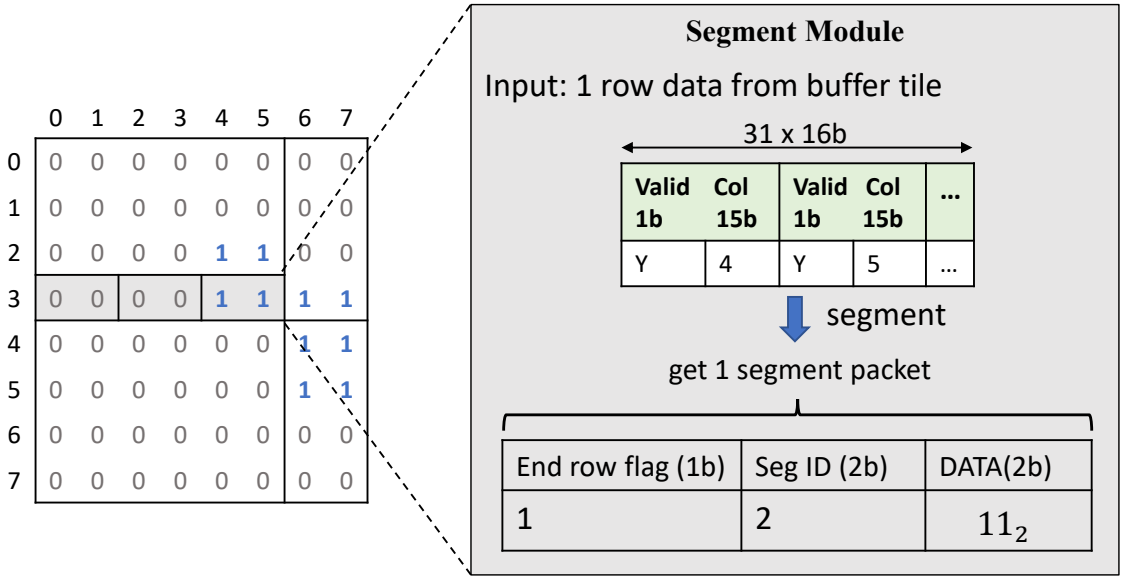

Figure 3.12: Hardware component: data segment module design.

The data segment module is one of the optimized components in LUT-based hardware design. The main feature is to utilize the bitwise logic operation (AND) to replace the mathematical multiplication for the binary sparse matrices. Equation 2.5 shows the formula for TC using bitwise logic computation. To multiply one B row with one C row with AND operation, data have to be stored in a dense format, which is inefficient for a sparse matrix since most of the data are 0s. Based on this condition we propose a data segmentation strategy that slices a large dense row into multiple segments. Only the non-empty segments

will be processed in the later computation.

$$\text{segment ID} = \frac{\text{column index}}{\text{segment size}} \tag{3.9}$$

$$\text{bit offset} = \text{column index mod segment size} \tag{3.10}$$

Figure 3.12 demonstrates the storage format of a data segment. Suppose we want to do data slicing on the 3rd row of Tile #0. At the beginning, the NE element column indices are stored in the index array which contains column indices (4, 5). The next step is to iterate through that row of indices and calculate the segment ID and bit offset, as shown in Equation 3.10. In this example the segment size = 2. Both column indices 4 and 5 have segment ID = 2, and bit offsets are 0 and 1 respectively. Since there is no more column indices in the 3rd row, the next step is to encode the metadata of this segment into a packet. In this example, the 1 packet (28 bits) consists of data (2 bits), segment ID (2 bits), and end row flag (1 bit). The bit width of segment data equals the segment size. The bit width of segment ID equals $log_2(\frac{max\ tile\ width}{segmentsize})$. All these bit width parameters can be adjusted to fit the actual data limitation. The end row flag is a 1-bit signal to denote whether the current row ends or not. It helps for following segment comparison in the LUT-based intersection module since all the segments are input in a streaming fashion.

3.4.3 LUT-based Intersect Module

After the data segmentation process, the next step is to count the intersection between segments. Algorithm 5 shows the computation process. Inputs of the LUT-based intersection module contain two sets of segments $Segs_B$ and $Segs_C$. There is a for loop to keep reading two streams and using two pointers i and j to tack the current position of each two streams. Since the stream lengths of $Segs_B$ and $Segs_C$ might differ, we first need to check whether the two streams are valid or not. We use $|Segs_B|$ and $|Segs_B|$ to indicate the length of two input streams. If both two streams are not reached at the end (Line 4), we do the following computation: We first decode two segment packets to get the metadata, including the end row flag, segment ID, and segment data. We then check whether segment B and segment C are at the end of the row. Only when both segments are not at the end of the row, we start comparing the segment ID. If the two segment IDs match, we multiply segment B with segment C to get the intersection result. The multiplication in binary is the same as the AND operation. We then use a self-defined lookup table called BitCount to count the number of 1 inside the intersection segment. For example, if the input is t-bit, the Bitcount array contains 2^t elements to represent all the possibilities of different input numbers.

Algorithm 5: LUT-based intersection ($Segs_B$, $Segs_C$).

Input: Two sets of segments $Segs_B$ and $Segs_C$
Output: Number of intersection count: *count*

1: $i \leftarrow 0$; $j \leftarrow 0$ // i and j track the index of sets $Segs_B$ and $Segs_C$;
2: $count \leftarrow 0$;
3: **while do**
4: | **if** $i < |Segs_B|$ *and* $j < |Segs_C|$ **then**
5: | | $(FLAG_{\text{end row B}}, bSeg_{ID}, bSeg_{data}) = \text{decoder}(Segs_B[i])$;
6: | | $(FLAG_{\text{end row C}}, cSeg_{ID}, cSeg_{data}) = \text{decoder}(Segs_C[j])$;
7: | | **if** $FLAG_{end\ row\ B} = 0$ *and* $FLAG_{end\ row\ C} = 0$ **then**
8: | | | **if** $bSeg_{ID} = cSeg_{ID}$ **then**
9: | | | | $count \leftarrow count+$ BitCount($bSeg_{data}$ AND $cSeg_{data}$);
10: | | | | $i \leftarrow i + 1$; $j \leftarrow j + 1$;
11: | | | **end**
12: | | | **else if** $bSeg_{ID} < cSeg_{ID}$ **then**
13: | | | | $i \leftarrow i + 1$;
14: | | | **end**
15: | | | **else**
16: | | | | $j \leftarrow j + 1$;
17: | | | **end**
18: | | **end**
19: | | **else if** $FLAG_{end\ row\ B} = 1$ *and* $FLAG_{end\ row\ C} = 0$ **then**
20: | | | $j \leftarrow j + 1$;
21: | | **end**
22: | | **else if** $FLAG_{end\ row\ B} = 0$ *and* $FLAG_{end\ row\ C} = 1$ **then**
23: | | | $i \leftarrow i + 1$;
24: | | **end**
25: | | **else**
26: | | | $i \leftarrow i + 1$; $j \leftarrow j + 1$;
27: | | **end**
28: | **end**
29: | **else if** $i = |Segs_B|$ *and* $j < |Segs_C|$ **then**
30: | | $j \leftarrow j + 1$;
31: | **end**
32: | **else if** $i < |Segs_B|$ *and* $j = |Segs_C|$ **then**
33: | | $i \leftarrow i + 1$;
34: | **end**
35: | **else**
36: | | break;
37: | **end**
38: **end**

3.4.4 Algorithm

Algorithm 6 shows the pseudo-code for TC accelerations with the proposed LUT-based hardware architecture. The part in Lines 1-11 is the same as the previous comparator-based

design. For $C_{mn}[i]$, we need to decode the entire row with a data slicing technique to tile $C_{mn}[i]$ into a set of segments $Segs_C$. Meanwhile, we use the NE column indices of A tile as the index to access corresponding rows in B tile, as shown in lines 13-15. For each row of $B_{kn}[j]$, we apply the same data slicing strategy to slice the current row into a set of segments $Segs_B$. Different iterations in the for loop (line 13) are run in parallel. Different $B_{kn}[j]$ needs to compare the data from the same $C_{mn}[i]$. The sub-function, called LUT-based Intersection gets the two sets of segments for B and C in a streaming fashion. The detailed logic for segment comparison and intersection can be found in Algorithm 5. This function will output a set of results. The last step (line 16) is to accumulate those results. The detailed computation logic for LUT-based intersection algorithm will be explained in the next subsection.

Algorithm 6: Block-based TC with LUT-based method.

Input: (1) three copies of matrix U, named matrix A, B, and C (2) matrix hyper-parameter m_{cuts}, n_{cuts}, k_{cuts} (3) Q pointer list
Output: number of triangles: tc

```
1:  tc ← 0;
2:  for k = 0 to k_cuts − 1 do
3:      for n = 0 to n_cuts − 1 do
4:          Buffer tiles B_kn;
5:          for m = 0 to m_cuts − 1 do
6:              Buffer tile C_mn;
7:              for p = 0 to P do
8:                  // run in parallel;
9:                  for r = Q_mk to Q_mk+1 do
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 30 do
14:                             //run in parallel;
15:                             Slice B_kn[j] into a set of segments Segs_B;
16:                             tc ← tc + LUT-based Intersect(Segs_B, Segs_C);
17:                         end
18:                     end
19:                 end
20:             end
21:         end
22:     end
23: end
```

3.4.5 Performance and On-chip Memory Analysis and Modeling

Since both the LUT-based hardware design and the previous comparator-based design used the same buffer module, the on-chip buffer usage is the same as the previous design. The detailed estimation can be found in the previous section.

For performance analysis, We use Algorithm 6 as an example. For better distinction, three copies of input matrices are named as matrices A, B, and C. The dimensions of the three matrices A, B, and C are $M \times K$, $K \times N$, and $M \times N$ respectively. Due to our specific tiling scheme as shown in Figure 3.3, we use m_{cuts}, n_{cuts}, and k_{cuts} to denote the number of cuts in the M, N, and K dimensions respectively. The example in Figure 3.3 contains $m_{cuts} = n_{cuts} = k_{cuts} = 2$. Meanwhile, we use m, n, and k to represent the tile indices. For example, $B_{1,2}$ means a sub-matrix of B at position $(1, 2)$.

1. In line 2-5, we process the input tile by tile, and we only process the data when both tile A, B and C are non-empty. In line 4 we buffer B tile on-chip, which is the same as the previous comparator-based design. The cycle count for buffer tile B is:

$$\begin{aligned} T_{\text{bufferB}} &= T_{\text{reset row mask}} + T_{\text{buffer index array}} \\ &= [(\text{row mask Height} - 1) \times \text{II} + \text{iteration latency}] \\ &+ [(\text{avg \#rows in B tile} - 1) \times \text{II} + \text{iteration latency}] \end{aligned} \tag{3.11}$$

2. The cycle count for buffer tile C is:

$$\begin{aligned} T_{\text{bufferC}} &= T_{\text{reset row mask}} + T_{\text{buffer index array}} \\ &= [(\text{row mask Height} - 1) \times \text{II} + \text{iteration latency}] \\ &+ \left[(\frac{\text{avg \#rows in B tile}}{\text{PEG}} - 1) \times \text{II} + \text{iteration latency}\right] \end{aligned} \tag{3.12}$$

   Unlike buffering the entire data in one tile, each C buffer only contains $\frac{1}{PEGs}$ part of C tiles.

3. The PEG region (lines 10-18) contains three parts: data segment of B, data segment of C, and LUT-based intersection. Before computation, rows in A tile have been filtered by C tile, and we used $\#rows_{A \bigcap C}\ perPEG$ to indicate the number of common rows in A tile generated after filtering.

   To segment one row of index array in a buffer tile, we need to iterate the column indices in a pipeline fashion updating the same segment packet and streaming out to the next computation module. The cycle count for data segmentation of C is:

$$
\begin{aligned}
T_{\text{seg C}} &= \#rows_{A\bigcap C}\ perPEG \times T_{\text{segment1row}} \\
&= \#rows_{A\bigcap C}\ perPEG \times [(bufferwidth - 1) \times \text{II} + \text{iteration latency}]
\end{aligned}
\tag{3.13}
$$

The cycle count for the data segmentation of B is given below. We use $\bigcap_{i\in R_{A\cap C}} A_{i,*} \cap R_B$ to represent the intersection of the column indices inside the rows of $\#rows_{A\bigcap B}\ perPEG$ with the rows in B tile in matrix notation.

$$
\begin{aligned}
T_{\text{seg B}} &= \bigcap_{i\in R_{A\cap C}} A_{i,*} \cap R_B \times T_{\text{segment1row}} \\
&= \bigcap_{i\in R_{A\cap C}} A_{i,*} \cap R_B \times [(bufferwidth - 1) \times \text{II} + \text{iteration latency}]
\end{aligned}
\tag{3.14}
$$

The cycle count for LUT-based Intersect is:

$$
\begin{aligned}
T_{\text{LUTIntersect}} &= \#rows_{A\bigcap C}\ perPEG \times T_{compare1row}\text{range}(min_{cc}, max_{cc}) \\
min_{cc} &= \max(\text{avg \#seg per B row}, \text{avg \#seg per C row}) \\
max_{cc} &= \text{avg \#seg per B row} + \text{avg \#seg per C row}
\end{aligned}
\tag{3.15}
$$

The LUT-based Intersection is similar to the set Intersection method that was used in previous hardware design. Both of them contain one loop with a pipeline across different iterations. However, the cycle count for LUT-based intersection depends on the segment size and NE element distribution. Due to the random NE distribution of the sparse matrix, for better estimation, we consider the average value of segments per row in each tile.

4. Total cycle count:

$$
\begin{aligned}
T_{\text{cc}} &= \text{Number of valid}(tileA, tileB, tileC) \\
&\quad \times [\max(T_{\text{bufferB}}, T_{\text{bufferC}}) + \max(T_{\text{seg B}}, T_{\text{seg C}}, T_{\text{LUTIntersect}})]
\end{aligned}
\tag{3.16}
$$

Chapter 4

Experimental Results

In this chapter, we describe our experimental setup and present our experimental results. For comparator-based design, we will show the performance changing when scaled-up hardware design. For the look-up table design, we will present the performance comparisons for different segment sizes and find a suitable segment size with better performance. After that, we will show the performance comparison between the comparator-based design and the LUT-based design. Using this segment size as the fixed parameter, we will illustrate the performance improvement when scaling up the design. Then we will compare our design with other FPGA work of the TC accelerator.

4.1 Evaluation Setup

We use the Xilinx high-level synthesis (HLS) tool with Vitis version 2021.2 to implement both comparator-based design and LUT-based design on AMD-Xilinx HBM-based Alveo U280 FPGA. We compare the performance of our design with the open-source Vitis TC FPGA library on the same FPGA board, as well as optimized multi-core CPU implementation on two 12-core Intel Xeon Silver 4214 CPUs.

We assess the performance of our proposed hardware designs through an evaluation conducted on 12 real-world graphs sourced from the SuiteSparse Matrix Collection [6]. Table 4.1 presents comprehensive details of these selected graphs. Our dataset comprises a diverse array of graphs ranging from small-scale to large-scale, exhibiting various degrees of sparsity, thereby ensuring a comprehensive evaluation of our hardware implementations. The graph datasets range in the number of vertices from 4K to 131K, and in the number of edges from 88K to 14M. The density of the graphs varies from 4.39E-04 to 8.78E-02.

Since the triangle counting project is an end-to-end project with CPU + FPGA codesign platforms, we have a few assumptions. We assume the input graph is stored in CSR format in RAM already. The comprehensive evaluation of TC's execution time does not incorporate the duration required for data transfer from the CPU's solid-state drive (SSD) to the random

Table 4.1: Selected graph dataset.

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

access memory (RAM), nor does it encompass the time taken for data transfer between the CPU and the FPGA.

4.2 Performance of Comparator-based Hardware Design

In this section, we analyze the performance of comparator-based hardware design, by dividing it into several parts. Firstly we will list all the configuration settings of the design that achieved the best performance. Then we will show a detailed runtime breakdown for each module inside the architecture. The final part of the analysis will discuss the performance improvements when scaling up the hardware design by increasing the number of PEGs.

4.2.1 Design Configuration

Table 4.2 provides a detailed configuration for the comparator-based hardware design. Due to the limited on-chip resource, we can put a maximum used 6 PEGs inside the architecture, with each PEG containing 31 PEs. The number of PEs per PEG in our design is decided by the HBM port width and our row-wise matrix multiplication way. In total, this design incorporates 25 High Bandwidth Memory (HBM) channels.

Moreover, it outlines the stream width when transferring data from off-chip data as 512 bits. The maximum dimensions for each tile are specified, with a width of 32,768 and a height of 18,432. The sizes of various arrays crucial to the design are: the index array (512 x 496 bits), the mask array (512 x 36 bits), and the address array (2304 x 72 bits). These parameters provide insight into the hardware configuration and specifications essential for the comparator-based design.

Table 4.2: Comparator-based hardware design parameters

Parameters	**Value**
# PEGs	6
# PEs per PEG	31
# HBM Channel	25
stream width	512 bits (32 x 16)
max tile width	32,768
max tile height	18,432
indices array size	512 x 496 bits
mask array size	512 x 36 bits
address array size	2304 x 72 bits

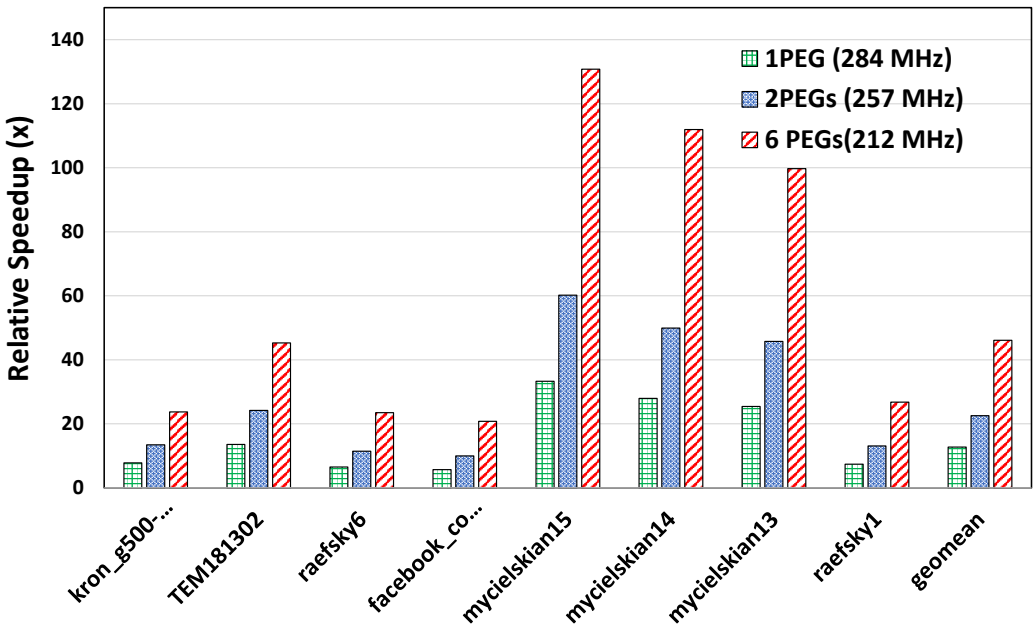

Figure 4.1: Comparison of the relative speedup for various scales of the comparator-based hardware design against the baseline.

4.2.2 Performance Scaling Analysis

The bar charts in Figure 4.1 illustrate the comparison of the relative speedup for various scales of the comparator-based hardware design against the baseline (1 core CPU execution time), including 1 PEG, 2 PEGs, and 6 PEGs. Our method exhibits notable speedup compared to the baseline on three datasets, namely *mycielskian13*, *mycielskian14*, and *mycielskian15*. This is attributed to their characteristics as strongly connected graphs with relatively higher densities, ranging from 1.84E-02 to 3.25E-02, as indicated in Table 4.1.

Note that the density of the input graph significantly influences the performance, with denser graphs generally resulting in better performance. For instance, although *raefsky6* and *mycielskian13* possess similar numbers of vertices, *raefsky6* has fewer edges compared to *mycielskian13*.

The geometric mean of the nine datasets is presented in the final three columns, revealing relative speedups of 14x, 26x, and 53x for 1 PEG, 2 PEGs, and 6 PEGs, respectively. The speedup increases by approximately 2 times when scaling up from 1 PEG to 2 PEGs. However, as the design scales from 2 PEGs to 6 PEGs, the performance does not linearly increase. This is attributed to the reduction in achievable frequency as the hardware design is scaled up by adding more PEGs. Specifically, the achievable frequency decreases from 284 MHz to 257 MHz and 212 MHz when scaling up from 1 PEG to 2 PEGs and 6 PEGs, respectively. After investigation, we found that adding more PEGs in FPGA designs can lead to increased routing congestion due to design complexity, resulting in longer signal propagation delays and decreased clock frequency. Additionally, more PEGs raise resource utilization, which can restrict resources for critical paths and reduce achievable frequency. Moreover, scaling up PEGs can exacerbate timing closure challenges, making it harder to meet timing requirements (300 MHz in this case). As the clock frequency increases, the duration of each clock cycle decreases, allowing more operations to be performed per unit of time, thereby reducing the execution time of a kernel or application. As a result, further scaling up PEGs cannot get linear improvement for the performance in reality.

4.2.3 Execution Time Breakdown

To further refine our design and facilitate design exploration, we have applied the analytical model detailed in Section 3.3.5 to a range of datasets. Figure 4.2 illustrates the estimated execution time breakdown for our comparator-based design, as calculated using Equation 4.1. These estimations were made using a realized frequency of 221 MHz, and the cycle count was informed by the performance analysis previously discussed in Section 3.3.5. The datasets are arranged in ascending order of density. Given that the TC problem does not necessitate storing a large volume of results, our focus was primarily on the load and computation aspects. In the loading phase, tiles B and C are buffered concurrently. During execution, the buffering times for B and C may vary. In the computation phase,

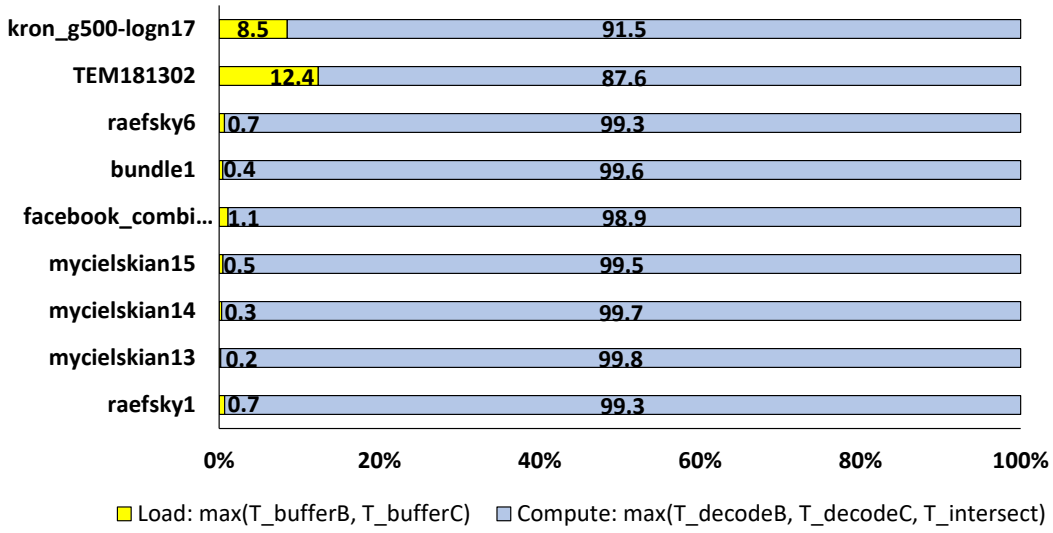

Figure 4.2: Comparator-based design execution time breakdown

three tasks—decoding B, decoding C, and the intersection—are executed in parallel. As tile densities differ, the bottleneck in the computation phase can shift among these three modules.

$$\text{Execution Time} = \frac{\text{Cycle Count}}{\text{Frequency}} \tag{4.1}$$

It is noteworthy that the datasets for TEM181302 and kron-g500 show a comparatively larger proportion of load time than others. These two datasets have a lower density, resulting in the computation occupying a smaller percentage of the total execution time. Our tiling strategy for the sparse matrix, which is contingent on the number of non-zero elements, means that the load times for different tiles are approximately equal. Consequently, if the load time remains constant and the computation time decreases, the relative share of computation time also diminishes. Overall, we alleviate the high memory-to-compute ratio of the matrix-multiplication way of TC. Because we use streaming design to overlap the loading time with the computation time.

4.3 Performance of Lookup Table-based Hardware Design

In this section, we analyze the performance of the LUT-based Hardware Design, breaking it down into several parts. We start by listing the configuration settings that yielded the best performance. Then, we provide a detailed runtime breakdown for each module within the architecture. Additionally, we demonstrate how to select an appropriate segment size based

on our evaluation. Finally, we discuss the performance enhancements achieved by scaling up the hardware design through an increase in the number of PEGs.

4.3.1 Design Configuration

Table 4.3 outlines the specific parameters for the LUT-based hardware design. Like the previous design, the LUT-based design is constrained by on-chip resource utilization and supports a maximum of 6 PEGs, with each PEG consisting of 31 PEs. While many configurations remain consistent with the comparator-based design, the LUT-based design introduces two additional parameters: segment data size (16) and segment ID bit-width (11 bits).

Table 4.3: LUT-based hardware design parameters

Parameters	**Value**
# PEGs	6
# PEs per PEG	31
# HBM Channel	25
stream width	512 bits (32 x 16)
max tile width	32,768
max tile height	18,432
indices array size	512 x 496 bits
mask array size	512 x 36 bits
address array size	2304 x 72 bits
segment data size	16
seg ID bit width	11

4.3.2 Segment Size Analysis

In the preceding section, we emphasize the significance of segment size selection. Larger segment sizes facilitate greater parallelization but come with increased resource usage and higher memory overhead. Therefore, balancing resource utilization and performance is crucial in LUT-based designs. For instance, if the segment size is set to 6, each NE element stored in the segment requires 1 bit of a 6-bit input-output LUT. In contrast, with a segment size of 32, accessing a single bit of 32-bit data necessitates a 32-bit input-output LUT. Given that the total number of segment modules equals $32 \times \#PEGs = 192$, choosing an appropriate segment size becomes imperative.

The bar chart in Figure 4.3 shows the comparison of the relative speedup for various data segment sizes of the LUT-based hardware design with 6 PEGs against the baseline (1 core CPU execution time). The segment sizes 8 (212 MHz), 16 (211 MHz), and 32 (204 MHz) are the tested date segment size, each associated with a final achievable frequency. The chart demonstrates that as resource utilization increases, the achievable frequency tends to decrease. Notably, the datasets mycielskian13, mycielskian14, and mycielskian15 exhibit significant speedup compared to the baseline due to their characteristics of strongly

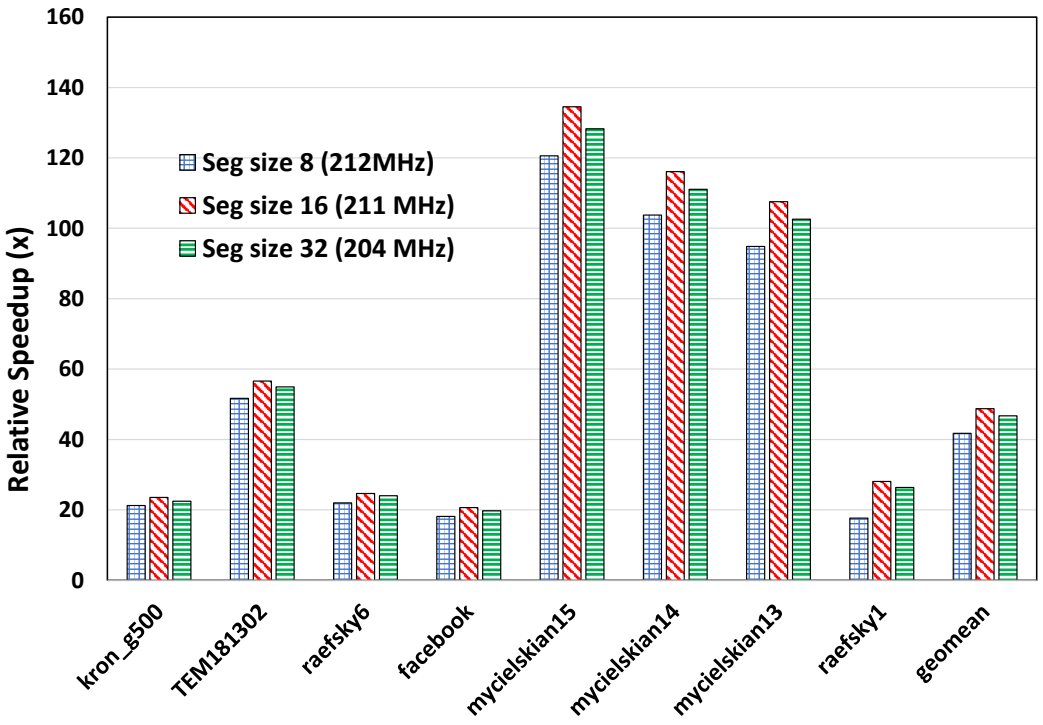

Figure 4.3: Performance comparison for LUT-based hardware design with variant segment size.

connected graphs with higher densities. The LUT-based design uses segment comparison, which is advantageous for denser matrices. With a fixed segment size, segments with more NE elements are processed concurrently, leading to higher parallelization. Conversely, if each segment only contains one NE element, processing time remains the same, but the optimization is lower, as each cycle processes only one NE element. In that case, the performance is the same as the comparator-based design. For a segment size of 8, the geometric mean speedup is approximately 40x, while for a size of 16, it is around 55x. The speedup drops for a segment size of 32, with a geometric mean of approximately 42x. Overall, the chart suggests that a segment size of 16 offers the best performance compared to sizes 8 and 32.

4.3.3 Performance Scaling Analysis

The bar chart in Figure 4.1 illustrates the comparison of the relative speedup for various scales of the LUT-based hardware design against the baseline (1 core CPU execution time). The hardware designs are scaled up to include 1 PEG, 2 PEGs, and 6 PEGs. Three datasets, namely *mycielskian13*, *mycielskian14*, and *mycielskian15*, exhibit notable speedup compared to the baseline, as we discussed previously, because the LUT-based TC is suitable for more dense sparse matrices. The last three columns show the geometric mean value for 1 PE, 2

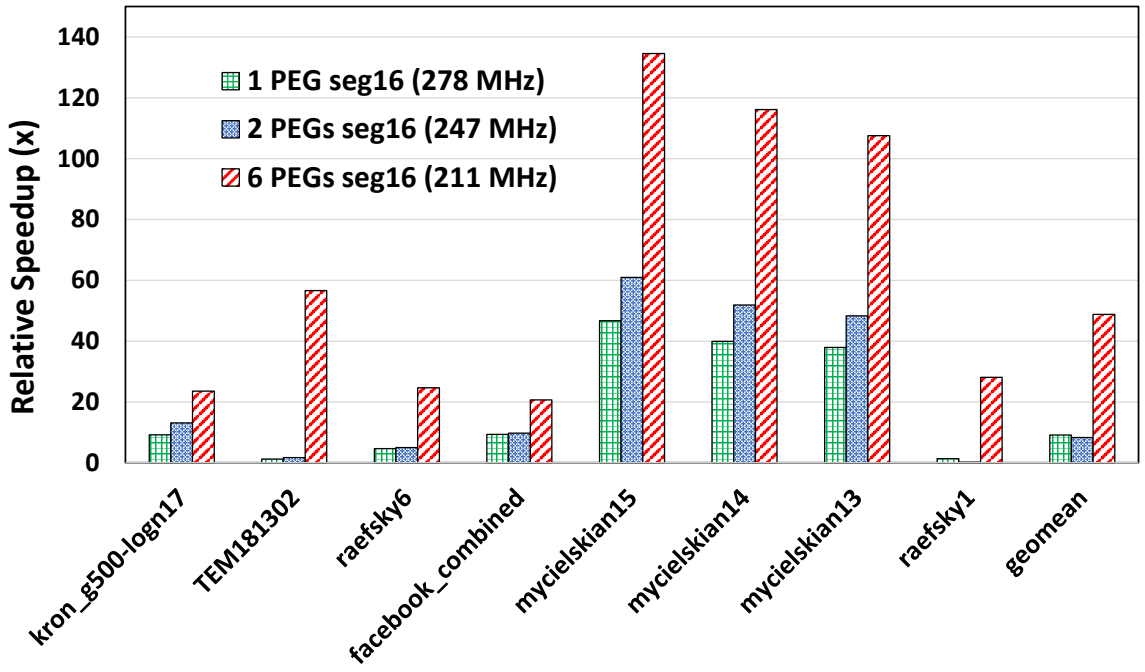

Figure 4.4: Performance comparison when scaled-up LUT-based hardware design.

PEGs, and 6 PEGs, which are approximately 10x, 10x, and 56x. Here the relative speedup for 1 PEG and 2 PEGs are roughly the same, because the achievable frequency of 1 PEG is significantly higher than the 2 PEGs design, even though the scale of 2 PEGs design is two times larger than the 1 PEG. As a result, it shows the importance of having a higher clock frequency.

In terms of timing optimization for HLS designs, we use the TAPA [13] framework along with the Autobridge framework. TAPA/Autobridge introduces a task-parallel HLS programming model along with a coarse-grained floorplanning strategy aimed at enhancing timing closure and clock frequency.

In summary, the chart illustrates that as the FPGA design is scaled up by increasing the number of PEGs, there is a noticeable increase in the relative speedup. This suggests that while scaling up can increase parallelism and potentially improve performance, it may also lead to challenges such as reduced achievable frequency and increased complexity, which can impact the overall speedup.

4.3.4 Execution Time Breakdown

Similar to the performance analysis in comparator-based designs, we propose an execution time breakdown for LUT-based designs, as illustrated in Chart 4.5. We employ a frequency of 211 MHz to convert the estimated cycle count into execution time. The ratio between the loading and computation segments for various datasets is similar to those observed in the comparator-based design. This similarity arises because we have merely replaced the

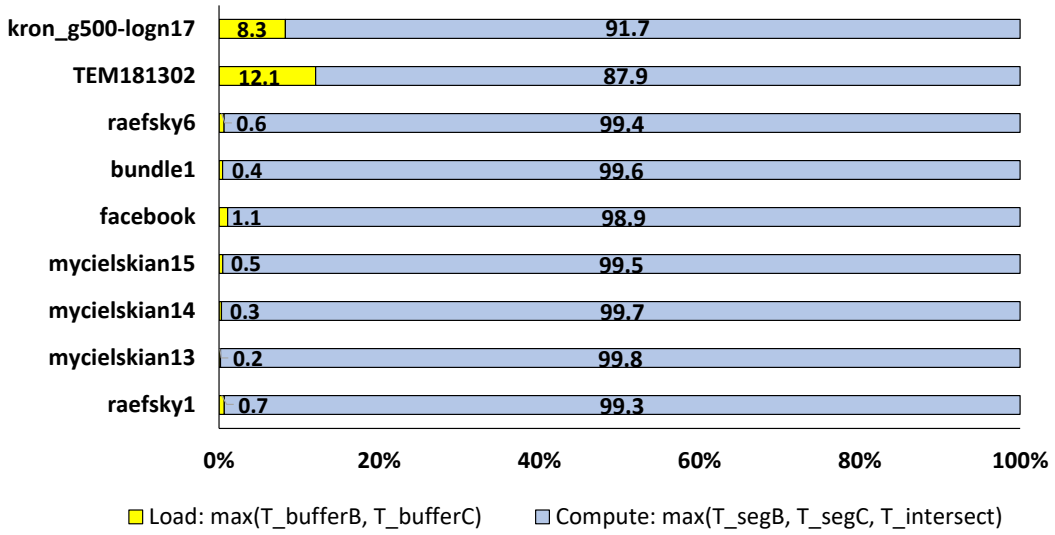

Figure 4.5: LUT-based design execution time breakdown

comparator module with a segment module in conjunction with the LUT module. Access to the LUT does not constitute a bottleneck in the computation process. Instead, the computational bottleneck primarily resides within the segment B and segment C modules.

$$\text{Accuracy} = 1 - \frac{|T_{\text{estimate}} - T_{\text{actual}}|}{T_{\text{actual}}} \tag{4.2}$$

Figure 4.6 illustrates the accuracy of the analytical module, which can be computed using Equation 4.2. Overall, the analytical module's accuracy across various datasets hovers around 90%. A primary factor influencing accuracy is the pipeline feature of our streaming design, wherein each module operates in a pipeline fashion. Consequently, there is an overlapping execution time among different hardware modules, which complicates accurate estimation. Since both hardware designs utilize a similar method for performance assessment, the precision of the LUT-based design's estimation is correspondingly relative to that of the comparator-based design.

4.4 Overall Performance Comparison between Design 1 vs Design 2

After evaluating the performance of both the comparator-based and LUT-based designs, we compare their respective performances. Each design is configured to achieve its best performance, with both employing 6 PEGs. For the LUT-based design, a segment size of 16 is chosen as it yields the best performance.

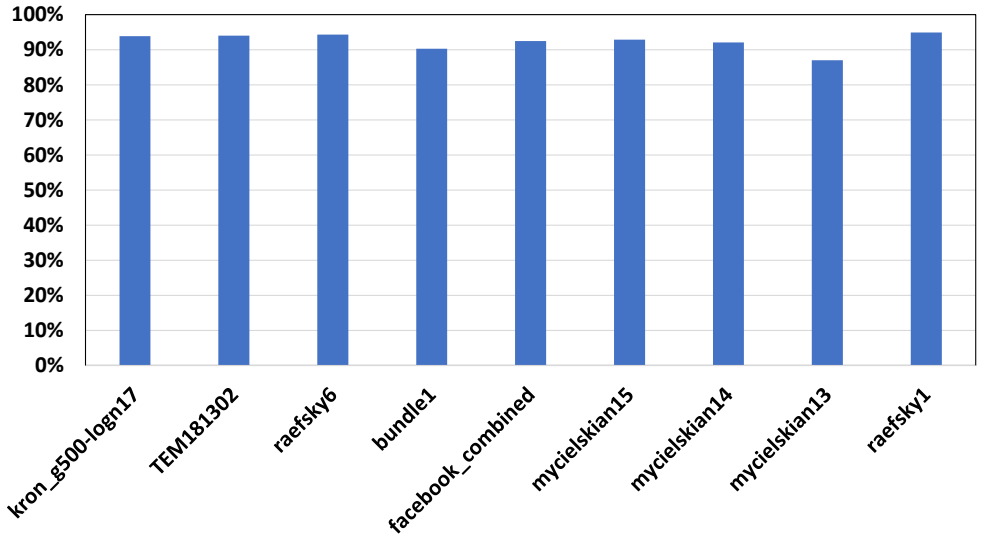

Figure 4.6: Accuracy of analytical module in LUT-based design

The bar chart in Figure 4.7 shows the speedup of the LUT-based design over the comparator-based design. The red line indicates a speedup of 1x, i.e., equal performance between the two designs. Bars above the red line indicate that the LUT-based design outperforms the comparator-based design in terms of execution time.

Out of the 12 datasets tested, only one dataset (*kron_g500-logn17*) shows that the comparator-based design is slightly better than the LUT-based design. The geometric mean value indicates that the LUT-based design is 1.1x faster than the comparator-based design. This highlights that the LUT-based design offers a higher relative speedup compared to the comparator-based design, suggesting that the LUT-based approach is more efficient in terms of FPGA kernel execution time.

4.5 Resource Utilization and Design Frequency

Table 4.4 provides a comparison of resource utilization and operating frequency for two designs of the HiTC accelerator implemented on the AMD Xilinx Alveo U280 platform. Resource utilization is presented in terms of Look-Up Tables (LUTs), Flip-Flops (FFs), Block RAMs (BRAMs), Ultra RAMs (URAMs), and Digital Signal Processors (DSPs). To fully leverage the maximum bandwidth of the HBM at 450MHz with 512-bit ports, our target frequency is 225MHz. We achieve 221 MHz and 211 MHz for comparator-based design and LUT-based design respectively. There is a slight frequency loss due to timing closure and routing congestion problems, which is reasonable.

In the first design, named "Comparator-based TC (U280)," The resource bottleneck for both two designs is URAM usage (69.6%). As we discussed in the previous chapter, both designs use the same buffer modules, thus the BRAM and URAM usage is roughly the

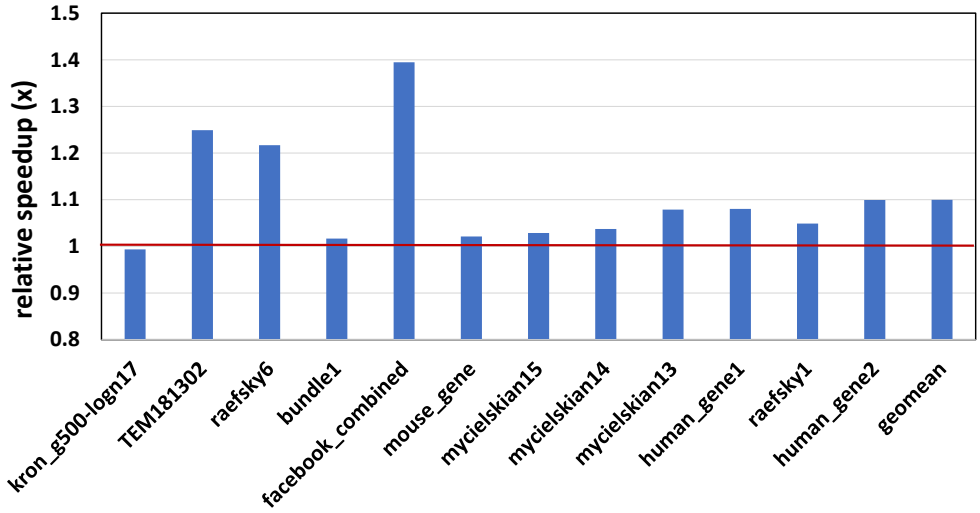

Figure 4.7: The speedup of LUT-based design over comparator-based design

Table 4.4: HiTC resource utilization and frequency

Design Name	**Resource Utilization**					**Freq. (MHz)**
	LUT	**FF**	**BRAM**	**URAM**	**DSP**	
Comparator-based (U280)	48.8%	31.5%	60.8%	69.6%	30.4%	221
LUT-based (U280)	57.6%	34.8%	65.6%	69.6%	34.9%	211

same. Comparing the resource usage between the first design and the second design, the second design used a little bit more resources for LUT, FF, and BRAM, because the second design uses LUT-based intersection modules, which require a wider FIFO channel to transfer data, since each 1 packet of segment takes 28 bits. On the other hand, the comparator-based design directly transfers column indices (each 16 bits) in a streaming fashion through FIFO. The FIFO by default is implemented by the LUT. Meanwhile, each LUT-based intersection module contains a self-define lookup table which is implemented by registers. As a result, the LUT-based hardware design has more resource consumption than the comparator-based design and also achieves lower frequency than the comparator-based design.

4.6 Comparison with CPU and Other FPGA Design

In this section, we conduct benchmarking on CPU and other FPGA designs. We run experiments using the 12 datasets from the SuiteSparse Collection and compare our HiTC design (LUT-based design, segment size = 16) with Vitis TC FPGA library on the AMD-Xilinx HBM-based Alveo U280 FPGA. We also compare it with the software implementation on two 12-core Intel Xeon Silver 4214 CPUs. As far as we know there are two existing FPGA

implements for TC, including Huang's work [18], and the other one is from the AMD Vitis™ unified software development platform, where the TC is one of the applications provided. However, since Hang's work only does the synthesis instead of implementing the design on the actual hardware board, we will not use Huang's work for comparison.

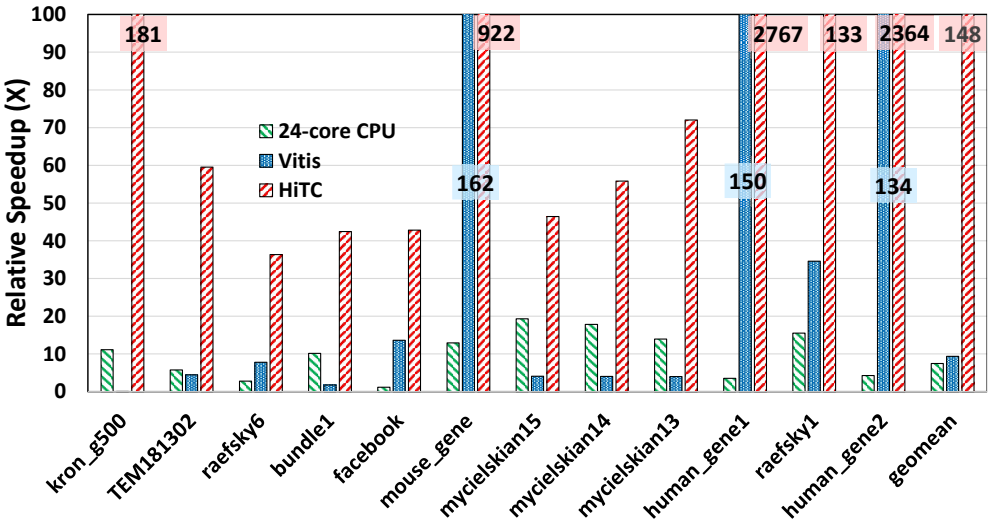

Figure 4.8: HiTC performance comparison of our design against 1-core CPU (baseline), multi-core CPU, and Vitis FPGA design on real-world graphs

The bar chart in Figure 4.8 provides a comparison of the HiTC performance of an FPGA design against a single-core CPU (baseline), a multi-core CPU (24-core), and a Vitis FPGA design, based on real-world graphs. The performance is measured in terms of relative speedup (X-axis), with higher values indicating better performance.

The single-core CPU, serving as the baseline, has a relative speedup of 1, as it is the standard for comparison. The multi-core CPU (24-core) shows a geometric mean speedup of approximately 8x and achieves 2x to 20x speedup across 12 datasets. The limitations in achieving speedup with 48 threads compared to 1 thread on a CPU are primarily due to overhead, data dependency, memory bandwidth constraints, and resource contention. These factors can outweigh the benefits of parallel execution and reduce the overall efficiency of the parallelization. The Vitis FPGA design further enhances the performance, achieving speedups ranging from 2x to 162x, indicating a substantial increase in efficiency.

Our HiTC design stands out with the highest relative speedups compared with the baseline, ranging from 36x to a remarkable 2767x, showcasing its superior performance in handling real-world graphs compared to the other works. There are 3 out of 12 datasets that have over 900x speed up compared with the baseline. It is because those three datasets,

mouse_gene, *human_gene1*, and *human_gene2*, have higher density than other datasets, which are 1.42E-02, 4.96E-02, and 8.78E-02 respectively.

Another important factor is the NE elements are distributed more dispersed than what would be expected in a random distribution. As a result, the nnzs in each row of the tile after tiling are roughly the same. When the HiTC schedules the PE workload, it will assign different rows to different PEGs in a cyclic manner. Based on the dispersion distribution, it is more likely to have a balanced workload for different PEGs. Balancing the workload among PEGs in FPGA acceleration is crucial for optimizing performance. This is because all PEGs operate in parallel, and the overall performance is constrained by the slowest-running PEG. For example the HiTC performance on dataset *Facebook combined* does not have a significant improvement ($\tilde{4}$2x) compared with other datasets, because this dataset consists of friends lists from Facebook, and some celebrities have thousands of friends on Facebook. Thus the adjacency matrix to represent this social network contains some dense rows with more NE elements. Those dense rows will make the PEG workloads imbalanced and further constrain the final performance.

The baseline code used in this thesis is a basic implementation of triangle counting using three compressed sparse row (CSR) matrices A, B, and C. It iterates over each row of matrix A, then for each non-zero entry in A, it iterates over corresponding columns in matrices B and C to check for triangles. For sparse matrices, this approach is efficient because it only processes non-zero entries. However, for dense sparse matrices, where most entries are non-zero, the algorithm becomes slow due to the nested loops iterating over all rows and columns multiple times, leading to a high number of operations. From the software perspective, the nested loops and repeated access to memory locations can cause cache misses and inefficient memory access patterns, especially for dense sparse matrices. This results in longer execution times. From a hardware perspective, repeated random memory accesses and computations can lead to increased resource utilization and longer critical paths. Dense sparse matrices require more operations and memory accesses, which can slow down the FPGA's processing speed.

In summary, the chart demonstrates that the HiTC FPGA design significantly outperforms the single-core CPU, multi-core CPU, and Vitis FPGA design in terms of relative speedup, highlighting its effectiveness in processing real-world graphs.

In addition to actual run time comparison, another evaluation metric is # edges of graph divided by execution time. This metric is commonly used in graph processing algorithms to estimate the throughput of a kernel. The bar chart in Figure 4.9 presents the absolute million edges per second (MEPS) for three implements: a multi-core CPU (24-core), a Vitis FPGA design, and out HiTC work, using real-world graphs.

The multi-core CPU (24-core) has a geometric mean value of 0.4 MEPS. The Vitis FPGA design shows a similar geometric mean of throughput to the multi-core CPU, which is approximately 0.9 MEPS. However, the HiTC FPGA design significantly surpasses both

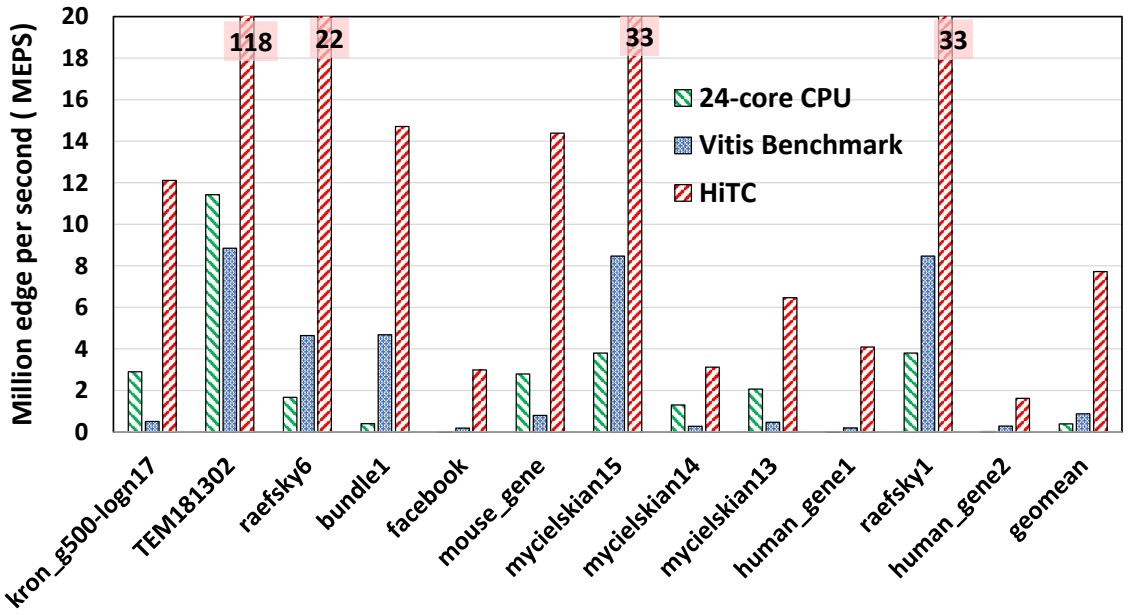

Figure 4.9: Throughput comparison with multi-core CPU and Vitis FPGA design and HiTC on real-world graphs.

the multi-core CPU and the Vitis FPGA design, achieving a geometric mean throughput of 7.7 MEPS, indicating a substantial enhancement in processing speed.

For the actual throughput results in HiTC, *TEM181302* dataset has significantly higher throughput than other datasets, which is 117.9 MEPS. Not only for HiTC, the *TEM181302* dataset achieves the highest result in both multi-core CPU and Vitis benchmarks. Several factors can affect the throughput result, including the graph density, NE elements distribution, data access pattern, and input graph size.

In summary, the chart illustrates that the HiTC FPGA design outperforms the multi-core CPU, and Vitis FPGA design in terms of throughput, showcasing its superior ability to process real-world graphs efficiently.

Chapter 5

Conclusion and Future Work

TC stands as a foundational task in graph computing and social networks, yet its acceleration poses challenges due to its high memory-to-computation ratio and random memory access patterns. In this thesis, we propose HiTC, a software/hardware codesign approach to accelerate triangle counting on HBM-equipped FPGAs. The development of a triangle counting accelerator on an HBM-equipped FPGA confronts three main hurdles:

1. Inefficient HBM Bandwidth Utilization: Traditional sparse matrix compression formats have been shown to be unfriendly and inefficient for HBM.

2. Random Memory Access Patterns: Sparse matrices exhibit an irregular distribution of non-zero elements.

3. Limited On-chip Buffers: FPGA's constrained buffer resources pose challenges for highly sparse matrices with irregular non-zero element distributions.

To tackle these challenges, we devise a triangle counting accelerator based on matrix multiplication, leveraging the bitwise operations inherent in the TC algorithm. We introduce hardware-friendly reordering, tiling, and encoding techniques to address random access issues and optimize bandwidth utilization. Our approach designs streaming-based hardware accelerators on FPGAs, leveraging HBM for higher bandwidth and customizing the computation pipeline for improved computing throughput.

In summary, our work presents HiTC, a novel matrix multiplication-based TC accelerator that overcomes the challenges inherent in accelerating TC on FPGA. We employ a graph reordering strategy, minimum degree order (MDO), to enhance data locality, followed by a sparsity-aware partitioning technique for large-scale datasets. Additionally, we propose efficient buffer techniques for accommodating random distribution nonzero elements within fixed on-chip buffers. Leveraging the characteristics of binary sparse matrix multiplication, we introduce two computation approaches: comparator-based and lookup table-based hardware designs. Our work represents the first implementation of matrix-multiplication-based triangle counting on FPGA, effectively addressing the aforementioned challenges.

Experimenting with the SuiteSparse dataset, HiTC achieves a geometric mean 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 software implementations on two 12-core Intel Xeon Silver 4214 CPUs, HiTC achieves a geometric mean speedup of 18.6x (up to 669.8x).

Future Work: The proposed HiTC accelerator demonstrates superior performance for sparse matrices with densities greater than 10^{-5}. However, its performance is limited when dealing with ultra-sparse matrices. Investigating the set intersect-based method for FPGA could enhance its effectiveness in solving less connected graphs, thereby expanding its applications.

Bibliography

[1] Ariful Azad, Aydin Buluç, and John Gilbert. Parallel triangle counting and enumeration using matrix algebra. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, pages 804–811, 2015.

[2] Vignesh Balaji and Brandon Lucia. When is graph reordering an optimization? studying the effect of lightweight graph reordering across applications and input graphs. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*, pages 203–214, 2018.

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

[4] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, page 151–158, New York, NY, USA, 1971. Association for Computing Machinery.

[5] E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proceedings of the 1969 24th National Conference*, ACM '69, page 157–172, New York, NY, USA, 1969. Association for Computing Machinery.

[6] Timothy A. Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1), dec 2011.

[7] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. Theoretically efficient parallel graph algorithms can be fast and scalable. *CoRR*, abs/1805.05208, 2018.

[8] J. Dongarra, A. Lumsdaine, X. Niu, R. Pozo, and K. Remington. A sparse matrix library in c++ for high performance architectures. In *Second Object Oriented Numerical Conference*, pages 214–218, 1994.

[9] Iain S. Duff. Computer solution of large sparse positive definite systems (alan george and joseph w. liu. *SIAM Review*, 26(2):289–291, 1984.

[10] Facebook. Facebook. `https://www.facebook.com/`. Accessed: [2024-03-21].

[11] Jianhua Gao, Weixing Ji, Fangli Chang, Shiyu Han, Bingxin Wei, Zeming Liu, and Yizhuo Wang. A systematic survey of general sparse matrix-matrix multiplication. *ACM Comput. Surv.*, 55(12), mar 2023.

[12] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, third edition, 1996.

[13] Licheng Guo, Yuze Chi, Jason Lau, Linghao Song, Xingyu Tian, Moazin Khatti, Weikang Qiao, Jie Wang, Ecenur Ustun, Zhenman Fang, Zhiru Zhang, and Jason Cong. Tapa: A scalable task-parallel dataflow programming framework for modern fpgas with co-optimization of hls and physical design. *ACM Trans. Reconfigurable Technol. Syst.*, 16(4), dec 2023.

[14] Fred G. Gustavson. Two fast algorithms for sparse matrices: Multiplication and permuted transposition. *ACM Trans. Math. Softw.*, 4(3):250–269, sep 1978.

[15] Huahai He and Ambuj K. Singh. Graphs-at-a-time: query language and access methods for graph databases. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, page 405–418, New York, NY, USA, 2008. Association for Computing Machinery.

[16] Yang Hu, Pradeep Kumar, Guy Swope, and H. Howie Huang. Trix: Triangle counting at extreme scale. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7, 2017.

[17] Jianqiang Huang, Haojie Wang, Xiang Fei, Xiaoying Wang, and Wenguang Chen. $tc-stream$tc-stream: Large-scale graph triangle counting on a single machine using gpus. *IEEE Transactions on Parallel and Distributed Systems*, 33(11):3067–3078, 2022.

[18] Sitao Huang, Mohamed El-Hadedy, Cong Hao, Qin Li, Vikram S. Mailthody, Ketan Date, Jinjun Xiong, Deming Chen, Rakesh Nagi, and Wen-mei Hwu. Triangle counting and truss decomposition using fpga. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*, pages 1–7, 2018.

[19] Tamara G. Kolda, Ali Pinar, Todd Plantenga, C. Seshadhri, and Christine Task. Counting triangles in massive graphs with mapreduce. *SIAM Journal on Scientific Computing*, 36(5):S48–S77, 2014.

[20] LinkedIn. Linkedin. `https://www.linkedin.com/`. Accessed: [2024-03-21].

[21] Harry M Markowitz. The elimination form of the inverse and its application to linear programming. *Management Science*, 3(3):255–269, 1957.

[22] Kaushik Ravichandran, Akshara Subramaniasivam, P.S. Aishwarya, and N.S. Kumar. Chapter eight - fast exact triangle counting in large graphs using simd acceleration. In Ripon Patgiri, Ganesh Chandra Deka, and Anupam Biswas, editors, *Principles of Big Graph: In-depth Insight*, volume 128 of *Advances in Computers*, pages 233–250. Elsevier, 2023.

[23] Y. Saad. Parallel iterative methods for sparse linear systems. In Dan Butnariu, Yair Censor, and Simeon Reich, editors, *Inherently Parallel Algorithms in Feasibility and Optimization and their Applications*, volume 8 of *Studies in Computational Mathematics*, pages 423–440. Elsevier, 2001.

[24] Ernst Schrem. Computer implementation of the finite-element procedure. In STEVEN J. FENVES, NICHOLAS PERRONE, ARTHUR R. ROBINSON, and WILLIAM C. SCHNOBRICH, editors, *Numerical and Computer Methods in Structural Mechanics*, pages 79–121. Academic Press, 1973.

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

[26] Linghao Song, Yuze Chi, Licheng Guo, and Jason Cong. Serpens: a high bandwidth memory based accelerator for general-purpose sparse matrix-vector multiplication. *Proceedings of the 59th ACM/IEEE Design Automation Conference*, 2021.

[27] Konstantinos Sotiropoulos and Charalampos E. Tsourakakis. Triangle-aware spectral sparsifiers and community detection. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, KDD '21, page 1501–1509, New York, NY, USA, 2021. Association for Computing Machinery.

[28] Konstantinos Sotiropoulos and Charalampos E. Tsourakakis. Triangle-aware spectral sparsifiers and community detection. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, KDD '21, page 1501–1509, New York, NY, USA, 2021. Association for Computing Machinery.

[29] Nitish Srivastava, Hanchen Jin, Jie Liu, David Albonesi, and Zhiru Zhang. Matraptor: A sparse-sparse matrix multiplication accelerator based on row-wise product. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 766–780, 2020.

[30] Ancy Sarah Tom, Narayanan Sundaram, Nesreen K. Ahmed, Shaden Smith, Stijn Eyerman, Midhunchandra Kodiyath, Ibrahim Hur, Fabrizio Petrini, and George Karypis. Exploring optimizations on shared-memory platforms for parallel triangle counting algorithms. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7, 2017.

[31] J. R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 23(1):31–42, jan 1976.

[32] Leyuan Wang, Yangzihao Wang, Carl Yang, and John D. Owens. A comparative study on exact triangle counting algorithms on the gpu. In *Proceedings of the ACM Workshop on High Performance Graph Processing*, HPGP '16, page 1–8, New York, NY, USA, 2016. Association for Computing Machinery.

[33] Xueyan Wang, Jianlei Yang, Yinglin Zhao, Yingjie Qi, Meichen Liu, Xingzhou Cheng, Xiaotao Jia, Xiaoming Chen, Gang Qu, and Weisheng Zhao. Tcim: Triangle counting acceleration with processing-in-mram architecture. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2020.

[34] David S. Watkins. *Fundamentals of Matrix Computations*. John Wiley & Sons, Inc., New York, second edition, 2002.

[35] Michael M. Wolf, Jonathan W. Berry, and Dylan T. Stark. A task-based linear algebra building blocks approach for scalable graph analytics. In *2015 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6, 2015.

[36] Michael M. Wolf, Mehmet Deveci, Jonathan W. Berry, Simon D. Hammond, and Sivasankaran Rajamanickam. Fast linear algebra-based triangle counting with kokkoskernels. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7, 2017.

[37] Xilinx. Vitis libraries - triangle count. https://xilinx.github.io/Vitis_Libraries/graph/2022.1/guide_L2/manual/triangleCount.html, 2022. Accessed: 2024-02-17.

[38] Abdurrahman Yaşar, Sivasankaran Rajamanickam, Jonathan W. Berry, and Ümit V. Çatalyürek. A block-based triangle counting algorithm on heterogeneous environments. *IEEE Transactions on Parallel and Distributed Systems*, 33(2):444–458, 2022.

[39] Abdurrahman Yaşar, Sivasankaran Rajamanickam, Michael Wolf, Jonathan Berry, and Ümit V. Çatalyürek. Fast triangle counting using cilk. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*, pages 1–7, 2018.

[40] Zhitao Ying, Andrew Wang, Jiaxuan You, Chengtao Wen, Arquimedes Canedo, and Jure Leskovec. Neural subgraph matching, 2021.

[41] Peixiang Zhao and Jiawei Han. On graph query optimization in large networks. *Proc. VLDB Endow.*, 3(1–2):340–351, sep 2010.