Triangle Counting and Truss Decomposition using FPGA

Static Graph Challenge: Subgraph Isomorphism

Sitao Huang‡, Mohamed El-Hadedy‡, Cong Hao‡, Qin Li‡, Vikram S. Mailthody‡, Ketan Date*,
Jinjun Xiong†, Deming Chen‡, Rakesh Nagi*, and Wen-mei Hwu‡
‡ECE, *ISE, University of Illinois at Urbana-Champaign, Urbana, IL 61801
†Cognitive Computing Systems Research, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, 10598
Emails: {shuang91, hadedy, congh, qinli2, vsm2, date2}@illinois.edu, jinjun@us.ibm.com, {dchen, nagi, w-hwu}@illinois.edu

***Abstract*—Triangle counting and truss decomposition are two essential procedures in graph analysis. As the scale of graphs grows larger, designing highly efficient graph analysis systems with less power demand becomes more and more urgent. In this paper, we present triangle counting and truss decomposition using a Field-Programmable Gate Array (FPGA). We leverage the flexibility of FPGAs and achieve low-latency high-efficiency implementations. Evaluation on SNAP dataset shows that our triangle counting and truss decomposition implementations achieve 43.5× on average (up to 757.7×) and 6.4× on average (up to 68.0×) higher performance per Watt respectively over GPU solutions.**

***Index Terms*—FPGA, graph algorithms, triangle counting, truss decomposition**

I. Introduction

Triangles and k-truss form basic substructure in almost all social network graphs. A triangle is defined as a cycle of length three whereas a k-truss is defined as a subgraph where every edge belongs to at least k-2 triangles. Triangle counting and k-truss decomposition algorithms are used to analyze the network to generate triangle counts and k-truss subgraphs that could identify cohesive subgroups of individuals. Furthermore, triangle counts calculation forms the fundamental step in calculating metrics such as clustering coefficient and transitivity ratio whereas k-truss provides an approximate idea about the community structure of the graph.

Triangle counting and k-truss problems are often regarded as memory bandwidth intensive in many application scenarios like social network activity analysis. A large number of CPU and GPU based optimization techniques have been proposed [1], [2]. These works show promising results of accelerating triangle counting and k-truss problems with tens to hundreds times of speedup.

As an alternative acceleration platform for computational intensive and latency sensitive applications, the Field Programmable Gate Array (FPGA), is playing a vital role. Compared to GPU, FPGA has lower power consumption with equal or even higher performance potential. Therefore, FPGA usually achieves better performance per watt than GPU [3]. On the other hand, unlike GPUs, FPGA solution requires hardware-design knowledge, which can lead to a higher barrier for wide adoption. One approach to deal with this challenge is to leverage High-Level Synthesis (HLS) for productive FPGA design, a powerful tool that allows FPGA designers to use high-level languages like C/C++ instead of low level hardware description languages. With the help of HLS, FPGA design is greatly simplified [4], [5].

In this paper, we leverage the strength of FPGAs to implement triangle counting and truss decomposition algorithms, aiming to achieve higher power efficiency than GPU platforms. We evaluate two variants of FPGA boards, to demonstrate the form factor, power efficiency and computational capabilities depending on application requirement.

1) Xilinx PYNQ-Z1 FPGA board, a small-scale, low-power and light-weighted System-on-Chip (SoC) platform, which fits well in energy constrained edge computations, like Internet-of-Things (IoT) applications. The power of the board is 2.5 Watt, and the weight is $0.164lbs$ without attachments.

2) Alpha Data ADM-PCIE-7V3 FPGA board, with a Coherent Accelerator Processor Interface (CAPI) connected to an IBM POWER8 machine. The maximum power consumption of the board is 25 Watt.

Our paper's contribution can be summarized as follows.

- To the best of our knowledge, this is the first work that solves triangle counting and truss decomposition problem end-to-end on FPGA platforms.
- Our graph analysis solutions achieve significant higher performance per Watt improvement over GPU solutions. For some graphs our solutions achieve comparable or even higher processing throughput compared to GPU solutions.
- Our implementations target two different FPGA platforms, one for embedded ultra-low-power processing scenarios and the other for high-performance servers.

II. Literature Review

A. Triangle Counting and Truss Decomposition Literatures

In triangle counting, the popular algorithms can be classified into three approaches, namely, matrix multiplication based [6], [7], subgraph matching-based, and set-intersection based approaches [8]. Both sequential and parallel versions of these algorithms have been proposed [2], [9], [10], [11], [12]. In

978-1-5386-5989-2/18/$31.00 ©2018 IEEE

[13], authors discussed the trade-offs of these three approaches in detail and concluded that the set-intersection based algorithm performs the best. We selected the set-intersection based algorithm for FPGA acceleration.

Similarly, efficient fast algorithms to calculate maximal k-truss for a given graph have been under discussion for decades [14]. MapReduce [15], Open-MP and shared memory systems [16], [17], [18], [19], and GPU [2], [20] based fast parallel k-truss decomposition have been proposed in the past. Two different methods are discussed for enumerating k-truss in a graph: Bottom-up approach (increments k starting from 2), and top-down approach (decrements k starting from a guessed large value). In this paper, we use the bottom-up approach based on [2].

B. Solving Graph Problems on FPGA

There are several existing approaches to solve graph problems on FPGA by leveraging its high parallelism and flexibility. Specifically, GraphStep [21] and GraphGen [22] have tried to solve large graph algorithms like page rank, BFS with efficient memory placements and have even built a compiler backend to support easy transition of code base. Similarly, GraphOps [23] provides a hardware library for efficient acceleration of graph algorithms. ForeGraph [24] performs large scale graph processing in multiple FPGAs where efficient partition of graphs is performed such that the communication between the partitions is minimal. However, none of these existing works has explored implementation of triangle counting and truss decomposition algorithms on an FPGA.

III. Algorithms

In this section we present the algorithms we used. We adapt and optimize the algorithms used in [2] so that they are more suitable for FPGAs.

A. Triangle Counting

One of the most widely used sequential triangle counting algorithm is presented in [25] as the *forward* algorithm. We adopt this intersection based method for triangle counting on FPGA, which iterates over each edge and finds common elements from two adjacency lists of head and tail nodes. The triangle counting algorithm takes a graph $G = (V, E)$ in adjacency list format as input, processes the adjacency list, performs set intersection for each edge e to count the number of triangles that contain the edge e, $\Delta(e)$, and finally accumulates these Δ's to get the total triangle count for G, $\Delta(G)$. We assume that each node u in the graph has a *unique* numerical index, denoted as $idx(u)$, and this indexing of nodes defines a total order $\prec$ of nodes, i.e., for any two vertices $u, v \in V$, $idx(u) < idx(v)$ indicates the order $u \prec v$. The preprocessing step constructs a list of directed edges and filters adjacency lists according to this ordering, so that each triangle in the graph is counted only once in the set intersection step. The full triangle counting steps are presented in Algorithm 1. Note that the adjacency lists in the SNAP dataset we use are all sorted by the node index, therefore `SetIntersect` could be done by scanning through two input sets only once. The `SetIntersect` pseudo code is presented in Algorithm 2. The compute complexity of `SetIntersect` is $O(|A_u| + |A_v|)$, where $|A_u|$ and $|A_v|$ are the size of two input sets.

Algorithm 1 `TriangleCount`(G) (intersection-Based)

Input: Graph $G = (V, E)$ in adjacency list format: $\{adj_0(u) = \{v_0^u, v_1^u, \ldots, v_{k^u}^u\} | u \in V\}$
Output: Triangle count $\Delta(G)$

1: $\Delta(G) \leftarrow 0$, $E^* \leftarrow \emptyset$, $adj^*(e) \leftarrow \emptyset, \forall e \in E$
2: **for each** vertex $u \in V$ **do**
3: **for each** neighbor $v \in adj_0(u)$ **do**
4: **if** $idx(u) > idx(v)$ **then** $E^* \leftarrow E^* \cup \{(u, v)\}$
5: **if** $idx(u) < idx(v)$ **then** $adj^*(u) \leftarrow adj^*(u) \cup \{v\}$
6: **end for**
7: **end for**
8: **for each** edge $(u, v) \in E^*$ **do**
9: $\Delta(G) \leftarrow \Delta(G)$+`SetIntersect`$(adj^*(u), adj^*(v))$
10: **end for**

Algorithm 2 `SetIntersect`(A_u, A_v) (sorted input sets)

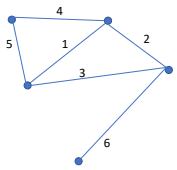
Input: sorted sets A_u, A_v
Output: size of intersection of A_u and A_v: $\Delta(e = (u, v))$

1: $i_u \leftarrow 0$, $i_v \leftarrow 0$
2: **while** $i_u < |A_u|$ and $i_v < |A_v|$ **do**
3: **if** $A_u[i_u] < A_v[i_v]$ **then** $i_u \leftarrow i_u + 1$
4: **if** $A_u[i_u] > A_v[i_v]$ **then** $i_v \leftarrow i_v + 1$
5: **if** $A_u[i_u] == A_v[i_v]$ **then**
6: $i_u \leftarrow i_u + 1$, $i_v \leftarrow i_v + 1$
7: $\Delta(e) \leftarrow \Delta(e) + 1$
8: **end if**
9: **end while**

B. Truss Decomposition

The second static graph challenge aims to discover the k-truss for all $2 \leq k \leq k_{max}$, where k_{max} is the maximal k such that k-truss is not an empty set. Given a graph G, a k-truss is defined as a subgraph of G where each edge in this subgraph is contained in at least $(k-2)$ triangles in the subgraph [26]. Our truss decomposition algorithm is shown in Algorithm 3 and has the same input format as triangle counting. The algorithm consists of two parts, initial triangle counting (line 1-13) and triangle count updates (line 14-32). The initial triangle counting part slightly differs from Algorithm 1. In the preprocessing of this initial triangle counting, instead of filtering adjacency lists as in Algorithm 1, we use the full adjacency list $adj_0(u)$. This provides an easy way to keep track of triangle count on both forward and backward edges for k-truss algorithm. Besides, in the initial triangle counting, for each edge, we stored the list of triangles that contain the edge (line 8). In our truss decomposition algorithm, we use edge-centric indexing where we assign indices to edges (line 21-31). This is different from the node indexing in the input graph file

978-1-5386-5989-2/18/$31.00 ©2018 IEEE



Triangles list:

Edge	Triangles
1:	(2, 3), (4, 5)
2:	(1, 3)
3:	(1, 2)
4:	(1, 5)
5:	(1, 4)
6:	-

Triangle Count list:

Edge	Triangle Count
1:	2
2:	1
3:	1
4:	1
5:	1
6:	0

Fig. 1. Example Data Structures for Truss Decomposition

and in Algorithm 1. Indexing edges simplifies edge lookup, reduces random accesses into DRAM, and speeds up edge lookup in all data structures used in the truss decomposition algorithm. Fig. 1 gives an example of data structures used in our truss decomposition implementation.

Algorithm 3 `TrussDecompose`(G)

Input: Graph $G = (V, E)$ in adjacency list format: $\{adj_0(u) = \{v_0^u, v_1^u, \ldots, v_{k^u}^u\} | u \in V\}$

Output: k-truss for $2 \leq k \leq k_{max}$

1: **for each** edge $e = (u, v) \in E$ s.t. $u > v$ **do**
2: $i_u \leftarrow 0$, $i_v \leftarrow 0$, $\Delta(e) \leftarrow 0$, $S_\Delta(e) \leftarrow \emptyset$
3: **while** $i_u < |adj_0(u)|$ and $i_v < |adj_0(v)|$ **do**
4: **if** $adj_0(u)[i_u] < adj_0(v)[i_v]$ **then** $i_u \leftarrow i_u + 1$
5: **if** $adj_0(u)[i_u] > adj_0(v)[i_v]$ **then** $i_v \leftarrow i_v + 1$
6: **if** $adj_0(u)[i_u] == adj_0(v)[i_v]$ **then**
7: $e_1 \leftarrow (adj_0(u)[i_u], u)$, $e_2 \leftarrow (adj_0(u)[i_u], v)$
8: $S_\Delta(e) \leftarrow S_\Delta(e) \cup \{(e_1, e_2)\}$
9: $i_u \leftarrow i_u + 1$, $i_v \leftarrow i_v + 1$
10: $\Delta(e) \leftarrow \Delta(e) + 1$
11: **end if**
12: **end while**
13: **end for**
14: $EdgeExists \leftarrow True$, $NewDeletes \leftarrow False$, $k \leftarrow 2$
15: **while** $EdgeExists$ **do**
16: **if** $NewDeletes == False$ **then**
17: Output current graph as k-truss subgraph
18: $k \leftarrow k + 1$
19: **end if**
20: $EdgeExists \leftarrow False$, $NewDeletes \leftarrow False$
21: **for each** edge $e \in E$ **do**
22: **if** $0 < \Delta(e) < (k - 2)$ **then**
23: $\Delta(e) \leftarrow 0$, $NewDeletes \leftarrow True$
24: **for each** $(e_1, e_2) \in S_\Delta(e)$ **do**
25: Delete any (e_a, e_b) in $S_\Delta(e_1)$ and $S_\Delta(e_1)$ that contains e
26: $\Delta(e_1) \leftarrow \Delta(e_1) - 1$, $\Delta(e_2) \leftarrow \Delta(e_2) - 1$
27: **end for**
28: **else if** $\Delta(e) > (k - 2)$ **then**
29: $EdgeExists \leftarrow True$
30: **end if**
31: **end for**
32: **end while**

IV. Graph Analysis with PYNQ SoC

A. PYNQ SoC Overview

The PYNQ board contains a Xilinx Zynq-7000 SoC, which has embedded ARM cores and programmable logic. The

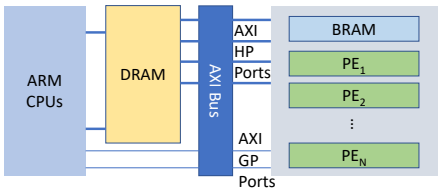


Fig. 2. PYNQ SoC Design Overview

overview of our SoC design is shown in Fig. 2. The processing elements (PEs) in FPGA is connected to external on-board DRAM via Xilinx AXI High Performance ports. The embedded ARM CPU and FPGA share the virtual memory space.

B. FPGA Processing Element (PE) Design

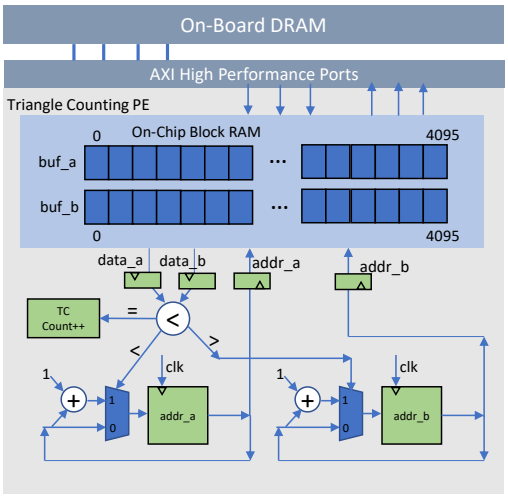


Fig. 3. Triangle Counting PE Structure

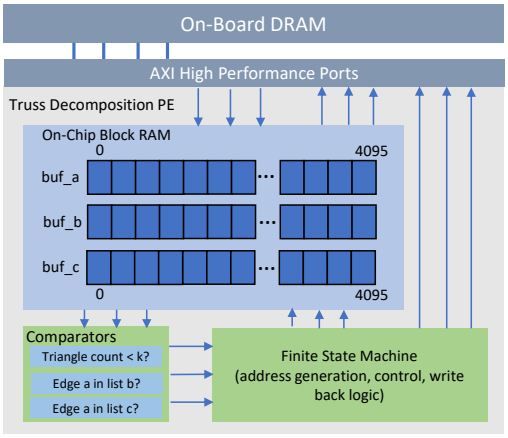


Fig. 4. Truss Decomposition PE Structure

We map triangle counting and truss decomposition computation to processing elements (PEs) implemented on FPGA. A PE is a hardware block that can work on either triangle counting or truss decomposition algorithm independently. Further, PEs can be duplicated to create multiple instances, a notion similar to parallel threads.

978-1-5386-5989-2/18/$31.00 ©2018 IEEE

For triangle counting, each PE works on a subset of edge list (lines 8-10 in Algorithm 1). Fig. 3 depicts the PE structure. Note that the datapath that starts from on-chip Block RAM (BRAM), then comparator, address increment units, and to address port are fully pipelined, that means all the function units in the circuit are fully utilized during execution and therefore highly efficient. Given an edge, our PE reads the adjacency lists of the two endpoints from the on-board DRAM and stores them into on-chip BRAM, and then computes the size of the intersection of those two adjacency lists. The output provides the number of triangles associated with that edge.

For truss decomposition, preprocessing and initial triangle counting are done by the embedded ARM cores. PE on the FPGA performs the actual truss decomposition (lines 14-32 in Algorithm 3). Fig. 4 shows the PE structure. The datapath consisting comparators and the address generation units are fully pipelined. The PE checks the triangle count list to see whether each edge has more than $k-2$ triangles. If no, the PE marks current edge as deleted, deletes all the edges in the associated triangles (affected edges) by 1, and updates the triangles list of those affected edges. The whole triangle count list and triangles list are stored in the on-board DRAM, and the data for the edge under processing is buffered in BRAM dynamically. Our PE is designed in C, and synthesized into circuit by Xilinx Vivado High-Level Synthesis (HLS) [27].

C. BRAM Buffering Scheme

Accesses to external DRAMs are expensive for FPGA. In our PE design, to reduce the off-chip DRAM accesses, we use on-chip FPGA BRAMs to buffer the adjacency lists before performing set intersection on the adjacency lists. This strategy not only reduces original random accesses to external DRAM, but also enables data in the burst access mode. This scheme further enables to exploit the locality provided by the neighbor nodes in buffered adjacency list efficiently.

D. Multi-PE Design and Workload Balancing

Given the constrained area, we can fit at most 8 PEs in the PYNQ board. Each of the PEs works on a part of the input edge list providing parallelism. We dispatch edges in the list to PEs using round-robin arbitration scheme. However, for neighbor edges, we make sure they are mapped to the same PE. These steps ensure the workload of PEs are balanced while best reusing data. For truss decomposition acceleration, it is hard to partition input/output set and distribute partitions across PEs. This is because multiple PEs could be updating the same memory location at the same time and an efficient atomic access scheme is needed. Depending on the implementation, this could introduce extra performance overhead. In this work we use 1 PE for truss decomposition.

V. CAPI Triangle Counting Integration

Although FPGAs offer flexibility to provide customized design for specific algorithms, they have limited fast on-chip memory. To circumvent this challenge, we leverage IBM's Coherent Accelerator Processor Interface CAPI [28], that allows

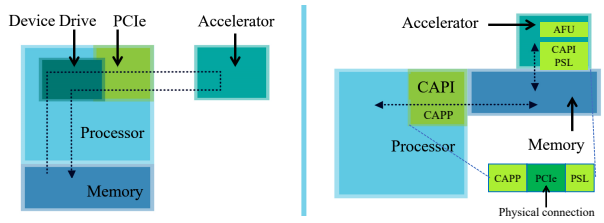


Fig. 5. Traditional Architecture (left) vs CAPI Architecture (right)

direct cache coherent interface to the CPU memory banks. As shown in Fig. 5, CAPI allows the FPGA to access the memory banks of the CPU side with shorter latency compared to the traditional architecture.

CAPI has a host side (Coherent Attached Processor Proxy (CAPP)) and an FPGA side (Power Service Layer (PSL)) driver proxies. Application functional units (AFU), the hardware accelerators, interact with the PSL whenever they require to access a data present in the host DRAM. In our design, the AFUs are group of triangle counting PEs with the associated modules to fetch the required data from the CPU side. Both CPU and FPGA can access the memory at the same time but with additional cost on maintaining coherency.

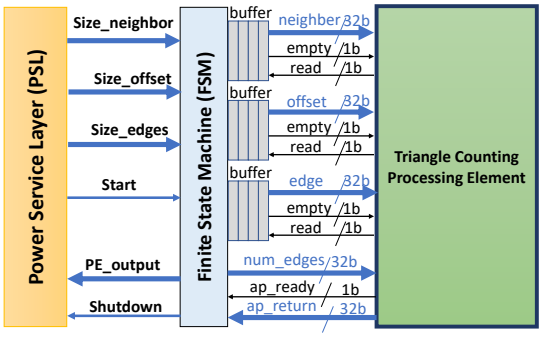


Fig. 6. Triangle Counting-CAPI Communication

Figure 6 shows interaction of PSL with Triangle counting accelerator that was created using HLS. PSL provides, the size of the neighbor, offset, and edges to the Processing element through a 13 state Mealy finite state machine (FSM). Once the start signal comes from the PSL, which is directed by the CAPP (the CPU-CAPI proxy), the Finite state machine receives the 32-bit num_edges (register-based). After that, the neighbor data will be transferred from the memory banks of the CPU side to the neighbor buffer on the FPGA. In the current design, the neighbor buffer hosts 128 bytes and divides them into 32 patches of 32-bit. The data transfer to the inner PE's FIFO is based on the read signal, which directs the FSM to enable the empty signal or disable it. When the empty signal is high, it indicates that data transfer is needed and when it goes down, it means that FIFO is full and there is no need for more data transfer until the PE finishes the computation. While the data transfer is performed, there is a counter inside the FSM that is decremented from the size of the neighbor to zero. When the counter reaches zero, the neighbor data movement with CAPI is over, and the same process will happen with the offset. Once the offset transfer process is done, the edge part

978-1-5386-5989-2/18/$31.00 ©2018 IEEE

will start. The ap_ready signal is an output flag, which directs the FSM to take the ap_return value (a 32-bit outputvalue of the PE) back to the CPU to be stored in the memory bank. After transferring the data back to the CPU, the FSM will issue the shutdown signal to close both sides of the CAPI bridge (PSL and CAPP). Afterwards, a couple of states will be traversed to request and load the data through CAPI-DMA into the PE's buffer. Once the output of the PE is generated, there is a state for moving the output back to the CPU memory side and closing the bridge between CAPP and PSL.

VI. Experimental Results

A. Experimental setup

We evaluated our triangle counting and truss decomposition implementations using real-world graphs from the SNAP dataset [29]. We tested our design on Xilinx PYNQ board and measured the processing time and power consumption for each graph. The performance of our implementation is compared with three existing implementations: sequential Python baseline provided by the Graph Challenge organizer [26], OpenMP implementation, and single GPU implementation from [2]. These are the implementations with highest performance in [2]. The experiments in [2] are conducted on the IBM Minsky machine two 80-core 4.02 GHz Power8 CPUs and four NVIDIA Tesla P100 GPUs [2].

On-board experiments are done with Xilinx PYNQ-Z1 FPGA, a low-power low-cost embedded SoC board. It contains a Xilinx Zynq-7000 SoC, 512MB DDR3 memory, and MicroSD card as storage. Inside the Zynq-7000 SoC there are embedded dual-core ARM Cortex-A9 processor (called processing system, PS) and programmable logic (PL), which includes 53.2k Look-Up Tables (LUTs), 220 DSPs, and 4.9Mbits fast on-chip Block RAM. The embedded ARM cores run at 650MHz clock frequency. PS runs an Ubuntu 16.04 operating system. The PS and PL share the virtual memory space which simplifies programming and data sharing in the SoC. The PL circuit is synthesized from C code using Vivado HLS and Vivado 2017.2. Our triangle counting implementation on PYNQ contains eight triangle counting PEs, while our truss decomposition implementation contains only one PE.

We also integrate our triangle counting accelerator with IBM CAPI, and targets IBM Power8 machine with FPGAs. The simulation results will be presented later in this section.

B. Triangle Counting on PYNQ SoC

In Table I and II, **Python** is the serial Python baseline provided by the challenge organizer, **OMP** is the OpenMP implementation on Power8 machine from [2] which uses all 160 CPU cores. **GPU** is the single Tesla P100 GPU implementation from [2], which is the fastest implementation in [2]. **Speedup over** is the speedup achieved by our PYNQ implementation over these three previous implementations. Note that the execution time of our work includes time of preprocessing, memory allocation, memory copy and execution. **Perf/Watt over GPU** is the performance per Watt number of our whole PYNQ implementation over that of the single P100 GPU implementation. We use the estimate of 250 Watts for the IBM Minsky machine with one single P100 GPU worker [30]. We use a USB power meter to measure the power consumption of PYNQ board during execution. The average PYNQ power consumption we got from the measurement is 2.59 Watts.

From Table I, PYNQ implementation outperforms Python baseline, OpenMP and GPU by up to 68.47×, 12.75× and 7.85× respectively. For graphs with larger number of edges, our speed up is lower compared to OMP and GPUs, but in terms of performance per Watt, we outperform from GPU significantly with higher power efficiency by 43.5× (geomean).

For larger graphs, performance limitation is due to frequent data movement between on-board DRAM and on-chip BRAM. To understand this further, we measure execution time break down of our implementation as shown in Fig. 7. *Preprocessing* is the step where PS CPU constructs filtered edge list and adjacency list as explained in Section III. *Malloc* is the time when PS CPU allocates arrays in the shared virtual memory space. *Memcpy* loads input data to the allocated shared memory space. All these three steps are done on the PS CPU side. *Kernel* is the actual PL FPGA execution time. On average 50% of time is spent on triangle counting kernel execution. This indicates that pipelined kernel execution is possible where the CPU does preprocessing while FPGA does the kernel execution for previously preprocessed data. This could be considered as a type of CPU+FPGA collaborative task partitioning scheme and is left as future work.

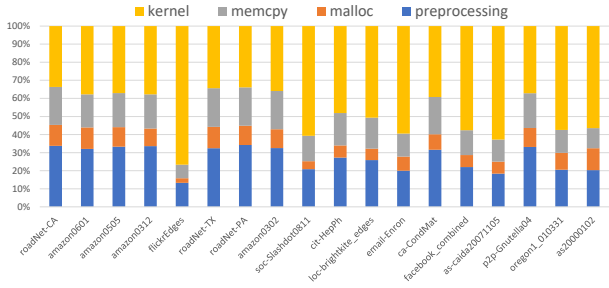


Fig. 7. Triangle Counting Execution Time Breakdown

C. Truss Decomposition on PYNQ SoC

Table II shows the truss decomposition performance of our PYNQ implementation and previous ones [2]. Our implementation outperforms the Python baseline by up to 74.3×, and it outperforms OMP on graphs with lower number of edges. Given that the FPGA does not have dedicated caches and also due to random memory access present in the k-truss kernel, the performance gain is limited. However, our implementation beats GPU ones in terms of power efficiency by 6.4× (geomean), which is noteworthy. Figure 8 in our truss decomposition design, most of the total execution time goes to kernel. We still could overlap preprocessing and FPGA computation to reduce batch processing latency.

D. Triangle Counting on Power Machine with CAPI

We integrated our triangle counting design with IBM CAPI and collected preliminary simulation results due to limited

978-1-5386-5989-2/18/$31.00 ©2018 IEEE

TABLE I
TRIANGLE COUNTING PERFORMANCE COMPARISON OF OUR DESIGN AGAINST P100 GPU AND MULTI-CORE POWER CPU ON REAL-WORLD GRAPHS

Graph	#node	#edge	Triangle Count	Execution Time (s)				Speedup over			Perf/Watt over GPU
				Python	OMP	GPU	This Work	Python	OMP	GPU	
roadNet-CA	1,965,206	5,533,214	120,676	3.543	0.29	0.18	2.342	1.513	0.124	0.077	7.419
amazon0601	403,394	4,886,816	3,986,507	23.036	0.264	0.163	2.011	11.457	0.131	0.081	7.825
amazon0505	410,236	4,878,874	3,951,063	23.203	0.264	0.162	1.958	11.849	0.135	0.083	7.985
amazon0312	400,727	4,699,738	3,686,467	22.058	0.295	0.162	1.880	11.735	0.157	0.086	8.319
flickrEdges	105,938	4,633,896	107,981,213	306.638	0.403	0.188	4.479	**68.469**	0.090	0.042	4.052
roadNet-TX	1,379,917	3,843,320	82,869	2.386	0.255	0.173	1.586	1.505	0.161	0.109	10.531
roadNet-PA	1,088,092	3,083,796	67,150	1.946	0.254	0.169	1.291	1.507	0.197	0.131	12.631
amazon0302	262,111	1,799,584	717,719	3.181	0.228	0.157	0.662	4.803	0.344	0.237	22.881
soc-Slashdot0811	77,360	938,360	551,724	24.388	0.228	0.155	0.499	48.882	0.457	0.311	29.988
cit-HepPh	34,546	841,754	1,276,868	8.578	0.264	0.154	0.343	24.980	0.769	0.448	43.288
loc-brightkite_edges	58,228	428,156	494,728	4.146	0.21	0.147	0.193	21.498	1.089	0.762	73.575
email-Enron	36,692	367,662	727,044	8.153	0.272	0.146	0.220	37.025	1.235	0.663	63.998
ca-CondMat	23,133	186,878	173,361	0.57	0.218	0.154	0.070	8.188	3.131	2.212	213.529
facebook_combined	4,039	176,468	1,612,010	2.629	0.232	0.15	0.093	28.358	2.502	1.618	156.177
as-caida20071105	26,475	106,762	36,365	4.286	0.264	0.147	0.072	59.144	3.643	2.029	195.803
p2p-Gnutella04	10,876	79,988	934	0.161	0.234	0.144	0.032	5.016	7.291	4.487	433.081
oregon1_010331	10,670	44,004	17,144	1.518	0.242	0.144	0.028	53.933	8.598	5.116	493.835
as20000102	6,474	25,144	6,584	0.538	0.234	0.144	0.018	29.326	**12.755**	**7.849**	**757.651**
geomean	-	-	-	-	-	-	-	**13.386**	**0.732**	**0.450**	**43.473**

TABLE II
TRUSS DECOMPOSITION PERFORMANCE COMPARISON AGAINST P100 GPU AND MULTI-CORE POWER CPU ON REAL-WORLD GRAPHS

Graph	#node	#edge	k_{max}	Execution Time (s)				Speedup over			Perf/Watt over GPU
				Python	OMP	GPU	This Work	Python	OMP	GPU	
roadNet-CA	1,965,206	5,533,214	4	526.181	0.680	0.249	7.083	**74.291**	0.096	0.035	3.393
amazon0505	410,236	4,878,874	11	2,666.413	5.395	1.717	110.833	24.058	0.049	0.015	1.495
amazon0312	400,727	4,699,738	11	2,213.735	5.557	1.151	97.622	22.677	0.057	0.012	1.138
roadNet-TX	1,379,917	3,843,320	4	368.975	0.597	0.226	7.083	52.096	0.084	0.032	3.080
roadNet-PA	1,088,092	3,083,796	4	295.109	0.579	0.209	6.446	45.780	0.090	0.032	3.130
amazon0302	262,111	1,799,584	7	306.633	1.562	0.366	14.634	20.953	0.107	0.025	2.414
soc-Slashdot0811	77,360	938,360	35	2,863.684	12.392	1.671	61.127	46.848	0.203	0.027	2.639
cit-HepPh	34,546	841,754	25	1,888.288	12.265	1.785	50.878	37.114	0.241	0.035	3.386
cit-HepTh	27,770	704,570	30	2,387.755	14.477	3.199	56.537	42.233	0.256	0.057	5.462
loc-brightkite_edges	58,228	428,156	43	1,498.010	11.84	1.786	29.233	51.243	0.405	0.061	5.897
ca-AstroPh	18,772	396,100	57	854.938	7.072	2.066	30.706	27.842	0.230	0.067	6.494
email-Enron	36,692	367,662	22	1,053.504	9.681	2.975	22.223	47.407	0.436	0.134	12.922
ca-HepPh	12,008	236,978	239	1,080.121	8.376	1.809	71.153	15.180	0.118	0.025	2.454
ca-CondMat	23,133	186,878	26	109.940	2.643	0.394	4.090	26.880	0.646	0.096	9.299
facebook_combined	4,039	176,468	97	1,235.489	26.478	6.593	40.258	30.690	0.658	0.164	15.808
as-caida20071105	26,475	106,762	16	143.366	3.405	0.380	2.541	56.415	1.340	0.150	14.434
p2p-Gnutella04	10,876	79,988	4	3.820	0.380	0.152	0.216	17.717	1.762	**0.705**	**68.046**
oregon1_010331	10,670	44,004	16	39.433	2.426	0.275	0.920	42.857	2.637	0.299	28.849
as20000102	6,474	25,144	10	12.128	1.472	0.207	0.308	39.414	**4.784**	0.673	64.934
geomean	-	-	-	-	-	-	-	**34.959**	**0.305**	**0.066**	**6.412**

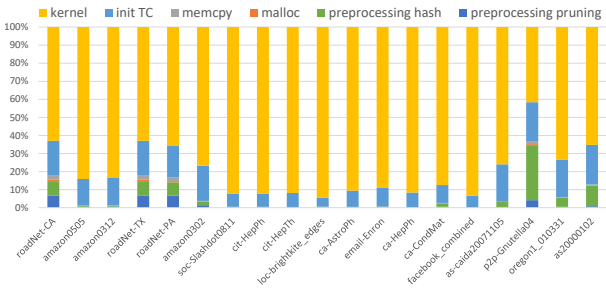


Fig. 8. Truss Decomposition Execution Time Breakdown

time. We targeted Alpha Data 7V3 FPGA board for IBM Power8 machine. Our synthesis results showed that up to 40 PEs can fit into 7V3 FPGA. From simulation the data loading time from CPU cache line to FPGA PE is 5.63ms, triangle counting time for a single PE running at 100MHz for soc-Epinions1 graph is 3.77s, and with 40 PEs triangle counting time can be reduced down to 9.25ms, which is 16.8× faster than the P100 GPU. Vivado power analysis shows that the power consumption for the FPGA chip itself is 10.7 Watt, and BRAM and signals make the majority (65%) of dynamic power consumption. In terms of Perf/Watt, it's 392.5× better than P100 GPU. We are still exploring the extra optimization space enabled by CAPI.

VII. CONCLUSION

To the best of our knowledge, we are the first to present FPGA based accelerators for triangle counting and k-truss decomposition algorithms. The FPGA based system provides 43.5× and 6.4× higher performance per Watt compared to GPU solutions for the two algorithms respectively, making it an ideal candidate for high-efficiency graph analysis.

978-1-5386-5989-2/18/$31.00 ©2018 IEEE

ACKNOWLEDGMENTS

This work is supported by IBM-ILLINOIS Center for Cognitive Computing Systems Research (C3SR) - a research collaboration as part of the IBM AI Horizons Network.

REFERENCES

[1] Chad Voegele, Yi-Shan Lu, Sreepathi Pai, and Keshav Pingali. Parallel triangle counting and k-truss identification using graph-centric methods. In *High Performance Extreme Computing Conference (HPEC), 2017 IEEE*, pages 1–7. IEEE, 2017.

[2] K. Date, K. Feng, R. Nagi, J. Xiong, N. S. Kim, and W. M. Hwu. Collaborative (CPU + GPU) algorithms for triangle counting and truss decomposition on the Minsky architecture: Static graph challenge: Subgraph isomorphism. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7, Sept 2017.

[3] Shuichi Asano, Tsutomu Maruyama, and Yoshiki Yamaguchi. Performance comparison of FPGA, GPU and CPU in image processing. In *Field programmable logic and applications, 2009. fpl 2009. international conference on*, pages 126–131. IEEE, 2009.

[4] Yun Liang, Kyle Rupnow, Yinan Li, Dongbo Min, Minh N. Do, and Deming Chen. High-level synthesis: Productivity, performance, and software constraints. *JECE*, 2012:1:1–1:1, January 2012.

[5] Keith Campbell, Wei Zuo, and Deming Chen. New advances of high-level synthesis for efficient and reliable hardware design. *Integration, the VLSI Journal*, 58:189–214, 6 2017.

[6] A. Azad, A. Bulu, and J. Gilbert. Parallel triangle counting and enumeration using matrix algebra. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, pages 804–811, May 2015.

[7] M. Bisson and M. Fatica. Static graph challenge on gpu. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–8, Sept 2017.

[8] Matthieu Latapy. Main-memory triangle computations for very large (sparse (power-law)) graphs. *Theoretical Computer Science*, 407(1):458 – 473, 2008.

[9] Shumo Chu and James Cheng. Triangle listing in massive networks and its applications. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 672–680. ACM, 2011.

[10] Oded Green, Pavan Yalamanchili, and Lluís-Miquel Munguía. Fast triangle counting on the GPU. In *Proceedings of the 4th Workshop on Irregular Applications: Architectures and Algorithms*, pages 1–8. IEEE Press, 2014.

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

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

[13] 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*, pages 1–8. ACM, 2016.

[14] Jonathan Cohen. Trusses: Cohesive subgraphs for social network analysis. *National Security Agency Technical Report*, page 16, 2008.

[15] Jonathan Cohen. Graph twiddling in a MapReduce world. *Computing in Science & Engineering*, 11(4):29–41, 2009.

[16] Pei-Ling Chen, Chung-Kuang Chou, and Ming-Syan Chen. Distributed algorithms for k-truss decomposition. In *Big Data (Big Data), 2014 IEEE International Conference on*, pages 471–480. IEEE, 2014.

[17] H. Kabir and K. Madduri. Shared-memory graph truss decomposition. In *2017 IEEE 24th International Conference on High Performance Computing (HiPC)*, pages 13–22, Dec 2017.

[18] H. Kabir and K. Madduri. Parallel k-truss decomposition on multicore systems. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7, Sept 2017.

[19] S. Smith, X. Liu, N. K. Ahmed, A. S. Tom, F. Petrini, and G. Karypis. Truss decomposition on shared-memory parallel systems. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6, Sept 2017.

[20] O. Green, J. Fox, E. Kim, F. Busato, N. Bombieri, K. Lakhotia, S. Zhou, S. Singapura, H. Zeng, R. Kannan, V. Prasanna, and D. Bader. Quickly finding a truss in a haystack. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7, Sept 2017.

[21] M. deLorimier, N. Kapre, N. Mehta, D. Rizzo, I. Eslick, R. Rubin, T. E. Uribe, T. F. Jr. Knight, and A. DeHon. Graphstep: A system architecture for sparse-graph algorithms. In *2006 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 143–151, April 2006.

[22] E. Nurvitadhi, G. Weisz, Y. Wang, S. Hurkat, M. Nguyen, J. C. Hoe, J. F. Martnez, and C. Guestrin. Graphgen: An fpga framework for vertex-centric graph computation. In *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 25–28, May 2014.

[23] Tayo Oguntebi and Kunle Olukotun. Graphops: A dataflow library for graph analytics acceleration. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '16, pages 111–117, New York, NY, USA, 2016. ACM.

[24] Guohao Dai, Tianhao Huang, Yuze Chi, Ningyi Xu, Yu Wang, and Huazhong Yang. Foregraph: Exploring large-scale graph processing on multi-fpga architecture. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '17, pages 217–226, New York, NY, USA, 2017. ACM.

[25] Thomas Schank and Dorothea Wagner. Finding, counting and listing all triangles in large graphs, an experimental study. In *International Workshop on Experimental and Efficient Algorithms*, pages 606–609. Springer, 2005.

[26] Siddharth Samsi, Vijay Gadepally, Michael Hurley, Michael Jones, Edward Kao, Sanjeev Mohindra, Paul Monticciolo, Albert Reuther, Steven Smith, William Song, Diane Staheli, and Jeremy Kepner. Static graph challenge: Subgraph isomorphism. In *IEEE HPEC*, 2017.

[27] Xilinx Vivado High-Level Synthesis. https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html.

[28] J. Stuecheli, B. Blaner, C. R. Johns, and M. S. Siegel. CAPI: A Coherent Accelerator Processor Interface. *IBM J. Research and Develop.*, 59(1):7:1–7:7, Jan 2015.

[29] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

[30] NVIDIA P100 GPU. https://images.nvidia.com/content/tesla/pdf/nvidia-tesla-p100-PCIe-datasheet.pdf.

978-1-5386-5989-2/18/$31.00 ©2018 IEEE