Scaling Triangle Counting and K-Truss on the UpDown Architecture

Jiya Su
Department of Computer Science
University of Chicago
Chicago, IL, USA
Email: jiya@uchicago.edu

Alexander Fell
Department of Computer Science
University of Chicago
Chicago, IL, USA
Email: alefel@uchicago.edu

Andronicus Rajasukumar
Department of Computer Science
University of Chicago
Chicago, IL, USA
Email: andronicus@uchicago.edu

David F. Gleich
Computer Science Department
Purdue University
West Lafayette, IN, USA
dgleich@purdue.edu

Andrew A. Chien
Department of Computer Science, University of Chicago
and MCS Division, Argonne National Lab
Chicago, IL, USA
aachien@uchicago.edu

2025 IEEE High Performance Extreme Computing Conference (HPEC) | 979-8-3315-7844-2/25/$31.00 ©2025 IEEE | DOI: 10.1109/HPEC67600.2025.11196311

***Abstract*—UpDown is a novel computing architecture that features accelerators with a custom architecture to make fine-grained parallel algorithms possible. It integrates into a system with a global shared memory address space that was designed to be extremely flat in terms of system resources, with off-node data access comparable to on-node access. This setup enables algorithms and software to scale from single nodes to an entire cluster easily. We demonstrate that UpDown's performance on the Graph Isomorphism Graph Challenge benchmark, utilizing a system designed under IARPA's AGILE program, fulfills this promise. For TC, simulation of the UpDown system predicts excellent scaling to over 900-fold speedup while using only a single global copy of the graph. For K-truss, on a single node, we are faster than a comparable GPU, and again with a single global copy of the graph, report the first scale-out performance, scaling to up to 16-fold performance on 16 nodes, and 232-fold on 256 nodes.**

***Index Terms*—graph algorithms, triangle counting, k-truss, parallel algorithms**

I. Introduction

The AGILE program for IARPA set out an ambitious challenge: invent new computer architectures that would enable graph algorithms to scale and achieve up to $100\times$ faster than reference CPU-based systems at similar or lower power levels [1]. Key aspects of the challenge included addressing memory hierarchy, irregular access, load balancing, and communication overheads that cause low performance on existing system architectures. The AGILE architectures were to be demonstrated and validated across a range of workflows and benchmarks [2]. The UpDown architecture [3]–[7] has been developed over the past few years through this program. On challenging graph applications such as PageRank and BFS, UpDown is projected to exceed the capabilities of current systems [8] substantially.

The hardware features of UpDown enable direct exploitation of vertex and edge parallelism in graph applications. Novel architecture features in each computing lane make fine-grained parallelism (10-100 instructions) efficient (a lane is the fundamental computing unit in UpDown, akin to a core). This results in achieving high compute utilization and tolerance to irregularity in graphs. Specific key features include: event-driven scheduling, split-transaction memory access, and software-managed lightweight threading. Together, this hardware support reduces the cost of a task or message to only a single cycle [3], [4].

To enable scaling to large graphs, UpDown provides extremely high network bandwidth (4.4 TB/s per node, 32 PB/s bisection) and low latency (1.1 microseconds for remote memory access). UpDown supports a sophisticated global address space, so combined with low latency, many irregular graph applications can view it as a global memory [3], [5], [6].

We created programs for UpDown for triangle counting (TC) and K-Truss for the Graph Challenge Subgraph Isomorphism challenge. The K-Truss implementation was built in a few days, leveraging the programming infrastructure developed for AGILE (see further discussion below), as well as the existing TC code. We evaluated them on our instruction-level cycle-accurate simulator to derive estimates of the performance for an UpDown system with up to 1024 nodes. In all of our simulations, the graph is present in the global memory *once* and the data are striped across the computation nodes. This is a notable difference from recent GPU Graph Challenge champion entries, which replicate data across GPUs [9].

Our results include:

- A single UpDown node computes TC on the friendster graph with 1.8 billion edges in 1,342 ms.
- A single UpDown node computes K-Truss on the cit-Patents graph in 2ms ($k = 3$) and 2.6ms ($k = k_{\max}$).
- With 1024 UpDown nodes, we see a 790 times speedup friendster with only a single copy of the graph shared

Supported by IARPA AGILE Program.

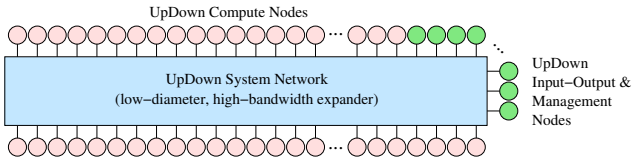

Fig. 1. UpDown System: 16,384 compute nodes, low-diameter Polarstar Network [11].

across the distributed computation. For a scale 28 RMAT graph, a 900 times speedup is achieved.

- For TC, a single UpDown node compared with the recent MERCURY Graph Challenge Champion [9] yields a geomean speedup of around 1.98 (based on 37 graphs) from the single A100 GPU, despite using simulations based on a much older silicon process (16nm for UpDown vs. 7nm for A100).
- For K-Truss, a single UpDown node compared with prior Graph Challenge winner [10], shows higher performance: 2.73× for k=3 (geomean) and 3.36× for $k = k_{\max}$ (geomean) vs a single V100 GPU.
- We also report the first K-Truss scale-out speedup number for the Graph Challenge, reaching 16-fold speedup on 16 UpDown nodes, and 232-fold on 256 nodes.

The remainder of the paper provides additional details on all of these points. We provide more information about the UpDown accelerator architecture, the UpDown system for AGILE, and programming tools for UpDown in Section II. We describe the implementations of TC and K-Truss in Section III. Our performance results are presented in Section IV, along with a comparison to recent Graph Challenge champions. Additional context and discussion of our results – with a focus on distributed implementations and the simulated nature of our results – are in Section V.

II. The UpDown Architecture and System

A. UpDown System

The UpDown system is a codesigned scalable graph supercomputer [3]–[7] and has been designed as part of IARPA AGILE program, an ambitious effort to create new building blocks with breakthrough capability for graph computing [1]. Key aspects of the system's capabilities include: global address space, extraordinary communication performance (efficient 64-byte messages, low-latency of 0.5 microseconds from the Polarstar topology, a diameter-3 direct network based on expander graphs [11]), large communication capacity (4.4TB/s injection per node and 32PB/s bisection bandwidth), high memory bandwidth (153PB/s/system, 9.4TB/s/node), and efficient execution of fine-grained parallelism (10-100 instructions). These capabilities enable UpDown to exploit vertex and edge parallelism in algorithms directly [7], [8], and to write programs that assume a nearly flat memory space. The global system view is depicted in Figure 1. UpDown addresses key systems-level challenges for global shared memory process isolation and memory translation, in other work [6].

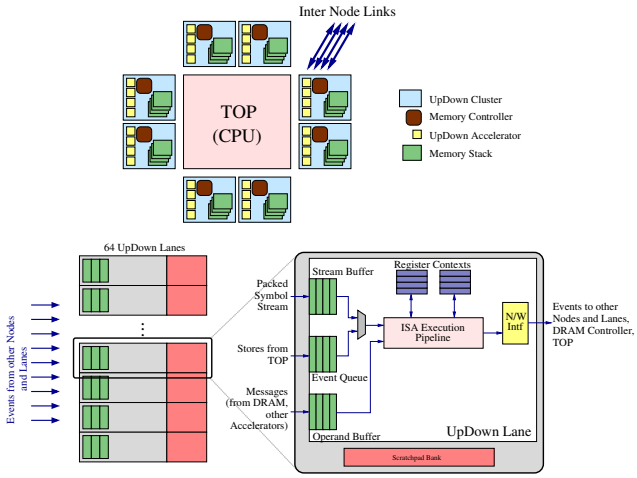

Fig. 2. Each UpDown Node (top) has 32 accelerators; each Accelerator has 64 lanes, designed for event-driven execution (bottom)

B. Node-level Design and Fine-grained Parallelism

Each node in the system integrates 32 UpDown accelerators, organized alongside 8 stacks of high-bandwidth memory (HBM), with each HBM stack shared by 4 accelerators, as illustrated in Figure 2 (top). Every accelerator contains 64 lanes consisting of 2GHz MIMD compute engines that execute programs in an event-driven fashion (Figure 2, bottom). The lane clock speed is based on a 16nm design, slightly older Si process than the 12nm used for the Nvidia V100. Altogether, a node provides 2,048 parallel lanes, scaling up to 33 million lanes in a 16K node deployment. We show performance results from simulations of up to 1,024 nodes.

The UpDown architecture is designed for fine-grained parallelism. Threads can perform meaningful computation with as few as 10–100 instructions, thanks to lightweight execution mechanisms. The HBM stacks deliver high bandwidth, making the system particularly effective for applications with low data reuse. This combination of fine-grained parallelism, abundant compute lanes, and memory bandwidth enables UpDown to achieve high performance on graph computations.

Because the UpDown has no data caches, each lane has a small scratchpad memory of 64KB that can be accessed in a single cycle. Scratchpad memory can be pooled within an accelerator for a total of 4MB capacity shared by 64 lanes. These small SRAM memories are used to exploit data reuse and to intermediate global system memory access (DRAM) with small programs and implement flexible memory atomicity and consistency properties.

C. Programming tools

UpDown has a simple programming system that leverages global address space (global naming), and low-latency network (good access to distributed data) to achieve high performance on a wide range of graph applications [2].

Programmers can express fine-grained parallelism conveniently in a map-reduce framework called UpDown Key-Value Map-Shuffle-Reduce (KVMSR) [7]. For example, maps over

979-8-3315-7844-2/25/$31.00 ©2025 IEEE

a graph's vertices or edges express millions of parallel tasks. These tasks are significantly more potent than in cloud map-reduce frameworks because, in UpDown, they compute over shared, mutable data abstractions.

The detailed program structure for UpDown is expressed in a C-like system programming language which supports basic data types, arithmetic expressions, and basic control flow constructs. It also expresses small-scale parallelism by exposing machine primitives such as events, message sends, split-transaction memory access, and more.

Together, the UpDown software (UDKVMSR, UDWeave, and libraries) has been used to implement the full suite of IARPA AGILE applications [2] and more.

III. TC and K-Truss on UpDown

The TC and K-Truss algorithms exploit fine-grained vertex and edge parallelism directly. For both, a single copy of the undirected graph is stored in a CSR-like format in the global shared memory. The graph data structure has two parts: 1) vertices and their metadata, such as IDs, degrees, and the starting offsets, and 2) an edge array that stores one destination vertex ID for each edge. Each vertex's neighbor list is sorted in ascending order of IDs. Vertex IDs are assigned in order of decreasing degree, and only edges (v, u) satisfying $v > u$ are retained, ensuring each edge appears only once. This structure is known to reduce total work in TC algorithms [12]–[15].

A. Triangle Counting

Our Triangle Counting algorithm (Algorithm 1) parallelizes computation across both vertices (using map threads), and edges (using reduce threads) for neighbor list intersections. The implementation leverages UpDown's ample memory bandwidth to perform all neighbor-list reads directly from DRAM. Despite this, TC on UpDown is compute-bound. We employ a linear merge-based intersection method with a complexity of $\mathcal{O}(|N(v)| + |N(u)|)$.

Vertex computations are distributed evenly across UpDown lanes (Line 1). Each UpDown lane processes vertices in parallel using multiple threads executing the KVMSR map functions on Lane i (Line 2). To tolerate DRAM latency, each map thread accesses 64B chunks of the neighbor list in parallel (Line 4). The chunks for each (v, u) are sent to Lane k, where a reduce thread computes the intersection of $N(v)$ and $N(u)$ (Line 6). These lanes can handle multiple tasks simultaneously in an interleaved fashion, so we do not need to worry about assigning tasks to the same lane, and this also helps achieve high lane utilization. Triangle counts are accumulated in each lane, then combined at the end.

Algorithm 1 Triangle Count

Input: Graph $G(V, E)$, Lanes m
Output: Triangle Count tc
1: Lane i processes vertices v_i, v_{i+m}, v_{i+2m},...
2: **for all** vertex v in Lane i **in parallel do**
3: **if** $v.deg > 1$ **then**
4: **for all** vertex u in $N(v)$ **in parallel do**
5: $k \leftarrow \text{hash}(v, u) \bmod m$
6: $tc \mathrel{+}= \text{Intersect}(N(v), N(u))$ on Lane k

B. K-Truss

The K-Truss algorithm extracts a subgraph in which every edge is part of at least $k - 2$ triangles [16]. Algorithm 2 shows one iteration of K-Truss. The algorithm iterates, first computing the triangle count (TC), then copying qualifying edges and then vertices into a new graph. Note that V' stores a list of all vertices with positive out-degree in the oriented graph.

Algorithm 2 K-Truss Step (1 Iteration)

Input: Non-trivial vertex list V and neighbor lists $N(v)$
Output: Non-trivial vertex list V' and neighbor lists $N'(v)$
1: $\forall E \in G : tcCount[E] \leftarrow \text{TC}(G(V, N))$
2: **for all** $v \in V$ **in parallel do**
3: $deg \leftarrow 0$
4: **for all** edge $e \in N(v)$ **do**
5: **if** $tcCount[e] \geq k - 2$ **do**
6: add e to $N'(v)$
7: $deg \mathrel{+}= 1$
8: **if** $deg > 0$ **then** add v to V'

For k-truss, we modified TC to accumulate triangle counts for each edge (Line 1). For each triangle (v, u, z) found, counts for edges (v, z) and (u, z) are incremented immediately. The (v, u) count is incremented by the size of the intersection (the # of triangles found) after the full intersection. Since the $tcCount$ array is shared across the entire machine, we implement an atomic-add function, using the scratchpad to both cache values and ensure atomicity.

During edge deletion, edges with at least $k-2$ triangles are copied to a new graph. We also track the number of removed triangles, terminating if the quantity is zero or if the new graph is empty. This guarantees that K-Truss with $k = 3$ terminates in a single iteration with only a single TC computation. Other approaches require two iterations [17].

In many $k = k_{\max}$ scenarios, the graph shrinks significantly after the first iteration. To speed these iterations, we reduce the computational resources down to a single UpDown node once there are fewer than 2^{22} edges. The graph data structure is also compacted after each iteration. This approach speeds the additional 5–130 iterations needed to converge.

IV. Performance Results

A. Methodology

We simulate the UpDown system, using an accurate, scalable model (**Fastsim**) that provides cycle-accurate modeling of UpDown accelerators (instruction-level simulation) and simpler capacity and latency models for DRAM and system network. The **Fastsim** simulator was calibrated in 1-4 node simulations to a (**Gem5sim**) model that includes detailed models of the UpDown accelerators, TOP cores (x86 model),

979-8-3315-7844-2/25/$31.00 ©2025 IEEE

TABLE I
MERCURY (MRCY) AND UPDOWN RUN TIMES FOR TC (IN MS).

Dataset	#E	Runtime (ms)			Spdup
		MRCY	UD (1)	UD (8)	
SNAP Datasets					
amazon0312	2359k	0.98	0.20	0.04	26.96
amazon0505	2439k	0.99	0.21	0.04	26.54
amazon0601	2443k	1.01	0.21	0.04	25.80
cit-Patents	16.5m	6.02	1.54	0.22	27.61
friendster	1806m	1778	1342.38	169.50	10.49
flickrEdges	2317k	3.50	1.89	0.35	9.94
roadNet-CA	2767k	1.66	0.21	0.05	31.17
roadNet-PA	1542k	1.07	0.12	0.04	29.60
soc-livej	42.9m	24.19	10.96	1.46	16.60
com-orkut	117m	105.74	64.22	8.08	13.09
Synthetic Kronecker Datasets (Theory graphs)					
25-81-256-B1k	2132k	1.11	0.52	0.13	8.85
25-81-256-B2k	2132k	0.67	0.18	0.13	5.00
3-4-5-9-16-25-B1k	11.1m	13.64	10.90	1.56	8.73
3-4-5-9-16-25-B2k	11.1m	3.54	4.11	0.65	5.41
4-5-9-16-25-B1k	1583k	2.12	0.74	0.14	14.81
4-5-9-16-25-B2k	1583k	0.94	0.31	0.09	10.56
5-9-16-25-81-B1k	28.7m	35.77	29.47	3.70	9.66
5-9-16-25-81-B2k	28.7m	7.55	11.04	1.61	4.68
9-16-25-81-B1k	2606k	1.90	0.90	0.18	10.70
9-16-25-81-B2k	2606k	0.96	0.41	0.14	6.68
MAWI Datasets					
201512012345	19.0m	1.27	0.46	0.22	5.91
201512020000	37.2m	1.81	0.84	0.42	4.28
201512020030	71.7m	2.83	1.38	0.66	4.29
201512020130	135m	3.63	3.29	1.56	2.32
201512020330	240m	5.24	6.35	2.78	1.89
Graph500 Datasets (RMAT)					
scale18-ef16	3800k	4.06	4.30	0.64	6.39
scale19-ef16	7730k	7.94	9.85	1.46	5.43
scale20-ef16	15.7m	21.96	26.21	3.47	6.33
scale21-ef16	31.7m	49.53	62.29	8.37	5.92
scale22-ef16	64.1m	108.20	155.80	20.52	5.27
scale23-ef16	129m	256.20	388.28	49.98	5.13
scale24-ef16	260m	542.20	968.42	124.38	4.36
scale25-ef16	523m	1294	2407.36	305.73	4.23
rmat-scale28-ef16	4134m	–	57258.49	7169.61	–
Protein k-mer Graphs					
P1a (Graph 2)	149m	66.22	9.51	2.69	24.64
U1a (Graph 3)	69.4m	24.23	4.47	1.31	18.56
V1r (Graph 4)	233m	81.16	14.67	4.11	19.76
V2a (Graph 5)	58.6m	25.93	3.75	1.07	24.28

DRAM memories (Dramsim3), node and system-level interconnects. This simulation is fully cycle-accurate. We use the Fastsim model because its much greater speed makes 1,024 UpDown nodes (over 2 million lanes) and large application data sizes feasible.

For both TC and K-Truss, we utilize a representative set of datasets from the Graph Challenge repository [18], which spans a wide range of graph types and sizes. We compare our results to those from [9], [10].

B. Triangle Count (TC) Performance

We present triangle counting (TC) results in Table I, grouping datasets by type. We report runtime (in milliseconds) for three systems: MERCURY (a single NVIDIA A100 GPU), a 1-node UpDown, and an 8-node UpDown. We also report 8-node UpDown speedup versus MERCURY. All UpDown runs exploit only a single copy of the graph in shared global DRAM, enabling scalability to tera- and petabyte graphs.

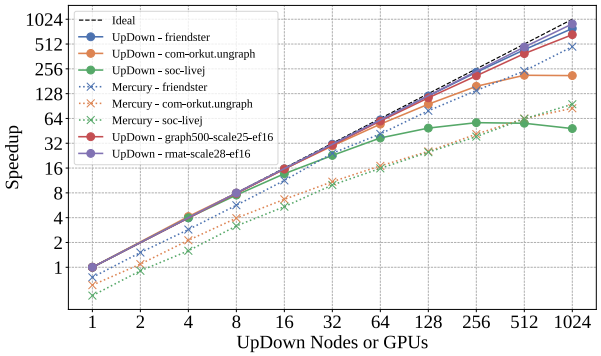

Fig. 3. Scalability of UpDown and Mercury

Overall, one UpDown node has a 1.97 geomean speedup over MERCURY. For the SNAP, Synthetic Kronecker, MAWI, and Protein k-mer datasets, the 1-node UpDown system outperforms MERCURY, achieving geomean speedups of $4\times$, $2\times$, $2\times$, and $6\times$, respectively. For the RMAT datasets MERCURY slightly outperforms the 1-node UpDown, with a geomean speed advantage of $1.4\times$. We attribute this advantage to the better performance of the hash-based intersection algorithm on these graphs. Good scaling is observed for the 8-node configuration across all large graphs.

In Figure 3, we examine TC's strong scaling behavior on UpDown, compared to the MERCURY system. Speedup is measured relative to a single UpDown node and plotted against the number of processing units. UpDown TC (circles, solid lines) achieves near-linear scalability on the Graph500 datasets up to 1024 nodes, closely following the ideal scaling line (dashed black). For 'rmat-scale28-ef16', UpDown reaches a speedup of approximately $900\times$ at 1024 nodes relative to the single-node baseline. This performance consistently exceeds MERCURY (x-marks, dotted lines) on large datasets.

For smaller graphs such as 'soc-livej' and 'com-orkut.ungraph' with a few million vertices, speedup falls off at 256 and 512 nodes, respectively, but still reflects $30-100\times$ speedup. This falloff in scalability is attributed to the limited graph size. We examine this effect in Figure 4.

Figure 4 shows TC performance for varied graph and UpDown system sizes. For small graphs, the 64-node UpDown configuration achieves the highest performance among all setups. In this regime, limited parallelism and the overhead of managing larger systems reduce the efficiency of configurations with more nodes. Each UpDown machine size achieves perfect scalability, provided sufficient graph size. Full performance is achieved with graphs having only 2^{29} edges, a remarkably strong scaling performance. Figure 4 also includes performance of the 96K-node BlueGene/Q system [15], a previous scalability champion. A single UpDown node out-

979-8-3315-7844-2/25/$31.00 ©2025 IEEE

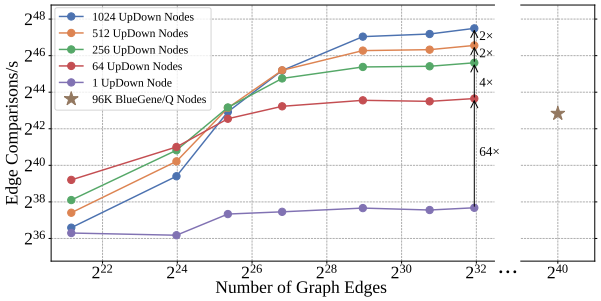

Fig. 4. TC performance versus graph size, various UpDown machine sizes, and a 96K-node BlueGene/Q [15]

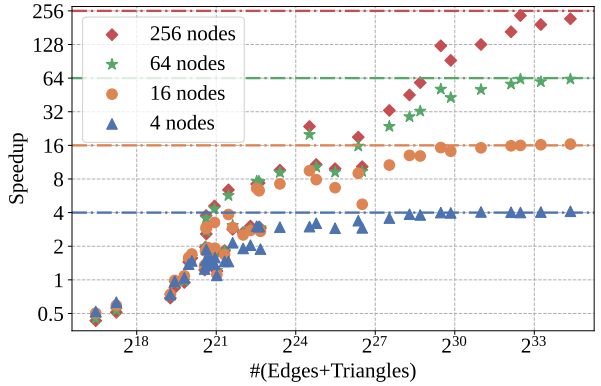

Fig. 5. K-Truss UpDown speedup vs a single node for $k = 3$ (refer to Table II). #(Edges+Triangles) represents the amount of work.

performs 1,500 BlueGene/Q nodes, and 64 nodes exceed the entire system.

C. K-Truss Performance

We compare UpDown performance to that in [10], using a single NVIDIA V100 GPU. We denote this system as "k-truss linear-algebra" (KTLA). Comparisons to 1, 4, and 16 UpDown nodes are presented in Table II, showing various runtime and speedup results. Speedups highlighted in **bold** indicate the highest speedups across UpDown configurations.

For $k = 3$, UpDown consistently outperforms KTLA across all machine sizes. A single UpDown node achieves a geometric mean speedup of $2.73\times$ over KTLA, with a maximum speedup of $11\times$. With 16 UpDown nodes, the geometric speedup increases to $6.69\times$, reaching up to $41\times$. For a few SNAP datasets, UpDown performance peaks on one node. However, it remarkably provides good speedups at 16 nodes for most graphs, including a few as small as 183K edges. Excellent scalability at 16-nodes with geomean speedups on SNAP ($7.12\times$), Kronecker ($5.91\times$), and RMAT ($12.71\times$) is achieved. The Kronecker datasets include two variants: high triangle counts (B1) and lower triangle counts (B2). UpDown outperforms KTLA on both, demonstrating robust performance. For Graph500 datasets (RMAT), UpDown achieves near-linear speedups (see also Figure 5).

For $k = k_{\max}$, a single UpDown node achieves a geometric mean speedup of $3.36\times$ over KTLA, with a maximum $7.82\times$ on 'ca-CondMat'. With 16 UpDown nodes, the geometric mean speedup is $5.21\times$, and a maximum of $18\times$ on 'cit-Patents'. A single UpDown node achieves $1.06\times$ to $7.82\times$ speedup over KTLA and a notable $4.33\times$ geomean speedup on SNAP datasets. The Kronecker datasets require few iterations (2–4) and thus have excellent scaling.

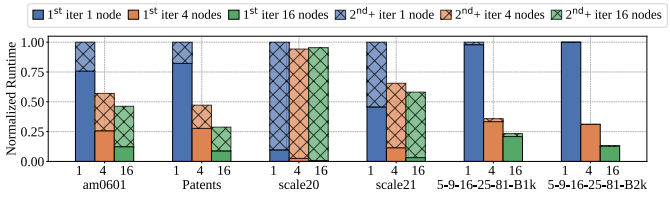

Fig. 6. Normalized runtime of $k = k_{\max}$ for 1, 4, and 16 UpDown nodes. The first iteration (solid) and subsequent iterations (hashed), relative to 1 node.

In Figure 5, we explore how UpDown K-Truss scaling depends on graph size and structure. Using #(Edges+Triangles) as a proxy for total work, our results show how much is needed to achieve maximum performance for each size machine. To provide enough parallelism to fill the UpDown machine, the plot indicates that 2^{23} (4-node), 2^{28} (16-node), 2^{30} (64-node), and 2^{33} (256-node) #(Edges+Triangles) are required. Note that the highest speedup achieved is 232-fold!

K-Truss for $k = k_{\max}$ is challenging to scale because of its iterative structure. In Figure 6, we show the normalized runtime for various graphs, separating the runtime for the first and subsequent iterations. The first iteration typically removes more than 95% of the edges (for all the graphs studied). As shown in Figure 6, for many graphs, the runtime of the second and following iterations (up to 134) dominates, limiting speedup. For the larger synthetic Kronecker graphs, only a few iterations (2-4) are required, so UpDown achieves good speedups (10-17x).

Overall, for K-Truss, UpDown scales, though sometimes below ideal. Scaling at all is good, as other implementations show little improvement with more hardware [19].

V. Discussion and Related Work

The performance results from UpDown illustrate the potential of the architecture to accelerate and scale graph computations from small graphs to massive graphs that require multiple compute nodes. The most significant finding is that UpDown's performance often outperforms GPUs and retains the ability to scale out to massive graphs.

Our results show a large performance increase over previous distributed Graph Challenge champions and honorable mentions targeting the scale-out case, such as [15], [20]. With respect to [15], we showed how 64 UpDown nodes should outperform the full Sequoia machine in Figure 4. The results from [20] show that 1092 Skylake cores can compute TC on a scale of 25 graphs in 0.725s. In comparison, a single UpDown takes 2.4s, and 8 UpDown nodes take 0.3s.

We were unable to find any scale-out results (more than 4 GPUs) for K-Truss to compare. The results from another Champion paper [17], which included a CPU benchmark and

979-8-3315-7844-2/25/$31.00 ©2025 IEEE

TABLE II
K-TRUSS PERFORMANCE, KTLA AND UPDOWN SYSTEMS. RUNTIMES IN MILLISECONDS, SPEEDUPS RELATIVE TO KTLA RUNTIME ON 1 V100.

Dataset	#E	$k_{\max}$	$k = 3$					$k = k_{\max}$				
			KTLA Rtime	Rtime (1 nd)	Spdup (1 nd)	Spdup (4 nd)	Spdup (16 nd)	KTLA Rtime	Rtime (1 nd)	Spdup (1 nd)	Spdup (4 nd)	Spdup (16 nd)
SNAP Datasets												
friendster	1,806,067,135	–	–	1543.17	1.00	4.01	**15.98**					
com-orkut	117,185,083	–	–	84.46	1.00	4.00	**15.28**					
soc-livej	42,851,237	–	–	19.93	1.00	3.87	**13.01**					
cit-Patents	16,518,947	36	9.07	2.11	4.31	12.91	**40.90**	12.66	2.47	5.13	10.86	**17.81**
roadNet-CA	2,766,607	4	0.61	0.29	2.10	3.06	**8.07**	1.07	0.30	3.57	4.44	**8.16**
roadNet-TX	1,921,660	4	0.53	0.21	2.55	3.52	**8.32**	0.80	0.22	3.65	4.32	**6.91**
roadNet-PA	1,541,898	4	0.48	0.17	2.83	3.84	**8.38**	0.72	0.19	3.93	4.52	**6.85**
amazon0505	2,439,437	11	1.04	0.40	2.60	7.77	**16.57**	1.10	0.52	2.12	3.61	**4.48**
amazon0601	2,443,408	11	1.04	0.40	2.62	7.89	**16.42**	1.83	0.51	3.60	6.30	**7.76**
amazon0312	2,349,869	11	1.06	0.39	2.72	8.21	**17.93**	1.59	0.51	3.16	5.48	**6.75**
email-EuAll	364,481	20	0.35	0.09	**3.85**	2.84	2.85	1.74	0.28	**6.34**	4.57	4.34
amazon0302	899,792	7	0.39	0.13	3.05	5.71	**8.79**	0.66	0.16	4.25	4.67	**5.21**
loc-gowalla_edges	950,327	29	0.60	0.27	2.23	4.79	**6.58**	1.44	0.42	3.46	4.54	**4.76**
soc-Slashdot0902	504,230	36	0.44	0.16	2.69	3.98	**4.59**	1.92	0.47	4.09	4.21	**4.21**
soc-Slashdot0811	469,180	35	0.43	0.15	2.86	3.95	**4.53**	2.48	0.54	**4.63**	4.62	4.62
soc-Epinions1	405,740	33	0.44	0.21	2.15	3.40	**4.13**	3.22	0.70	4.64	**4.84**	4.77
loc-brightkite_edges	214,078	43	0.32	0.07	**4.44**	4.32	4.38	1.05	0.23	**4.67**	4.02	3.87
email-Enron	183,831	22	0.32	0.08	4.07	4.24	**4.42**	1.93	0.37	**5.29**	4.76	4.48
cit-HepPh	420,877	25	0.40	0.13	3.11	4.74	**6.03**	0.94	0.22	4.38	4.69	**4.80**
cit-HepTh	352,285	30	0.38	0.14	2.78	4.17	**4.68**	1.25	0.30	4.24	**4.53**	4.53
as-caida20071105	53,381	16	0.26	0.02	**11.27**	5.87	5.67	0.77	0.10	**7.73**	5.68	4.80
ca-CondMat	93,439	26	0.26	0.03	**8.58**	5.4	5.06	0.39	0.05	**7.82**	4.30	3.68
ca-AstroPh	198,050	57	0.35	0.10	3.39	4.23	**4.57**	0.57	**0.15**	**3.83**	3.46	3.56
Geomean					3.32	4.86	7.12			4.33	4.77	5.36
Synthetic Kronecker Datasets (Theory graphs)												
Theory-5-9-16-25-81-B1k	28,667,380	84	68.00	33.22	2.05	5.97	**9.72**	79.49	33.610	2.36	6.61	**10.16**
Theory-5-9-16-25-81-B2k	28,667,380	4	19.86	10.91	1.82	5.85	**14.36**	23.61	10.915	2.16	6.91	**16.27**
Theory-3-4-5-9-16-25-B1k	11,080,030	62	17.04	12.89	1.32	3.84	**8.85**	20.84	13.22	1.58	4.28	**8.40**
Theory-3-4-5-9-16-25-B2k	11,080,030	7	4.92	4.26	1.15	3.42	**8.33**	6.11	4.285	1.42	4.08	**9.10**
Theory-9-16-25-81-B1k	2,606,125	28	2.52	1.33	1.89	3.57	**5.14**	3.192	1.41	2.26	3.84	**4.72**
Theory-9-16-25-81-B2k	2,606,125	5	1.01	0.46	2.18	3.19	**3.67**	1.229	0.47	2.61	3.55	**3.89**
Theory-25-81-256-B1k	2,132,284	28	1.74	0.76	2.28	4.35	**5.76**	1.833	0.80	2.30	3.81	**4.40**
Theory-25-81-256-B2k	2,132,284	4	0.65	0.26	2.49	2.73	**2.77**	0.760	0.27	2.81	**2.91**	2.80
Theory-4-5-9-16-25-B1k	1,582,861	28	1.67	1.06	1.57	3.22	**4.36**	1.993	1.17	1.70	2.97	**3.23**
Theory-4-5-9-16-25-B2k	1,582,861	6	0.74	0.35	2.11	3.16	**3.83**	0.886	0.37	2.43	3.27	**3.60**
Geomean					1.84	3.80	5.91			2.12	4.05	5.65
Graph500 Datasets (RMAT)												
graph500-scale25-ef16	260,261,843	–	–	3136.19	1.00	4.11	**16.42**					
graph500-scale24-ef16	64,097,004	–	–	1287.82	1.00	4.05	**16.20**					
graph500-scale23-ef16	260,261,843	–	–	528.45	1.00	4.03	**15.84**	54.10	1385.12	7.26	9.85	**10.69**
graph500-scale22-ef16	64,097,004	485	–	216.53	1.00	4.05	**15.21**	91.50	466.65	7.41	10.93	**12.39**
graph500-scale21-ef16	31,731,650	373	–	88.86	1.00	3.97	**14.17**	588.83	194.21	5.74	8.75	**9.87**
graph500-scale20-ef16	15,680,861	284	–	35.82	1.00	3.81	**12.85**	1114.24	370.06	1.59	**1.69**	1.67
graph500-scale19-ef16	7,729,675	213	–	14.60	1.00	3.58	**10.66**	3458.24	54.93	1.67	**2.05**	2.04
graph500-scale18-ef16	3,800,348	159	–	6.24	1.00	3.38	**9.01**	10061.22	50.98	1.06	**1.15**	1.12
Geomean					1.00	3.80	12.71			3.09	3.94	4.13

a fine-grained parallel computation, were outperformed by the latter paper [10], which is included in our comparison. Earlier work on distributed K-Truss computations [21] focused on a few massive K-Truss runs with an algorithm that updated the graph structure and triangle counts as edges were removed, making it hard to compare against our results. UpDown's simple algorithm highlights its potential performance and productivity.

In terms of power usage compared to the recent Mercury results [9], each UpDown node is estimated at 0.5KW of power (including networking infrastructure). In contrast, each Mercury node with 4 GPUs is 1.5KW, with each GPU consuming 0.3KW. Since we see a 1.97 geomean speedup, this shows UpDown achieving an isopower speedup vs a single A100.

Perhaps the most significant point of discussion is that these results on UpDown reflect the performance of a simulator rather than an actual machine. Our execution simulators are cycle-accurate at the instruction level and use simple approximations for other system components (DRAM and network access). They have been designed to be conservative concerning the possible performance should a machine be built. For instance, detailed RTL simulations conducted as part of AGILE suggest that faster clock speeds and lower power should be an option. Also, UpDown is currently based on estimates from much older Si processes (16nm). Multiple networks are being designed that will likely exceed the performance of the network utilized for UpDown simulations. Finally, we note that the algorithms implemented for UpDown are relatively simple and have only minimal architecture-specific tuning. Our linear search intersection-based TC code could likely be improved using a carefully designed hash-based lookup. Consequently, we believe these performance results may be conservative. And crucially, they show what performance should be possible for these graph algorithms and how the hardware features of UpDown enable simple, scalable computations.

979-8-3315-7844-2/25/$31.00 ©2025 IEEE

REFERENCES

[1] "Advanced graphic intelligence logic computing environment," 2022, https://www.iarpa.gov/research-programs/agile.

[2] "AGILE Program Workflows and Benchmarks," 2022, https://www.iarpa.gov/images/PropsersDayPDFs/AGILE/AGILE_Program_Workflows_FINAL.pdf.

[3] Andrew A Chien, et. al, "UpDown: A Supercomputer Co-designed for Scalable Graph Processing," *under review*, 2024, available from http://people.cs.uchicago.edu/~aachien/lssg/research/10x10/UpDown_System_Paper___TPDS_submittedv2.pdf.

[4] A. Rajasukumar, J. Su, Yuqing, Wang, T. Su, M. Nourian, J. M. M. Diaz, T. Zhang, J. Ding, W. Wang, Z. Zhang, M. Jeje, H. Hoffmann, Y. Li, and A. A. Chien, "UpDown: Programmable fine-grained Events for Scalable Performance on Irregular Applications," 2024. [Online]. Available: https://arxiv.org/abs/2407.20773

[5] A. Rajasukumar, T. Zhang, R. Xu, and A. A. Chien, "UpDown: A Novel Architecture for Unlimited Memory Parallelism," in *Proceedings of the International Symposium on Memory Systems*, ser. MEMSYS '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 61–77. [Online]. Available: https://doi.org/10.1145/3695794.3695801

[6] Y. Wang, S. Perarnau, and A. A. Chien, "UpDown: Combining Scalable Address Translation with Locality Control," in *SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2024, pp. 1014–1024.

[7] Y. Wang, A. Rajasukumar, T. Su, M. Nourian, A. P. Jose M Monsalve Diaz, J. Ding, C. Colley, W. Wang, Y. Li, D. F. Gleich, H. Hoffmann, and A. A. Chien, "Efficiently exploiting irregular parallelism using keys at scale," in *Proceedings of Conference Workshop on Languages and Compilers for Parallel Computing*, Nov 2023.

[8] Y. Wang, C. Colley, B. Wheatman, J. Su, D. F. Gleich, and A. A. Chien, "How Fast Can Graph Computations Go on Fine-grained Parallel Architectures," 2025. [Online]. Available: https://arxiv.org/abs/2507.00949

[9] Z. Lin, C. Xu, K. Meng, and G. Tan, "Mercury: Efficient subgraph matching on gpus with hybrid scheduling," in *2024 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, Sep. 2024, p. 1–7. [Online]. Available: http://dx.doi.org/10.1109/HPEC62836.2024.10938527

[10] R. Wang, L. Yu, Q. Wang, J. Xin, and L. Zheng, "Productive High-Performance k-Truss Decomposition on GPU Using Linear Algebra," in *2021 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, Sep. 2021, p. 1–7. [Online]. Available: http://dx.doi.org/10.1109/HPEC49654.2021.9622792

[11] K. Lakhotia, L. Monroe, K. Isham, M. Besta, N. Blach, T. Hoefler, and F. Petrini, "Polarstar: Expanding the scalability horizon of diameter-3 networks," *arXiv preprint arXiv:2302.07217*, 2023.

[12] N. Chiba and T. Nishizeki, "Arboricity and subgraph listing algorithms," *SIAM Journal on Computing*, vol. 14, no. 1, p. 210–223, Feb. 1985. [Online]. Available: http://dx.doi.org/10.1137/0214017

[13] T. Schank and D. Wagner, *Finding, Counting and Listing All Triangles in Large Graphs, an Experimental Study*. Springer Berlin Heidelberg, 2005, p. 606–609. [Online]. Available: http://dx.doi.org/10.1007/11427186_54

[14] J. Cohen, "Graph twiddling in a mapreduce world," *Computing in Science & Engineering*, vol. 11, no. 4, p. 29–41, Jul. 2009. [Online]. Available: http://dx.doi.org/10.1109/MCSE.2009.120

[15] R. Pearce, T. Steil, B. W. Priest, and G. Sanders, "One quadrillion triangles queried on one million processors," in *2019 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, Sep. 2019, p. 1–5. [Online]. Available: http://dx.doi.org/10.1109/HPEC.2019.8916243

[16] J. Cohen, "Cohensive subgraphs for social network analysis," Unpublished tech report, available from https://github.com/rhpran/Networkx-V3-Ref-Papers/blob/main/Trusess_%20Cohesive%20subgraphs%20for%20social%20network%20analysis.pdf.

[17] M. P. Blanco, T. M. Low, and K. Kim, "Exploration of fine-grained parallelism for load balancing eager k-truss on gpu and cpu," in *2019 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, Sep. 2019, p. 1–7. [Online]. Available: http://dx.doi.org/10.1109/HPEC.2019.8916473

[18] "MIT GraphChallenge," https://graphchallenge.mit.edu/, 2025, accessed: 2025-07-08.

[19] V. S. Mailthody, K. Date, Z. Qureshi, C. Pearson, R. Nagi, J. Xiong, and W.-m. Hwu, "Collaborative (cpu+ gpu) algorithms for triangle counting and truss decomposition," in *2018 IEEE High Performance extreme Computing Conference (HPEC)*. IEEE, 2018, pp. 1–7.

[20] S. Acer, A. Yasar, S. Rajamanickam, M. Wolf, and U. V. Catalyurek, "Scalable triangle counting on distributed-memory systems," in *2019 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, Sep. 2019, p. 1–5. [Online]. Available: http://dx.doi.org/10.1109/HPEC.2019.8916302

[21] R. Pearce and G. Sanders, "K-truss decomposition for scale-free graphs at scale in distributed memory," in *2018 IEEE High Performance extreme Computing Conference (HPEC)*. IEEE, Sep. 2018, p. 1–6. [Online]. Available: http://dx.doi.org/10.1109/HPEC.2018.8547572

979-8-3315-7844-2/25/$31.00 ©2025 IEEE