# STING: Near-storage accelerator framework for scalable triangle counting and beyond

Seongyoung Kang
seongyk3@uci.edu
University of California Irvine
Irvine, California, USA

Sang-Woo Jun
swjun@ics.uci.edu
University of California Irvine
Irvine, California, USA

## ABSTRACT

One of the most critical limitations to scalable graph mining is memory capacity, as graphs of interest continue to grow while the rate of DRAM scaling diminishes. While high-performance NVMe storage is cheap and dense enough to better support larger graphs, the relative performance limitations of secondary storage force a cost-performance trade-off. We present STING, which uses an asynchronous callback function to provide a general interface to in-storage graphs while allowing transparent near-storage acceleration. Using triangle counting, we show with transparent filtering and sorting acceleration, STING can improve state-of-the-art by 3x for cost and power efficiency.

## CCS CONCEPTS

• **Computer systems organization** → **Reconfigurable computing**; • **Information systems** → *Storage architectures*; • **Mathematics of computing** → Graph algorithms.

## KEYWORDS

FPGA, Near-storage acceleration, Triangle counting, Graph mining

**ACM Reference Format:**
Seongyoung Kang and Sang-Woo Jun. 2024. STING: Near-storage accelerator framework for scalable triangle counting and beyond. In *61st ACM/IEEE Design Automation Conference (DAC '24), June 23–27, 2024, San Francisco, CA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3649329.3658265

## 1 INTRODUCTION

A very wide range of real-world information is naturally expressed as graphs, and graph mining algorithms such as subgraph isomorphism, clustering, and triangle counting are key components of many important applications including computer system security [15] and machine learning [22]. Triangle counting is a representative example of such algorithms in terms of complexity and behavior, and is also a well-known fundamental component of many complex graph mining algorithms [1]. In fact, triangle counting and k-truss identification together form one of the three challenges of the IEEE/HPEC Graph Challenge benchmark [18].

One prominent limitation of such complex graph mining algorithms in the real world is their scalability, especially regarding memory capacity. Real-world graphs of interest are scaling at a much faster pace than the slowly diminishing scalability of DRAM [12]. This issue is exacerbated by the often high memory capacity requirements of efficient graph mining algorithms, which are often dramatically different from simpler algorithms such as PageRank or Breadth-First-Search [9]. For example, we show that triangle counting using performance-optimized GraphBLAS on relatively small graphs, with only around one billion edges, can require hundreds of GB of fast random-access memory capacity.

Many existing efforts attempt to address the scalability issue of such complex graph mining algorithms, including the use of higher-density secondary storage devices with corresponding external-memory algorithms [10, 11, 13], as well as approximate algorithms for reduced resource requirements [8, 21]. Unfortunately, the lower-cost external memory systems typically achieve much lower performance compared to in-memory systems due to the orders of magnitude performance gap between the memory technologies as well as I/O amplification due to coarse-grained storage pages. This introduces a trade-off between performance and cost when considering an external-memory system. Similarly, approximate algorithms also introduce a trade-off between accuracy and cost.

In this work, we present the design and evaluation of the STING framework, which uses near-storage acceleration coupled with a general-purpose graph interface designed for incorporating transparent optimizations. STING exposes an asynchronous, out-of-order interface for generic graph access functionalities such as neighborhood queries. This approach has the dual benefit of supporting a wide range of algorithms via a flexible interface, but at the same time hiding the added latencies of application-specific and storage-specific optimizations and accelerators without exposing them in the interface. For example, the near-storage hardware accelerator sorts the neighborhood and edge queries to group together requests with spatial locality, and hardware-accelerated bloom filters are used to filter queries to nonexistent edges.

We evaluate our approach using an exact triangle counting algorithm applied to a wide range of graphs spanning millions to billions of edges, with the number of discovered triangles spanning dozens to billions. We implement STING using the Samsung SmartSSD computational storage device, which incorporates a Field-Programmable Gate Array (FPGA) for implementing near-storage acceleration. We compare against a wide range of existing, optimized systems for graph analytics and triangle counting, including GraphBLAS [3] and previous Graph Challenge champions [20], some of which use GPU acceleration [17]. Our evaluations showed that across all configurations, STING achieves significantly higher


*DAC '24, June 23–27, 2024, San Francisco, CA, USA*

ACM ISBN 979-8-4007-0601-1/24/06
https://doi.org/10.1145/3649329.3658265

scalability by requiring less than 1/3 the hardware resources to achieve competitive performance against the best of the comparison systems. The transparent bloom filters were able to filter storage accesses by up to and over 90%, and sorting the filtered requests resulted in almost no I/O amplification during storage access.

STING provides a scalable and flexible path to using secondary storage and near-storage hardware acceleration for complex graph mining algorithms. As graphs of interest continue to grow, STING shows that denser, coarser memory technologies such as NAND flash and other NVM fabric can continue performance scaling even while memory capacity scaling struggles to keep up.

## 2 BACKGROUND AND RELATED WORK

Many complex graph mining algorithms including triangle counting are inherently computation and memory intensive, posing a serious scalability challenge. To address these challenges, a variety of approaches have been devised, each focusing on minimizing computational demands via approximate computing [21], optimizing memory usage [17, 20], enhancing parallel processing efficiency [7, 19], etc. This section offers an overview of two fundamental strategies using triangle counting as the driving example.

***Intersection-Based Approach:*** The intersection-based approach is a pivotal technique in graph analytics. This approach identifies common vertices shared between two distinct vertices. As detailed in Algorithm 1, it involves examining the intersections of adjacency lists of connected nodes.

Many intersection-based methods for triangle counting utilize hash tables [7, 16, 17, 20] or filters [6, 8] to improve process efficiency. These techniques expedite edge existence verification and enhance computational efficiency. Hmap [20], a notable example, optimizes computation through strategic hash table construction. Specifically, in Algorithm 1, Hmap creates hash tables for each vertex $v$ with a lower degree than vertex $u$. This approach not only facilitates rapid edge verification but also improves memory usage.

Using GPUs, or distributed systems through MPI is a common strategy in intersection-based methods [5, 7, 16, 17]. Although these methods benefit from parallelism, they typically demand preliminary steps to guarantee efficient parallel scheduling and optimal resource utilization [5, 17].

***Matrix Multiplication-Based Approach:*** The matrix multiplication-based approach in graph analytics utilizes algebraic operations to address complex problems, including triangle counting, based on the fact that multiplying the adjacency matrix is equivalent to neighborhood queries.

**Algorithm 1** Intersection-Based Approach

1: **Input:** $G(V, E)$
2: **Output:** Triangle count in $G$
3: $t \leftarrow 0$ {Initialize triangle count}
4: **for all** $u \in V$ **do**
5: **for all** $v \in \text{Adj}(u)$ **do**
6: $t \leftarrow t + |\text{Adj}(u) \cap \text{Adj}(v)|$
7: **end for**
8: **end for**
9: **return** $t$

The conventional method to count triangles involves calculating the cube of the adjacency matrix, which identifies triadic closures. However, for undirected graphs, as delineated in Algorithm 2, performing and simply performing an AND operation with the original matrix can significantly reduce the memory footprint and computational load. GraphBLAS supports multiplication and masking functions to enable such efficient triangle counting [2].

**Algorithm 2** Triangle Counting via Matrix Multiplication

1: **Input:** Lower Triangular Matrix $L$ of Graph $G$
2: **Input:** Upper Triangular Matrix $U$ of Graph $G$
3: **Output:** Triangle count in $G$
4: $H \leftarrow L \cdot U$ {Matrix multiplication of $L$ and $U$}
5: $H \leftarrow G \wedge H$ {Apply element-wise AND with $G$}
6: $t \leftarrow \text{reduction}(H)$ {Sum over $H$ to get the triangle count}
7: **return** $t$

## 3 STING ARCHITECTURE

STING aims to provide a flexible framework to implement complex graph mining algorithms in an accelerated external memory environment. Figure 1 describes our target hardware environment for such a system. One or more NVMe storage devices connect to the host machine over an interconnect fabric such as PCIe, and each storage device is augmented with a reconfigurable hardware accelerator fabric such as an FPGA. The graph in question is stored in dense, high-capacity storage instead of the costly host memory.

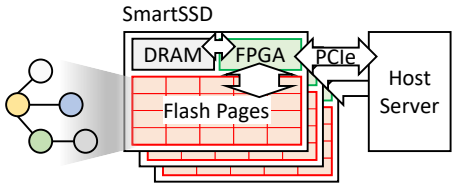


**Figure 1: Accelerated external-memory graph analytics.**

There are a few restrictions we must overcome to achieve high performance graph mining on such a system. Most prominently, the bandwidth of NVMe storage is over an order of magnitude slower than host DRAM (sub-10 GB/s vs. 100s of GB/s). Also, the storage fabric is organized into multi-KB *pages*, meaning fine-grained random accesses incur very I/O amplification. STING aims to address these concerns, without harming the generality of the interface.

### 3.1 Graph Storage Interface

We developed a generic interface similar to existing, proven work on Graph-Semantic storage devices developed for graph mining (e.g, [14]), with the following commands:

- `STING.get_vertex( vertexID, callback )`
- `STING.get_edge( vertex_A, vertex_B, callback )`
- `STING.get_neighbors( vertexID, callback )`

The functionality of these functions is straightforward, but one special design consideration is that instead of the function blocking until the return values are available, we opt for an *asynchronous*

**Algorithm 3** Triangle counting in STING

```
1:  Input: G(V, E)
2:  Output: Triangle count in G
3:  t ← 0                                   {Initialize triangle count}
4:  for all u ∈ V do
5:    for all e ∈ Pairwise(Adj(u)) do
6:      if STING.get_edge(e) then
7:        t ← t + 1
8:      end if
9:    end for
10: end for
11: return t
```

interface where each call provides a callback function which will be invoked with the query results. This simple addition allows us to transparently introduce various optimizations to address most of the issues mentioned above regarding external memory analytics.

Figure 2 illustrates the timeline of an example sequence of queries on STING. Because the asynchronous interface relaxes the latency-bound aspect of a conventional blocking interface, STING is able to insert pre-processing and other operations, potentially near-storage accelerated into the query process. Figure 2 shows two such optimizations: First, windows of requests are reordered to group requests to the same storage page into bursts, improving caching effectiveness and reducing I/O amplification. Second, some requests can be filtered out by using near-storage caches or other filters, reducing the storage access overhead and improving effective bandwidth. We implement both functions and observe performance improvements in graph mining.

## 3.2 Driving Example: Triangle Counting

As mentioned earlier, we target triangle counting as the driving example for evaluation of STING, since it is not only an important application by itself, it is the key component of many other graph mining algorithms. Using the STING interface, triangle counting can be expressed with Algorithm 3. For each vertex, the algorithm creates pairwise permutations of all of its neighbors (`Pairwise`), and checks if an edge exists between vertices of each pair using `get_edge`. While Algorithm 3 shows `get_edge` being called in a blocking fashion for simplicity, in reality incrementing $t$ should happen in an asynchronous callback function.

## 3.3 Accelerator Architecture

The goal of the near-storage accelerator is threefold: Safely filter out as many requests as possible, group bursts of spatially local requests together, and do these without causing changes to the graph store interface. The architecture illustrated in Figure 3 aims to achieve both goals using two accelerators: Bloom filter and Radix sorter. Both accelerators are introduced transparently by the framework without causing the graph store interface to change. We also show in the detailed architecture sections below that these two accelerators actually share a lot of resources.

Figure 3 also describes the process of filtering a stream of requests, via labeled arrows representing the flow of data. STING currently stores the graph in Compressed Sparse Row (CSR) format.

**Step 0: Bloom filter programming:** Step *zero* programs the bloom filter using graph data. This is done by the STING framework during initialization, without being explicitly invoked by the user program. It needs to be done only once per graph before any query can be issued. The bloom filter can be programmed with different information depending on the goal of the graph mining algorithm. For example, both triangle counting, as well as the pruning process for the Ullman subgraph isomorphism algorithm, can benefit from filtering `get_edge` requests, so the bloom filter is programmed with the list of edges via source and destination index pairs.

**Step 1 to 5: Filtering edge queries:** Once the bloom filter is programmed, the STING accelerators are ready to process queries. Edge queries come from the host over PCIe, (Step 1) and simply filtered by the bloom filter. This will probabilistically reduce the number of requests, and many false positives may still remain in the stream. Then, the filtered requests are pushed through a hardware radix sorter, to group together requests into buckets with spatial locality in the CSR graph (Step 2). The size and number of buckets depend on the available size of host memory. The partitioned streams are accumulated into buckets in the SSD (Step 3). Once all input queries are filtered and sorted, each bucket can be read out sequentially, to ensure high-locality access into the CSR graph.

**Benefits of near-storage acceleration:** The benefit of placing computation near-storage is mainly to reduce traffic over PCIe. In Figure 3, traffic reduction happens in two places.

First, in Step 0, the CSR graph can stream into the bloom filter without bothering the host-side PCIe bandwidth. If the PCIe bandwidth is not slower than the SSD bandwidth, this step by itself will not remove any bandwidth bottleneck. On the other hand, traffic reduction can have significant performance benefit if the internal bandwidth of the SSD is higher than the PCIe link (which is common for off-the-shelf SSDs [4]), or if enough SSDs are plugged into the system to overwhelm the host's PCIe root complex.

Second, in Step 4, the radix sorted stream is written to the local SSD without bothering the host-side PCIe link. Similar to Step 0, this

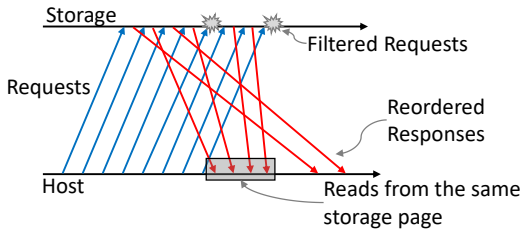


**Figure 2: Asynchronous interface can hide optimizations.**

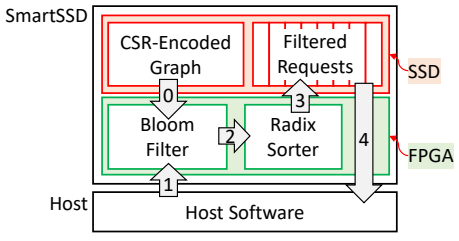


**Figure 3: STING configured for triangle counting.**

can relax the pressure on the host's PCIe root complex. In our actual implementation, Step 4 involves multiple phases of storage I/O involving temporary data, as we show in the detailed architecture sections below. Near-storage acceleration can avoid traffic over the potentially congested host PCIe root complex.

## 3.4 Radix Sorter Architecture

STING implements a two-layer bursting radix sorter, in order to support a large number of buckets within the limited chip space. The critical performance limitation of building one large radix sorter is the random access performance of off-chip DRAM, which suffers from orders of magnitude performance loss for fine-grained random accesses. For example, the DDR4 memory on the Samsung SmartSSD reports 16+ GB/s bandwidth on 8 KB bursts, but much less than 1 GB/s on 8 Byte bursts. This means sorted results must be organized into long bursts within on-chip memory, but the chip cannot support tens of thousands of multi-KB on-chip buffers.

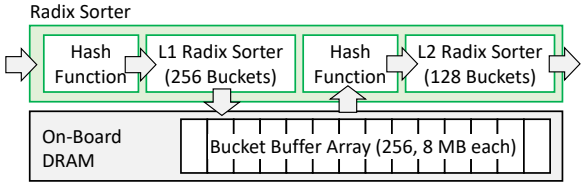


**Figure 4: A two-layer radix sorter for many buckets.**

Our implementation implements two separate layers of radix sorters, as illustrated in Figure 4. The L1 sorter maintains 256 on-chip burst buffers, each 1 KB in size. Given an element to sort, the L1 sorter looks at the most significant 8 bits, and accumulates them to an off-chip double-buffered DRAM bucket. Each DRAM bucket needs to be as large as possible, as each bucket is further sorted by the L2 radix sorter. The L2 sorter maintains 128 buckets (due to BRAM capacity limitations) and looks at the next 7 bits to further sort each L1 bucket. The L2 sorter must be flushed after each input bucket, to avoid output bursts with data mixed across L1 buckets. This way, our radix sorter can support $2^{15}$ buckets within the on-chip limitation.

## 3.5 Bloom Filter Architecture

The bloom filter of STING uses the two-layer radix sorter to achieve high performance memory accesses, as depicted in Figure 8. The job of the bloom filter is to reduce the number of edge requests, given as source and destination vertex ID pairs. To support this type, the input to the bloom filter is a pair of 32-bit vertex IDs, which is hashed together into a 32-bit hash value. The radix sorter is configured to sort the resulting 3-tuple of 32 bit values.

The output bursts from the radix sorter is used to fetch the corresponding chunks of the bloom filter table from off-chip DRAM. As illustrated in Figure 6, given a 32-bit hash, the two-layer sorter is able to sort the upper 15 bits, resulting in the lower 17 bits of each burst being unsorted. We further sort one additional bit by using two separate filter engines, and have each burst be further divided according to another address bit. The remaining 16 bits

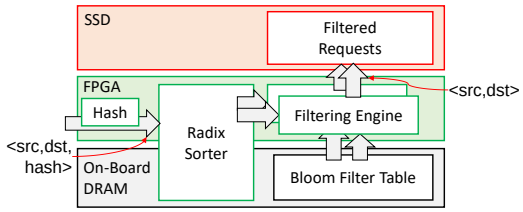


**Figure 5: The bloom filter uses the two-layer radix sorter accelerator to probabilistically filter requests.**

represent bitwise random accesses within an 8 KiB region, which can be effectively cached on-chip.

The filtering engines can filter out some of the sorted tuples based on the bloom filter table. After filtering, the hash is removed from the tuple and a stream of vertex ID pairs are written to storage. We note that the filtered request list has not been sorted into in-storage buckets as described in Section 3.3, because the radix sorter is busy. The filtered list must be read back and sorted again, to separate them into spatially local buckets.

# 4 EVALUATION

## 4.1 Evaluation Environment

We evaluated STING using a computer system augmented with an off-the-shelf Samsung SmartSSD, equipped with a Kintex Ultrascale FPGA and 4 GiB of DDR4 memory on top of 3.84 TB of NVMe storage over a PCIe Gen3x4 link. The host server included a Xeon Gold 6136 processor with 12 cores and 24 threads, 192 GiB of DDR4 memory, as well as an NVIDIA V100 GPU for GPU-enabled systems.

The host software portion of STING uses very little resources, typically only 4 threads and 4 GiB of DRAM, meaning all of the performance results below can be **achieved with a desktop-class machine**. Unfortunately, the SmartSSD hardware itself required a server-class BIOS due to its PCIe configuration, preventing us from evaluating this configuration. Table 1 shows the chip resource utilization of the SmartSSD's FPGA, synthesized at 200 MHz. The major limiting factor was the BRAM capacity, which prevented us from constructing two 256-bucket radix sorters.

**Table 1: Chip resource utilization of STING on SmartSSD.**

| | LUTs | RAMB36 | RAMB18 |
|---|---|---|---|
| Total | 205,779 (39.4%) | 677 (68.8%) | 29 (1.5%) |

For comparison systems, we chose a range of applications as mentioned in Section 2, including MonetDB, Neo4J, Hmap [20], GraphBLAS [3], and TRUST [17]. TRUST was the only open-source

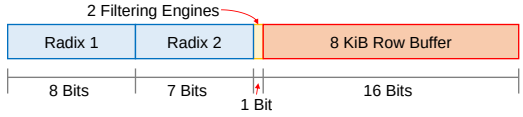


**Figure 6: The two-layer radix sorter can support a 32 bit of address space bloom filter.**

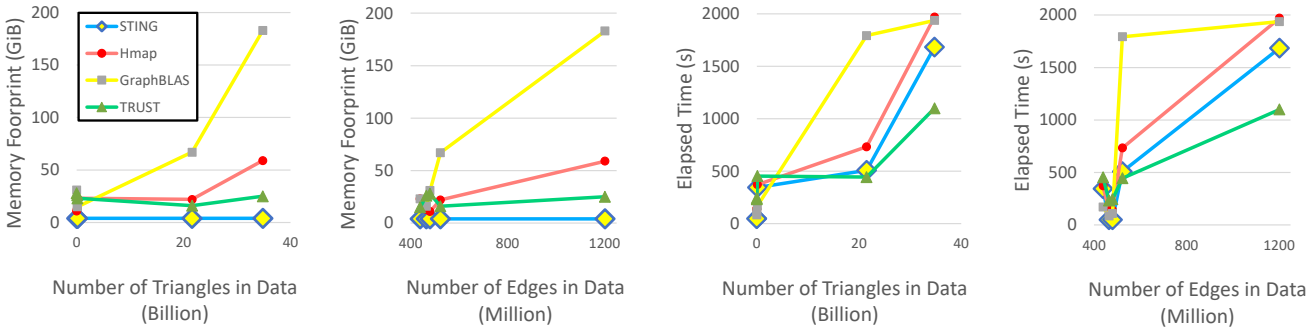


**Figure 7: STING demonstrates superior memory efficiency (left), as well as competitive performance (right, lower is better).**

GPU-accelerated triangle counting system we could find, and all others were CPU-centric systems. Unfortunately, we could not measure MonetDB and Neo4J performance, as they required excessive computational and memory resources leading to timeouts and memory overflows.

## 4.2 Dataset

Table 2 describes the graph datasets we used for evaluation. The datasets fall into two categories based on their characteristics. The first category includes graphs with a vast number of triangles, such as the Twitter social network graph, the synthesized Graph500 dataset, and the DARPA TC dataset constructed from system audit logs. The second category comprises of datasets with fewer number of triangles, including the MAWI network traffic graphs, and the V1r K-mer dataset. All datasets represent the largest publicly available data in their respective domains, offering comprehensive insight into evaluating the capability of each platform to manage large-scale graph data efficiently.

## 4.3 End-to-End Performance Evaluation

The right two graphs of Figure 7 shows the end-to-end performance comparison among evaluated platforms, with varying graph sizes in terms of Billions of triangles or millions of edges. STING demonstrates competitive performance, outperforming all but the GPU-accelerated TRUST on all graphs.

The major reason STING demonstrates such high performance is the high filtering rate of the hardware-accelerated bloom filter. To evaluate the scalability of this approach, we present Figure 8, which measures the filtering rate of the bloom filter with different filter table sizes, as well as the resulting average performance in Million Traversed Edges Per Second (MTEPS). We see that size benefits start to diminish after 512 MiB, which is a very small size compared to the available 4 GiB on-board memory, meaning on-board memory capacity is not the primary performance bottleneck. Our measured performance scaling ends at 512 MiB because the radix sorter cannot be made bigger due to on-chip BRAM limitations, but performance is projected to grow beyond this, if we can use either more radix sorter layers or other on-chip memory technologies such as URAM.

**Table 2: Evaluated graph datasets.**

| Name | $\|V\|$ | $\|E\|$ | Triangles |
|---|---|---|---|
| MAWI | 226,196,185 | 480,047,894 | 26 |
| V1r | 214,005,017 | 465,410,904 | 49 |
| DARPA | 173,764,372 | 439,252,151 | 136,233,928 |
| G500-scale25 | 17,043,780 | 523,467,448 | 21,575,375,802 |
| Twitter | 41,652,230 | 1,202,513,046 | 34,824,916,864 |

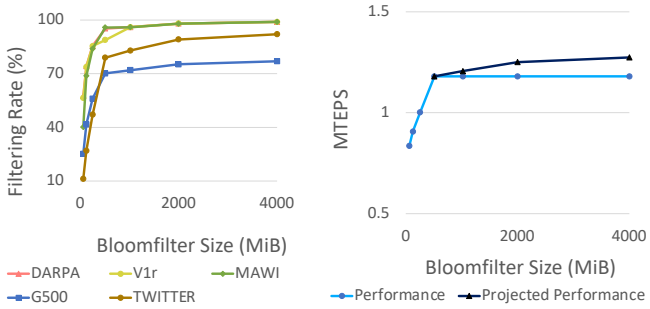


**Figure 8: Impact of host Bloom filter Size on Performance.**

## 4.4 Scalability and Memory Efficiency

The left two graphs of Figure 7 illustrates the memory footprint of each evaluated platform, with varying graph sizes in terms of billions of triangles or millions of edges. Thanks to the external memory configuration, STING demonstrates superior memory usage efficiency.

The only significant use of host memory capacity for STING is during actually evaluating the filtered and sorted queries against

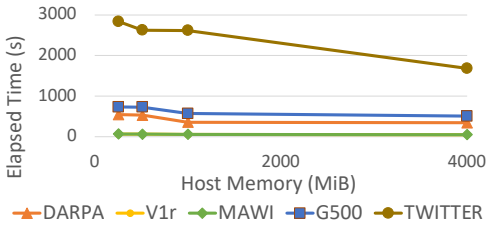


**Figure 9: Impact of Host DRAM Size on Performance.**

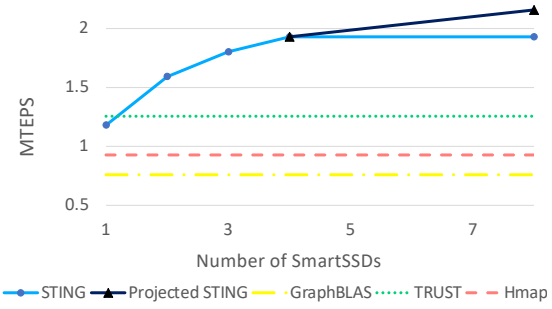


**Figure 10: Performance scaling with more SmartSSDs.**

the actual graph data structure, namely Step 4 in Figure 3. The available host memory capacity determines the number of buckets the filtered stream should be sorted to. To evaluate the scalability of this approach, Figure 9 shows the performance with different host memory usage (lower is better). The graph shows that STING performance is not very sensitive to host memory capacity, ensuring good scalability with even larger graphs.

Another interesting scalability factor is using more SmartSSDs, since the CPU and memory requirements per SmartSSD are quite low. Figure 10 shows the performance scaling of STING with more SmartSSDs in terms of MTEPS. We observe a semi-linear performance benefit from using multiple SmartSSDs, up to four, beyond which the host-side PCIe becomes the bottleneck.

## 4.5 Power Efficiency

Figure 11 compares the power efficiency of the evaluated systems. For STING, SmartSSD adds roughly 25 W of additional power consumption during load. But thanks to the low CPU and memory resource requirements of STING despite its competitive performance, STING delivers superior power efficiency, around 3× compared to the other evaluated systems.

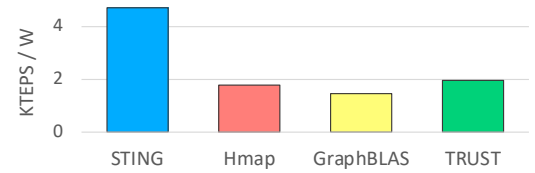


**Figure 11: STING achieves superior power efficiency.**

# 5 CONCLUSION

We present STING, which includes a flexible graph store interface capable of transparent introduction of near-storage acceleration for storage access optimizations. Thanks to the high performance and efficiency of the transparently inserted bloom filter and radix sorters, STING was able to demonstrate competitive performance at around 1/3 the cost and power efficiency compared to state-of-the-art graph mining implementations.

# ACKNOWLEDGEMENTS

This research is partially supported by the PRISM (000705769) center under the JUMP 2.0 program by DARPA/SRC.

# REFERENCES

[1] Mohammad Al Hasan and Vachik S Dave. 2018. Triangle counting in large networks: a review. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 8, 2 (2018), e1226.
[2] Timothy A Davis. 2018. Graph algorithms via SuiteSparse: GraphBLAS: triangle counting and k-truss. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*. IEEE, 1–6.
[3] Timothy A Davis. 2019. Algorithm 1000: SuiteSparse: GraphBLAS: Graph algorithms in the language of sparse linear algebra. *ACM Transactions on Mathematical Software (TOMS)* 45, 4 (2019), 1–25.
[4] Jaeyoung Do, Yang-Suk Kee, Jignesh M Patel, Chanik Park, Kwanghyun Park, and David J DeWitt. 2013. Query processing on smart ssds: Opportunities and challenges. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. 1221–1230.
[5] Tianyu Fu, Chiyue Wei, Zhenhua Zhu, Shang Yang, Zhongming Yu, Guohao Dai, Huazhong Yang, and Yu Wang. 2023. CLAP: Locality Aware and Parallel Triangle Counting with Content Addressable Memory. In *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1–6.
[6] Sayan Ghosh. 2022. Improved distributed-memory triangle counting by exploiting the graph structure. In *2022 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–6.
[7] Sayan Ghosh and Mahantesh Halappanavar. 2020. Tric: Distributed-memory triangle counting by exploiting the graph structure. In *2020 IEEE high performance extreme computing conference (HPEC)*. IEEE, 1–6.
[8] Jiezhong He, Zhouyang Liu, Yixin Chen, Hengyue Pan, Zhen Huang, and Dongsheng Li. 2022. FAST: A Scalable Subgraph Matching Framework over Large Graphs. In *2022 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–7.
[9] Sang-Woo Jun, Andy Wright, Sizhuo Zhang, Shuotao Xu, and Arvind. 2018. GraFBoost: Using Accelerated Flash Storage for External Graph Analytics. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. 411–424. https://doi.org/10.1109/ISCA.2018.00042
[10] Pradeep Kumar and H Howie Huang. 2016. G-store: high-performance graph store for trillion-edge processing. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE.
[11] Pradeep Kumar and H Howie Huang. 2020. Graphone: A data store for real-time analytics on evolving graphs. *ACM Transactions on Storage (TOS)* 15, 4 (2020).
[12] Philip Levis. July 2023. It's the End of DRAM As We Know It. https://datatracker.ietf.org/meeting/117/materials/slides-117-anrw-sessa-keynote-its-the-end-of-dram-as-we-know-it-00.
[13] Steffen Maass, Changwoo Min, Sanidhya Kashyap, Woonhak Kang, Mohan Kumar, and Taesoo Kim. 2017. Mosaic: Processing a trillion-edge graph on a single machine. In *Proceedings of the Twelfth European Conference on Computer Systems*.
[14] Kiran Kumar Matam, Gunjae Koo, Haipeng Zha, Hung-Wei Tseng, and Murali Annavaram. 2019. GraphSSD: graph semantics aware SSD. In *Proceedings of the 46th international symposium on computer architecture*. 116–128.
[15] Sadegh M Milajerdi, Birhanu Eshete, Rigel Gjomemo, and VN Venkatakrishnan. 2019. Poirot: Aligning attack behavior with kernel audit records for cyber threat hunting. In *Proceedings of the 2019 ACM SIGSAC conference on computer and communications security*. 1795–1812.
[16] Santosh Pandey, Xiaoye Sherry Li, Aydin Buluc, Jiejun Xu, and Hang Liu. 2019. H-index: Hash-indexing for parallel triangle counting on GPUs. In *2019 IEEE high performance extreme computing conference (HPEC)*. IEEE, 1–7.
[17] Santosh Pandey, Zhibin Wang, Sheng Zhong, Chen Tian, Bolong Zheng, Xiaoye Li, Lingda Li, Adolfy Hoisie, Caiwen Ding, Dong Li, et al. 2021. Trust: Triangle counting reloaded on GPUs. *IEEE Transactions on Parallel and Distributed Systems* 32, 11 (2021), 2646–2660.
[18] 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. 2017. Static graph challenge: Subgraph isomorphism. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–6. https://doi.org/10.1109/HPEC.2017.8091039
[19] Trevor Steil, Geoffrey Sanders, and Roger Pearce. 2021. Towards distributed square counting in large graphs. In *2021 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–7.
[20] Ancy Sarah Tom, Narayanan Sundaram, Nesreen K Ahmed, Shaden Smith, Stijn Eyerman, Midhunchandra Kodiyath, Ibrahim Hur, Fabrizio Petrini, and George Karypis. 2017. Exploring optimizations on shared-memory platforms for parallel triangle counting algorithms. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–7.
[21] Pinghui Wang, Yiyan Qi, Yu Sun, Xiangliang Zhang, Jing Tao, and Xiaohong Guan. 2017. Approximately counting triangles in large graph streams including edge duplicates with a fixed memory usage. *Proceedings of the VLDB Endowment* 11, 2 (2017), 162–175.
[22] Feng Xia, Ke Sun, Shuo Yu, Abdul Aziz, Liangtian Wan, Shirui Pan, and Huan Liu. 2021. Graph learning: A survey. *IEEE Transactions on Artificial Intelligence* 2, 2 (2021), 109–127.