2020 IEEE High Performance Extreme Computing Conference (HPEC)

Towards an Objective Metric for the Performance of Exact Triangle Count

Mark P. Blanco*, Scott McMillan†, Tze Meng Low*
*Dept. of Electrical and Computer Engineering †Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA, United States
{markb1, scottmc, lowt}@cmu.edu

***Abstract*—The performance of graph algorithms is often measured in terms of the number of traversed edges per second (TEPS). However, this performance metric is inadequate for a graph operation such as exact triangle counting. In triangle counting, execution times on graphs with a similar number of edges can be distinctly different as demonstrated by results from the past Graph Challenge entries. We discuss the need for an objective performance metric for graph operations and the desired characteristics of such a metric such that it more accurately captures the interactions between the amount of work performed and the capabilities of the hardware on which the code is executed. Using exact triangle counting as an example, we derive a metric that captures how certain techniques employed in many implementations improve performance. We demonstrate that our proposed metric can be used to evaluate and compare multiple approaches for triangle counting, using a SIMD approach as a case study against a scalar baseline.**

***Index Terms*—Performance Metric, Graph Algorithms, Triangle Counting, High Performance, CPU, Performance Measurement**

I. Introduction

It is widely accepted that software-hardware co-design is required for attaining high performance implementations. Therefore, any performance metric used in design and evaluation of an implementation must have properties that bridge the gap between a platform's capabilities and the operations inherent to the problem.

For many graph algorithms, traversed edges per second (TEPS) is a widely used figure of merit. While TEPS suggests that performance is related to the number of edges that are traversed (read/written during the course of the computation) over time, a more common (mis)use of the metric is simply the number of edges in the graph divided by execution time. In many works on triangle counting specifically, the metric used is the simpler definition [1]–[5]. From here, we refer to this common usage as edges-per-second.

To illustrate the inadequacy of the edges-per-second metric for a graph operation such as exact triangle count, consider the plot in Fig. 1 wherein we report the metric performance obtained from a sequential implementation

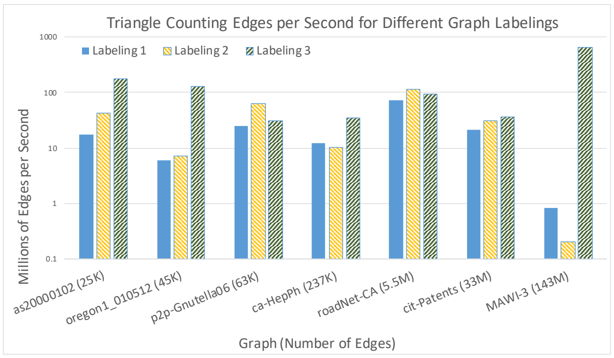

Fig. 1: The inadequacy of the edges-per-second performance metric is demonstrated by how the metric varies widely for the each graph that has been sorted in different order, while still retaining the same number of edges, vertices and triangles.

of triangle count for several graphs from the Graph Challenge Dataset [3], [6]. For each graph, we report performance numbers for three different ways in which the vertices have been labeled and sorted. Notice that despite having the same number of edges, vertices, and triangles, the performance numbers attained for each graph vary dramatically even when using the edges-per-second metric. In essence, the metric is as informative as raw execution time. It provides little insight into the software innovation or the hardware capabilities that contribute to the attained performance.

Samsi et. al. [7], [8] use edges-per-second and the metric:

$$T_{tri} = \left(\frac{N_e}{N_1}\right)^{\beta},$$

where T_{tri}, N_e, and N_1 correspond to the execution time, number of edges in a graph, and number of edges processed in one second. In this metric, β is a value (smaller is better) representing technology advancement that arises from implementation, algorithmic, or hardware improvements. This metric, however, does not explain the difference in execution times seen in Fig. 1. In addition, it is unclear how this metric can be used to account for the different aspects of technology improvements such as hardware differences and algorithmic improvements.

In this paper, we propose two performance metrics (match checks, and match checks per unit time) that we believe are more objective than edges-per-second. These metrics more accurately describe the performance of triangle counting attained in Fig. 1, and better capture the relationship between execution time and the expected amount of work that has to be performed. In addition, the proposed metrics expose the relationship between the amount of work and the hardware capabilities required to perform the required work, which in turn provides newer insights into commonly used techniques employed in many triangle counting implementations.

II. Properties of a Performance Metric

In this section, we describe properties which we consider desirable in an objective performance metric for hardware-software co-design. We illustrate these properties using dense matrix-matrix multiplication where the use of number of floating point operations per second (FLOPS/s) is a well-established performance metric for dense linear algebra.

A. Indicative of the Expected Amount of Work Performed

A performance metric needs to measure the expected amount of useful work performed as increasing the amount of work necessarily increases the execution time. By work, we mean the basic unit of computation (and data movement) that is *necessary to compute the desired result using a generally accepted algorithm*. It is necessary to note that the expected amount of work is not the minimum amount of work that could be performed.

To illustrate this, consider multiplying two matrices of sizes $m \times k$ and $k \times n$. The quantity of work (measured in floating point operations) is approximately $2mnk$ using the traditional triply-nested loop algorithm. Algorithmic innovations such as Strassen [9] reduce the expected amount of work, resulting in a faster execution time and thus a higher FLOPS/s when using $2mnk$ floating point operations as the expected amount of work. This metric can potentially report performance above the theoretical peak of the hardware [10] because the expected amount of work exceeds that of the actual work.

B. Measures Hardware Capabilities

The performance metric has to be sufficiently low-level to expose hardware capabilities related to the work being performed. Hardware with more capability for computing the basic work unit should yield a correspondingly increased score using the metric. When normalized to the theoretical capability of the available hardware, the performance metric should also indicate how well the available hardware is being utilized.

Increased hardware capabilities such as the introduction of the fused-multiply-accumulate unit (FMA) unit and Single Instruction Multiple Data (SIMD) instruction set extensions can be objectively quantified using FLOPS/s as these hardware capabilities allow more floating point operations to be computed per unit time. Comparing attained FLOPS/s as a function of percentage of theoretical peak hardware capability allows for introspection on implementations to determine if more can be done to achieve better performance.

C. Captures Implementation Innovations

Implementation innovations, such as tiling and data layout changes, increase performance through improved data access while maintaining the amount of work that is performed. A performance metric should reflect these innovations with a better score despite having no change in both the hardware or the expected amount of work.

A high performance matrix matrix multiplication is often implemented as multiple nested loops that partition a matrix into submatrices through loop tiling/blocking [11], [12]. In addition, input matrices are repacked in order to ensure that 1) data is brought into the appropriate level of caches and 2) data is repacked such that accesses can be performed with unit stride [13]. These implementation choices maintain the same amount of (useful) work, but increase the performance due to better data access. These benefits are demonstrated via a higher FLOPS/s score.

D. Application to Graph Algorithms

While we have identified the desirable properties of a performance metric, the vertices of a graph can be relabeled and reordered without changing the structure of the graph. Moreover, the adjacency matrix of the different isomorphic graphs often exhibit different structures that may change the amount of work that needs to be computed. As a convention, we take the original unsorted graph as the canonical graph from which the expected amount of work is computed.

III. Deriving a Metric for Triangle Count

For the sake of completeness, we begin with a brief description of triangle count and commonly used approaches for computing the number of triangles in a graph. We then derive a plausible performance metric for the triangle count operation: match checks.

A. Triangle Counting

Triangle counting, as its name suggests, counts the number of triangles in a undirected graph G. This graph can be represented by its adjacency matrix which is a symmetric sparse matrix. For the purposes of this paper, we assume that only the lower-triangular portion of the adjacency matrix is stored.

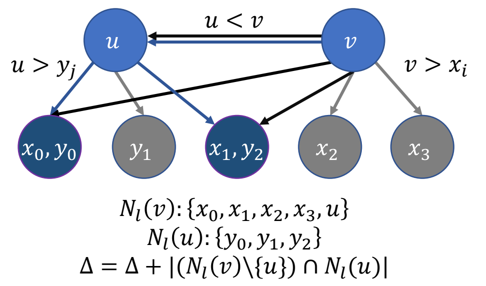
$u < v$
u
v
$u > y_j$
$v > x_i$
x_0, y_0
y_1
x_1, y_2
x_2
x_3
$N_l(v)$: $\{x_0, x_1, x_2, x_3, u\}$
$N_l(u)$: $\{y_0, y_1, y_2\}$
$\Delta = \Delta + |(N_l(v) \backslash \{u\}) \cap N_l(u)|$

Fig. 2: Diagram of connected vertices v and u. To count triangles, their lower neighborhoods are intersected, with u subtracted from $N_l(v)$. Δ is the triangle count.

Processing only the lower-triangular (or upper) part of the adjacency matrix allows one to count each triangle exactly once. This approach is common in many implementations [14]–[16]. Elements in this part of the matrix correspond to edges leading from one vertex to a neighbor with lower vertex ID. This portion of each vertex's neighborhood is referred to as the lower neighborhood, denoted as $N_l(v)$ for vertex v.

B. *Exact triangle count with wedges and intersections*

In general, there are two approaches to counting the number of triangles in a graph. The first method is the wedge-check method, where given a wedge (i.e. three vertices connected with two edges), a check is performed to test if there exists an edge that closes the wedge into a triangle. The second approach, based on set-intersection, identifies if there exists a common vertex in the neighborhood of the two vertices of an edge.

We demonstrate the equivalence of the two approaches using Fig. 2. In the following discussion, $v > u$ for both approaches. In the wedge-check method, the wedges between each pair of vertices v, u are found by way of the shared neighbors (e.g. the vertices labeled x_0/y_0 and x_1/y_2) between their lower neighborhoods $(N_l(v), N_l(u))$. The existence of the triangle is then confirmed by checking for the edge directly connecting v to u. In the set-intersection approach, for the endpoints of an existing edge (v, u), their lower neighborhoods are intersected to find shared neighbors (again, x_0/y_0 and x_1/y_2 in the diagram). Because the intersection is only performed given that edge (v, u) exists, each shared neighbor found in the intersection indicates a triangle.

In practice, work for the first approach is often pre-filtered by existing edges, fusing the two steps and making the two approaches equivalent. These ways of describing triangle counting mechanics are also compatible with sparse-matrix based approaches. In particular, the work by Wolf et al. for miniTri specifically highlights a wedge-check approach based on a linear algebraic specification [17].

Algorithm 1: Scalar merge-based set intersection kernel. Asterisk (*) before an address symbol indicates access to memory.

```
Input: a1 and a2 initially give the start addresses of each set.
       a1_nd and a2_nd are the addresses just after the end
       of each set.
Output: $\Delta$ is the number of matches among both sets.
$\Delta \leftarrow 0$
while $a1 < a1\_nd \wedge a2 < a2\_nd$ do
    if $*a1 == *a2$ then
        a1 $\leftarrow$ a1 + 1
        a2 $\leftarrow$ a2 + 1
        $\Delta \leftarrow \Delta + 1$
    else if $*a1 > *a2$ then
        a2 $\leftarrow$ a2 + 1
    else
        a1 $\leftarrow$ a1 + 1
return $\Delta$
```

From either perspective, the work of finding wedges that may close into triangles constitutes an intersection of $N_l(v)$ and $N_l(u)$. Algorithm 1 illustrates the core operation in triangle counting by way of a naive approach for set intersection, commonly referred to as merge-based set intersection as in [18], [19].

C. *Expected work for Exact Triangle Count*

Given the previous discussion, the core operation in triangle counting can be viewed as finding the size of neighborhood set intersections. Given two neighborhoods of vertices, the first vertex in each of the two neighborhoods are compared (intersected). In the case of a match, the match is recorded and the next vertices in each neighborhood are compared. Otherwise, the next vertex in the neighborhood of the vertex with the smaller of the IDs is compared with the vertex with the larger ID. This process is repeated until all vertices in one or both neighborhoods have been checked.

Notice that this work has to be performed even if there are no triangles in the graph since the absence of any triangle can only be ascertained after iterating through the neighborhoods. This suggests that the number of match checks a graph requires better reflects the amount of work in triangle counting than the number of edges does. In addition, the number of match checks performed by triangle counting equals the number of iterations executed by the loop in Algorithm 1. From this point in the paper, we refer to such wedge or vertex comparisons as *match checks or matches.*

D. *Match Checks in Hardware*

As a metric, match checks do not prescribe which instructions or implementation should be used, but generally indicate that some form of compare instruction followed by conditional updates are required for each

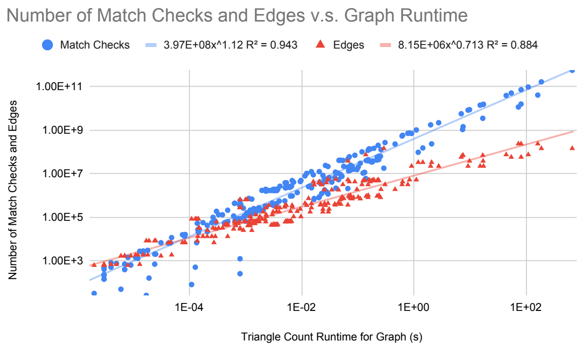

Fig. 3: Number of match checks and edges plotted against runtime for scalar merge-sort-based triangle counting, for 3 graph orderings per graph. Increased number of match checks and edges both generally correspond to increased runtime. Match checks show a stronger fit to runtime ($r^2 = 0.94$) compared to raw number of edges ($r^2 = 0.884$).

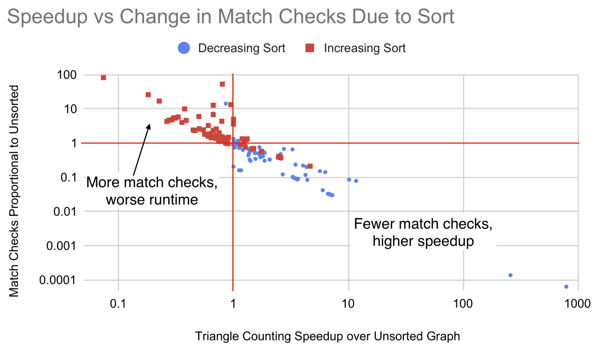

Fig. 4: Speedup over non-sorted graph plotted against the change in number of match checks performed for different graph sorts. Different markers represent sort orders. Different sort orders are necessary on different graphs to reduce match checks and execution time.

match check. Possible implementations of Algorithm 1 include 1) a compare operation followed by branches, and 2) predicated instructions to avoid branches [19].

However, knowing how match checks are mapped (either manually or by the compiler) to specific instructions or hardware components allows us to use match checks as a proxy for the amount of hardware resources required to compute the necessary amount of work. For example, mapping match checks to a compare instruction naturally limits the rate of match checks to the throughput of the compare (`CMP`) instruction on a given architecture.

IV. Applying Match Checks as a Metric

In this section, we demonstrate that match checks are an acceptable metric for work in triangle counting.

A. Matches Reflecting Work in Triangle Count

We validate the use of match checks as a metric of the work for triangle count in Fig. 3. Each point represents a graph from a real or synthetic graph in the Graph Challenge dataset, processed using a sequential merge-based triangle count implementation [3], [6]. The number of match checks per graph was obtained from a second non-timing run, shown in blue. As expected, the runtime increases with increased number of match checks across the 93 graphs. Hence, match checks are a representative metric for the work in triangle counting. By contrast, we also show the number of edges in each graph in red in Fig. 3. While there is clearly some correlation between the number of edges in a graph and the number of checks, it is a weaker one.

B. Matches Reflecting Implementation Innovation

It has been observed in numerous works that graph reordering can improve triangle counting performance. Chiba and Nishizeki observed that processing vertices in decreasing order of degree enables better running-time bounds for triangle counting based on the graph's aboricity [20]. Recent Graph Challenge implementations such as Bisson et al. attribute improvements in performance to more balanced threads on the GPU [21]. More generally for graph algorithms, Balaji et al. note improvements due to improved memory layout and access patterns [22]. Using the proposed matches-checked metric, we show that new insights can be gained into the need for sorting.

Observation 1: Sorting changes the number of match checks performed. Besides load-balancing, sorting in the appropriate fashion can significantly reduce the amount of work. Fig. 4 shows the speedup of triangle counting against the decrease (below 1) or increase (above 1) in proportion of match checks over the unsorted original graph for sequential triangle count. The number of match checks is changed by relabeling vertices based on degree in decreasing order (blue) or increasing order (red). For most graphs, we observe that sorting changes the number of match checks performed, and thus a corresponding change in execution time is observed.

Observation 2: Different label orders are required for different graphs. While sorting vertices by decreasing vertex degree often leads to lower execution time, a number of graphs benefit from sorting in the reversed order. This is demonstrated in Fig. 4 when certain red triangular markers are in the bottom right quadrant, while a small number of blue circular markers are in the top left quadrant. This suggests that a possible area of future research into heuristics that determine the appropriate sort order or alternative labeling for a given graph.

C. Matches Reflected in Hardware Capability

Here we introduce a case study of using match checks to evaluate an alternate approach (SIMD)

Algorithm 2: SIMD 8x8 set intersection kernel. Leftover set elements go to the scalar kernel.

```
Input: a1 and a2 initially give the start addresses of each set.
       a1_nd and a2_nd are the addresses just after the end
       of each set.
Output: $\Delta$ is the number of matches among both sets.
$\Delta \leftarrow 0$
while $a1 < a1\_nd \wedge a2 < a2\_nd$ do
    a1_max ← a1[7]
    a2_max ← a2[7]
    // Early termination checks
    a1_min ← a1[0]
    a2_min ← a2[0]
    if $a1\_max < a2\_min$ then
        a1 ← a1 + 8; continue
    if $a2\_max < a1\_min$ then
        a2 ← a2 + 8; continue
    vector_1[0:7] ← a1[0:7]
    // packed load
    vector_2_1[0:7] ← a2[0]
    // broadcast
    // Omitted: similar broadcasts for
     a2[1] through a2[7]...
    cmp_mask_1 ← cmp_eq(vector_1, vector_2_1)
    cmp_mask_2 ← cmp_eq(vector_1, vector_2_2)
    cmp_mask_3 ← cmp_eq(vector_1, vector_2_3)
    cmp_mask_4 ← cmp_eq(vector_1, vector_2_4)
    cmp_mask_5 ← cmp_eq(vector_1, vector_2_5)
    cmp_mask_6 ← cmp_eq(vector_1, vector_2_6)
    cmp_mask_7 ← cmp_eq(vector_1, vector_2_7)
    cmp_mask_8 ← cmp_eq(vector_1, vector_2_8)
    // Omitted: logically or all cmp_masks
     together into and_mask...
    $\Delta \leftarrow \Delta$ + popcount(and_mask)
    if $a1\_max \leq a2\_max$ then
        a1 ← a1 + 8
    if $a2\_max \leq a1\_max$ then
        a2 ← a2 + 8
return $delta$
```

against a baseline (merge-based scalar) with a focus on hardware-software co-design. Single-instruction multiple-data (SIMD) hardware is commonly used in regular applications due to the higher throughput it affords. Use of SIMD in graphs is less common as graph algorithms (inclusive of triangle counting) are considered irregular. Using match checks, we showcase that SIMD is measurably faster than the scalar approach owing to performing more *effective* match checks per cycle.

Recall that match checks can be used as a proxy for the number of times a compare (`CMP`) instruction is required. The use of SIMD compare instructions (e.g. `VPCMPEQ`) can increase the number of comparisons that are performed, thus increasing the overall throughput.

We implemented multiple SIMD set intersection kernels, one of which is shown in Algorithm 2. This kernel compares eight vertices from each neighborhood against each other. This effectively computes 64 match checks within the kernel. However, notice that the number of

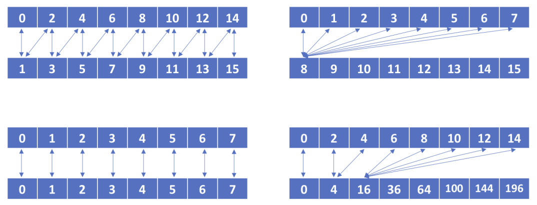

Fig. 5: Different distributions of vertex IDs results in different number of match checks. Arrows show the match checks that will be performed by Algorithm 1. The interleaved pattern has the largest number of match checks (15) between two neighborhoods of size 8 each.

SIMD match checks corresponding to match checks which the scalar implementation would have performed depends on the distribution of vertex IDs in each vector. The impact of the different distribution of vertex IDs is illustrated in Fig. 5. In our analysis, we therefore consider *effective* match checks, where performance of the SIMD approach is measured based on the number of effective match checks it performs relative to the scalar baseline. The remaining match checks performed in the SIMD approach are considered wasted work.

The performance of triangle count using the scalar and SIMD versions of the set-intersection kernels, in match checks-per-cycle, is reported in Fig. 6 for the following systems:

- Intel Xeon E5-2667 CPU (Haswell)
- Intel i7-7700K CPU (Kaby Lake)
- Intel Xeon Platinum 8153 CPU (Skylake-X)

Sequential performance numbers are shown for all graphs, which are relabeled in decreasing vertex degree order. Sorting time is NOT included in the overall execution time. The graphs in each plots are ordered by increasing graph size (size in memory).

The utility of SIMD set intersection is mixed on the sorted graphs. For smaller graphs, scalar outperforms SIMD. This is likely due to smaller neighborhoods in small graphs reducing opportunities to apply the SIMD kernels. The added logic for switching between multiple kernels and the wasted work in the SIMD approach are potential reasons for diminished performance of the SIMD implementation. Additionally, the standard deviation of the lower-neighborhood size for most graphs is reduced after sorting, so there are fewer large neighborhoods that SIMD can process. For larger graphs, in spite of the wasted work, SIMD is often faster than the scalar implementation. This implementation innovation is reflected in the effective match checks per cycle shown in Fig. 6, particularly for larger graphs.

V. Discussion and Conclusion

In this work, we proposed the use of match checks and match checks per cycle as a performance metric for exact

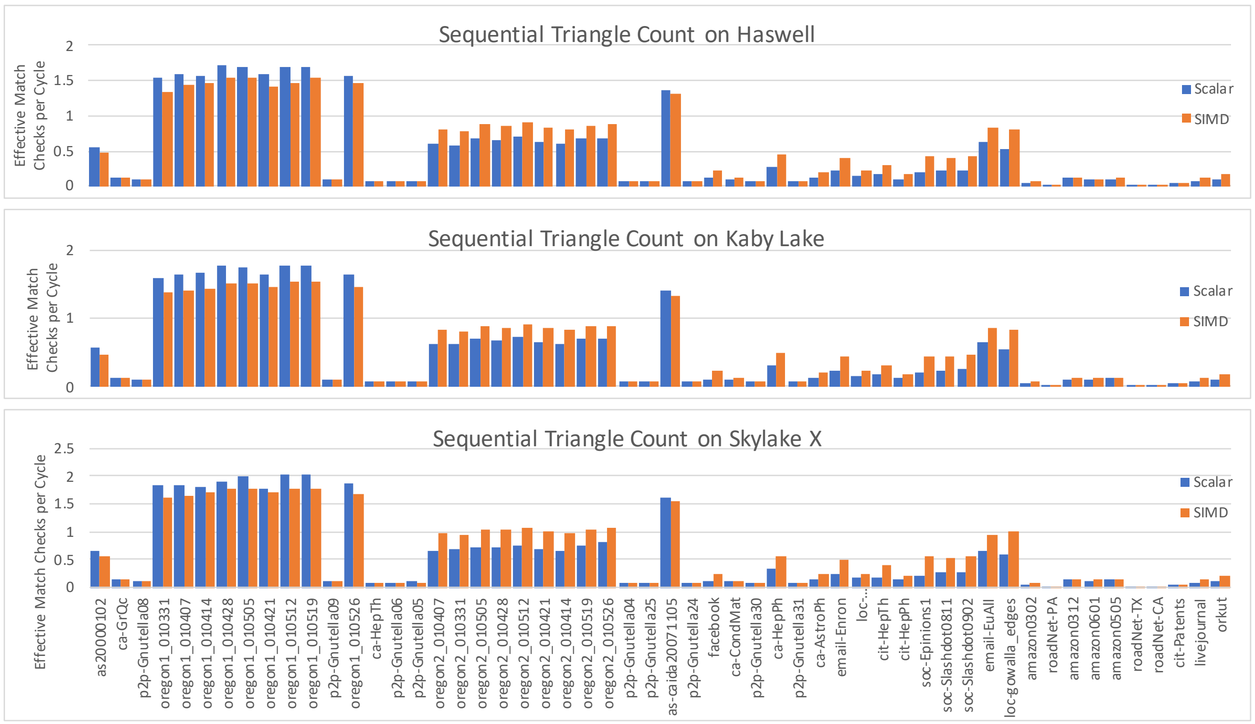

Fig. 6: Scalar and SIMD triangle count performance, in match checks-per-second, for a variety of graphs from the SNAP dataset on Top) Haswell, Middle) Kabylake, and Bottom) Skylake-X architectures. Use of of SIMD instructions results in higher attained performance for graphs with larger neighborhoods.

triangle counting. This is motivated by the need for a metric that explains the performance of triangle counting implementations in terms of the amount of work and the hardware capabilities of the system.

We demonstrated that the metric represents the amount of work that has to be performed in triangle counting. The metric also provides new insights into commonly-employed techniques found in many triangle counting implementations such as sort-based relabeling. While we have focused on only sequential implementations, we believe that extending the metric to represent parallel implementations should be straight-forward.

The observant reader may note that match checks and the number of actually traversed edges in the graph are very similar in numeric value. In fact, the actual number of edges traversed equals the number of match checks plus the number of triangles in the graph. The case we make in this work is not that match checks are better than counting the number of truly traversed edges, but that edges-per-second as used in many recent works is not appropriate and some hardware-focused metric like match checks is needed for hardware-software co-design.

More generally, we believe that the approach we took to identify the proposed metric can be replicated for different (classes of) graph algorithms to identify objective metrics. Having better metrics would provide greater insights into the amount of work to be performed and the hardware capabilities that are required. These insights could lead to faster and more efficient software implementations, and could also suggest hardware features needed by graph algorithms.

Identified metrics for one graph algorithm could potentially apply to other graph algorithms. For example, the set intersection kernel in triangle counting is similar to a sparse dot product. This suggests that other graph algorithms and sparse linear algebraic workloads may benefit from similar metrics for performance evaluation. We will pursue these directions in future work.

Acknowledgement

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center [DM20-0657].

Mark Blanco is supported by the National Science Foundation Graduate Research Fellowship Program under Grant No. DGE 1745016. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

[1] M. M. Wolf, M. Deveci, J. W. Berry, S. D. Hammond, and S. Rajamanickam, "Fast linear algebra-based triangle counting with KokkosKernels," in *2017 IEEE High Performance Extreme Computing Conference (HPEC)*. Waltham, MA: IEEE, Sep. 2017, pp. 1–7. [Online]. Available: http://ieeexplore.ieee.org/document/8091043/

[2] Y. Hu, P. Kumar, G. Swope, and H. H. Huang, "TriX: Triangle counting at extreme scale," in *2017 IEEE High Performance Extreme Computing Conference (HPEC)*. Waltham, MA: IEEE, Sep. 2017, pp. 1–7. [Online]. Available: http://ieeexplore.ieee.org/document/8091036/

[3] S. Samsi, V. Gadepally, M. Hurley, M. Jones, E. Kao, S. Mohindra, P. Monticciolo, A. Reuther, S. Smith, W. Song, D. Staheli, and J. Kepner, "Static Graph Challenge: Subgraph Isomorphism," Aug. 2017. [Online]. Available: https://arxiv.org/abs/1708.06866v1

[4] 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)*, Sep. 2018, pp. 1–7, iSSN: 2377-6943.

[5] M. Bisson and M. Fatica, "Static graph challenge on GPU," in *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, Sep. 2017, pp. 1–8.

[6] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, Jun. 2014.

[7] S. Samsi, V. Gadepally, M. Hurley, M. Jones, E. Kao, S. Mohindra, P. Monticciolo, A. Reuther, S. Smith, W. Song *et al.*, "Graphchallenge. org: Raising the bar on graph analytic performance," in *2018 IEEE High Performance extreme Computing Conference (HPEC)*. IEEE, 2018, pp. 1–7.

[8] S. Samsi, J. Kepner, V. Gadepally, M. Hurley, M. Jones, E. Kao, S. Mohindra, A. Reuther, S. Smith, W. Song, D. Staheli, and P. Monticciolo, "Graphchallenge.org triangle counting performance," 2020.

[9] V. STRASSEN, "Gaussian elimination is not optimal." *Numerische Mathematik*, vol. 13, pp. 354–356, 1969. [Online]. Available: http://eudml.org/doc/131927

[10] J. Huang, T. M. Smith, G. M. Henry, and R. A. van de Geijn, "Strassen's algorithm reloaded," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '16. IEEE Press, 2016.

[11] K. Goto and R. A. v. d. Geijn, "Anatomy of high-performance matrix multiplication," *ACM Transactions on Mathematical Software*, vol. 34, no. 3, pp. 1–25, May 2008. [Online]. Available: https://dl.acm.org/doi/10.1145/1356052.1356053

[12] F. G. Van Zee and R. A. van de Geijn, "Blis: A framework for rapidly instantiating blas functionality," *ACM Trans. Math. Softw.*, vol. 41, no. 3, Jun. 2015. [Online]. Available: https://doi.org/10.1145/2764454

[13] G. Henry, "Blas based on block data structures," Cornell University, Tech. Rep., 1992.

[14] M. Lee and T. M. Low, "A Family of Provably Correct Algorithms for Exact Triangle Counting," in *Proceedings of the First International Workshop on Software Correctness for HPC Applications*, ser. Correctness'17. New York, NY, USA: ACM, 2017, pp. 14–20, event-place: Denver, CO, USA. [Online]. Available: http://doi.acm.org/10.1145/3145344.3145484

[15] A. Azad, A. Buluç, and J. Gilbert, "Parallel Triangle Counting and Enumeration Using Matrix Algebra," in *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, May 2015, pp. 804–811.

[16] C. Voegele, Y.-S. Lu, S. Pai, and K. Pingali, "Parallel triangle counting and k-truss identification using graph-centric methods," in *2017 IEEE High Performance Extreme Computing Conference (HPEC)*. Waltham, MA, USA: IEEE, Sep. 2017, pp. 1–7. [Online]. Available: http://ieeexplore.ieee.org/document/8091037/

[17] M. M. Wolf, J. W. Berry, and D. T. Stark, "A task-based linear algebra Building Blocks approach for scalable graph analytics," in *2015 IEEE High Performance Extreme Computing Conference (HPEC)*, Sep. 2015, pp. 1–6.

[18] H. Inoue, M. Ohara, and K. Taura, "Faster set intersection with SIMD instructions by reducing branch mispredictions," *Proceedings of the VLDB Endowment*, vol. 8, no. 3, pp. 293–304, Nov. 2014. [Online]. Available: http://dl.acm.org/doi/10.14778/2735508.2735518

[19] J. Zhang, Y. Lu, D. G. Spampinato, and F. Franchetti, "FESIA: A fast and simd-efficient set intersection approach on modern cpus," in *36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020*. IEEE, 2020, pp. 1465–1476. [Online]. Available: https://doi.org/10.1109/ICDE48307.2020.00130

[20] N. Chiba and T. Nishizeki, "Arboricity and Subgraph Listing Algorithms," *SIAM Journal on Computing*, vol. 14, no. 1, pp. 210–223, Feb. 1985. [Online]. Available: http://epubs.siam.org/doi/10.1137/0214017

[21] M. Bisson and M. Fatica, "Update on Static Graph Challenge on GPU," in *2018 IEEE High Performance extreme Computing Conference (HPEC)*, Sep. 2018, pp. 1–8.

[22] V. Balaji and B. Lucia, "When is Graph Reordering an Optimization? Studying the Effect of Lightweight Graph Reordering Across Applications and Input Graphs," in *2018 IEEE International Symposium on Workload Characterization (IISWC)*, Sep. 2018, pp. 203–214.