2024 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW) | 979-8-3503-6460-6/24/$31.00 ©2024 IEEE | DOI: 10.1109/IPDPSW63119.2024.00106

Unlocking the Potential: Performance Portability of Graph Algorithms on Kokkos Framework

Shaikh Arifuzzaman*, Hasan S. Arikan*, M.A.M. Faysal*, Maximilian Bremer†, John Shalf†, Doru Popovici†
*University of Nevada, Las Vegas (UNLV), USA. †Lawrence Berkeley National Laboratory, USA.
Emails: shaikh.arifuzzaman@unlv.edu, {arikan, faysal}@unlv.nevada.edu, {mb2010, jshalf, dtpopovici}@lbl.gov

***Abstract*—Graph algorithms are fundamental tools for deriving valuable insights from complex network structures. As network data grows in scale, and hardware architecture becomes more diverse than ever, the demand for efficient and portable graph algorithms has surged. The Kokkos framework has emerged as a promising solution for achieving high performance and portability in parallel computing applications. This paper presents a thorough evaluation of Kokkos-implemented triangle counting, an important graph kernel. Employing diverse algorithmic and implementation methods, we benchmark Kokkos-enabled graph algorithms targeting CPUs and GPUs. We explore the impact of both graph properties and Kokkos' parallel execution model on algorithmic efficiency. Our results indicate that thread scheduling can improve performance by up to $10\times$, data structure choice by $6\times$, and configuring the parallel hierarchy based on degree properties can result in a remarkable $300\times$ difference in performance over untuned implementations on Kokkos.**

***Index Terms*—Graph algorithms, performance, portability, linear algebra, large graphs, sparse graphs, GPU**

I. Introduction

Graph algorithms play a critical role in analyzing complex network data, finding applications in various fields such as social network analysis, recommendation systems, and bioinformatics [8], [9], [18], [19]. As the size and complexity of graphs continue to grow exponentially, there is a pressing need for efficient parallel processing techniques to handle these massive datasets [6], [16], [19]. The Kokkos framework [18] has emerged as a promising solution for achieving performance portability across different high-performance computing architectures. This paper aims to investigate the performance and portability of graph analytic kernels using vertex-iterative [10] and linear algebra-based [12] methods on the Kokkos framework. We focus on triangle counting [7].

The Kokkos framework [18] is specifically designed to facilitate performance portability across diverse parallel architectures, including CPUs, GPUs, and accelerators [11], [14]. It provides a programming model that enables developers to write code once and target multiple architectures without extensive modifications. By harnessing the capabilities of Kokkos, graph algorithm implementations have the potential to exploit the computational power of various hardware platforms. However, the performance and portability of diverse graph algorithms with a range of data structure choices and varied graph properties on the Kokkos framework have not been thoroughly explored, to the best of our knowledge, highlighting the need for this research.

The Challenges for Performance Portability. Developing performance portable codes for modern heterogeneous architectures is a complex and challenging undertaking [15]. The task of achieving performance on a CPU is already cumbersome. Porting algorithms to GPUs becomes even more complicated due to the high number of threads that work in parallel. Consequently, it is highly desirable for programmers and practitioners to strive for performance portability through a single-source implementation or a multi-variant single programming model. Several programming models, such as OpenMP [4], Kokkos [18], Raja [17], and OpenCL, offer portability across diverse architectures. The achievement of performance portability through a programming model depends on the capacity to express algorithmic variants, scheduling, and optimization strategies. The work in this paper aims to investigate the tradeoffs between the flexibility of a the programming model and the different design choices.

Our Specific Aims. We comprehensively evaluate the performance and portability of vertex-iterative and linear algebra based graph algorithms on the Kokkos framework. The primary objective is to analyze the impact of different architectural features and programming choices on the performance of these algorithms. The findings of this research have broad implications for unlocking the potential of the Kokkos framework as a high-performance and portable graph analytics tool. Moreover, the outcomes of this study can guide the development of optimized implementations and algorithms that fully exploit the features and capabilities of a particular framework (e.g., Kokkos). This can also guide the design of a framework considering all the performance characteristics of a particular application domain that it targets.

Contributions. Our contributions in this paper are as follows. Firstly, we evaluate the portability of triangle counting implemented using the Kokkos framework. Secondly, we provide a thorough analysis of the different design choices, e.g, data layout, level of parallelism, load balancing. Lastly, we draw preliminary conclusions based on the different design choices.

II. Background

A. Triangle Counting in Graphs

Triangle counting [6], [8] is a popular graph analysis kernel used for assessing the quality of network structure in social networks, computing clustering coefficients, identifying communities in graphs, detecting anomalies, and so on.

General overview. The provided network is represented as $G(V, E)$, where V and E denote the sets of vertices and edges,

979-8-3503-6460-6/24/$31.00 ©2024 IEEE
DOI 10.1109/IPDPSW63119.2024.00106

TABLE I: Naming of different Triangle Counting algorithmic implementations we used for experiment

Name	Execution Domain	Description
TC-Bin-aln	CPU	Our vertex-centric, standalone implementation using binary search; based on [8]
TC-Hash-aln	CPU	Our vertex-centric, standalone implementation using hash map; based on [8]
TC-Bin-kok	CPU, GPU	Our vertex-centric, Kokkos implementation using binary search; based on [8]
TC-Hash-kok	CPU	Our vertex-centric, Kokkos implementation using hash map; based on [8]
TC-gapbs	CPU	Vertex-centric implementation provided in GAP benchmark suite [10]
TC-kkernel	CPU	Linear Algebra-based, Kokkos implementation based on Wolf et al. [19]
TC-lagraph	CPU	Linear Algebra-based implementation based on Davis et al. [12]

respectively. We consider the input graph undirected, meaning that if $(u, v) \in E$, we say that both u and v are neighbors of each other. A triangle is a set of three nodes $u, v, w \in V$ such that there is an edge between each pair of these three nodes, i.e., $(u, v), (v, w), (w, u) \in E$.

Vertex-centric Approach to Triangle Counting. The vertex-centric method for triangle counting involves iterating through each node $v \in V$ and determining the number of edges among its neighbors, essentially counting the pairs of neighbors that form a triangle with vertex v. Most such algorithms use a total ordering $\prec$ of the nodes to avoid duplicate counts of the same triangle. For instance, in [6], [8], a degree based ordering is used as defined below:

$$u \prec v \iff d_u < d_v \text{ or } (d_u = d_v \text{ and } u < v). \quad (1)$$

Linear Algebra Approach to Triangle Counting. In a linear algebra formulation, let A be the adjacency matrix of the graph, where $A[i][j] = 1$ if there is an edge between nodes i and j, and $A[i][j] = 0$ otherwise. The number of triangles in the graph can be calculated using matrix multiplication as follows: $T = \frac{1}{6}\text{trace}(A^3)$. In [19], Wolf et al. use a Kokkos-based SpGEMM method, KKMEM. Triangles are counted using the sparse matrix-matrix multiplication: $D = (L \times L).\times L$. They also add a constraint such that $C(v_1, v_2)$ (resulting from $L \times L$) is nonzero if and only if $v_1 > v_2$, which reduces the wedges stored in C leading to a reduction in operations and runtime.

B. Kokkos Overview

Kokkos [18] is an open-source C++ template and metaprogramming library focused on performance portability. Its goal is to be architecture-agnostic, allowing programmers to abstract away low-level details of different hardware architectures and programming models. It is implemented as a template library on top of various high-performance computing programming models, such as CUDA and OpenMP. The parallel Kokkos constructs consist of four main components: a string for debugging and profiling, the number of iterations for a for-loop related to the size of arrays, a C++ lambda for loop indexing, and a functor C++ object acting as a function and holding information about the computation for each iteration on Kokkos arrays (example in [2]). The fundamental data structure used in Kokkos programming is the *view* construct, representing an array of zero or more dimensions. The *view* provides abstractions for three core concepts in the Kokkos memory model: memory space, memory layout, and memory trait. Kokkos provides two primary data-parallel constructs: *parallel for* and *parallel reduce*, which can be executed using different policies: single range (SR), multidimensional range (MD), and hierarchical parallelism (HR) (more details in [1]).

```
Kokkos::parallel_reduce("Triangle-Count",
team_policy(vertices -1, 1024), KOKKOS_LAMBDA(
const member_type& team, int &outer_reduction){
    int inner_reduction_result = 0;
    int a = team.league_rank();
    int i = indexPointerView(a);
    int limit = indexPointerView(a + 1);
    Kokkos::parallel_reduce(Kokkos::
    TeamThreadRange (team, i, limit), [=] (int b,
    int& inner_reduction){
      int ngbr = indicesView(b);
      int j = indexPointerView(ngbr);
      int limit2 = indexPointerView(ngbr+1);
      int k = b;
      while(k < limit && j < limit2){
        if(indicesView(k) > indicesView(j)) {
          j++;
        } else if(indicesView(j) > indicesView(k)){
          k++;
        } else {
          j++; k++; inner_reduction++;
        }
      }
    }, inner_reduction_result);
Kokkos::single(Kokkos::PerTeam(team), [&] (){
    outer_reduction += inner_reduction_result;
}); }, result);
```

Fig. 1: Example of hierarchical parallelism (HR) on GPU using the Kokkos framework, enabling fine grain parallelism.

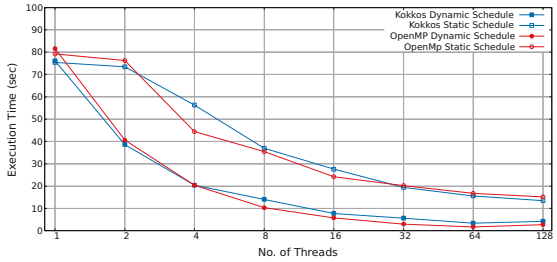
III. Evaluation

A. Experimental Settings

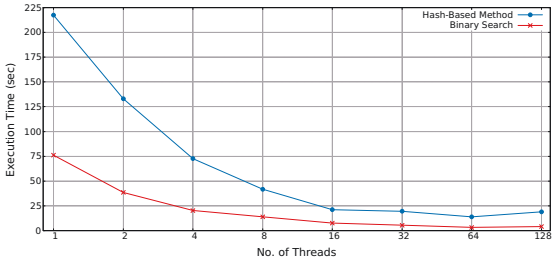
We implemented our algorithms using C++ programming language, OpenMP frameworks for multi-threading, and GNU g++ compiler for building the code. We used *Perlmutter* CPU and GPU compute nodes from National Energy Research Scientific Computing Center [5]. We use network datasets, both real-world and synthetic, representing diverse domains, e.g., social, road networks, kronecker graphs, etc. The datasets are collected from SNAP [16] and SuiteSparse [13].

B. Evaluating Triangle Counting (TC) Kernel

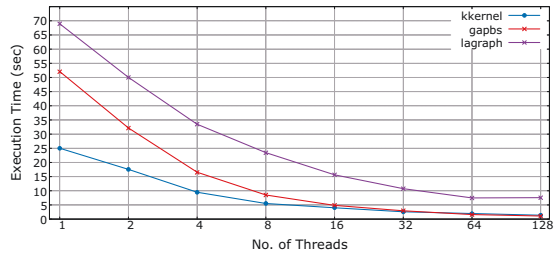
Framework Overhead for Multicore Execution. Frameworks like Kokkos provide performance portability by encapsulating high-performance programming models such as CUDA and OpenMP, albeit with some overhead. Our triangle count implementations in Kokkos, both hash and binary, are comparable to OpenMP-only implementations (Figure 2(**a**)), with expected slight overhead, which is deemed acceptable considering Kokkos' flexibility across CPUs and GPUs.



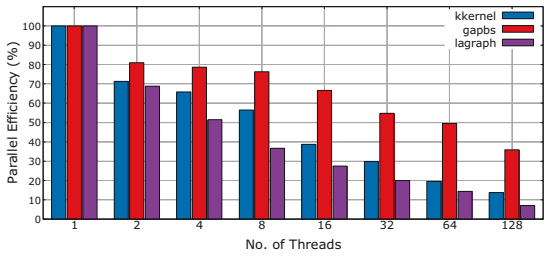
(a) Kokkos versus OMP-only Implementations on CPU



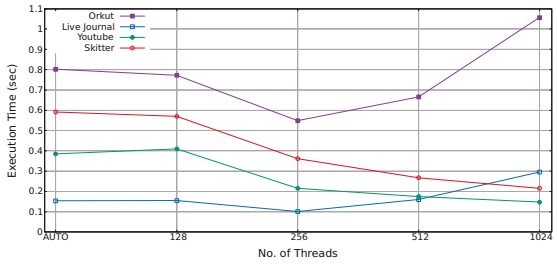
(b) Hash versus Binary Implementations in Kokkos on CPU



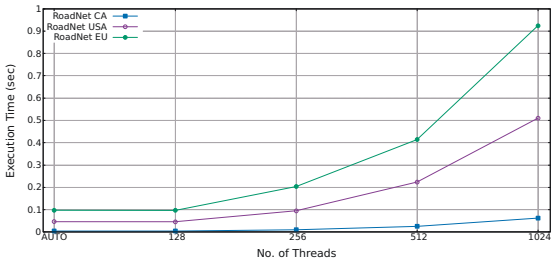
(c) Scalability of TC on CPUs



(d) Parallel Efficiency on CPUs



(e) GPU Policies for TC for Social Networks



(f) GPU Policies for TC for Road Networks

Fig. 2: Performance evaluation of triangle counting kernels across various data structures, thread schedules, implementation methods, GPU thread hierarchies, and graph properties. We also compared Kokkos with OpenMP-only implementations.

Effects of Scheduling the Computation. Graph properties like sparsity and skewness affect performance. Hence, scheduling the computation is extremely important. For example, Orkut is a social network that has a few nodes with a large number of neighbors, and most with fewer connections. We experiment with static and dynamic scheduling on both CPUs and GPUs. Static scheduling distributes a fixed amount of iterations, which creates imbalance. Dynamic scheduling allows work stealing to provide better balance of the computation. Figure 2(**a**) shows that dynamic scheduling provides a performance improvement by almost 3x. On GPUs, dynamic scheduling degrades performance, because GPUs already have a dynamic scheduling heuristic when launching the warps.

Effect of Data Structures. In our vertex-centric triangle counting implementation, we evaluated two data structures and search techniques–binary search and the hash-based methods. We observe that binary search is significantly faster than the hash-based method as outlined in Figure 2(**b**) when the neighbor lists are sorted. We used the C++ unordered set as the hash-based data structure. Hash operations like insertion or look-up may suffer from branch misprediction and collision chaining, degrading performance as dataset grows.

Vertex-centric and Linear Algebra Approaches. Figure 2(**c** and **d**) illustrate scalability for vertex-centric (gapbs) and Linear Algebra-based triangle counting methods (kkernel and LAGraph). In a sequential setting, gapbs initially has longer runtimes due to vertex numeric ordering, but eventually catches up with kkernel as thread count increases. Despite employing an $(L \times L).\times L$ approach, LAGraph exhibits higher runtimes than both gapbs and kkernel, indicating potentially inefficient GraphBLAS implementation compared to kkernel's algorithm. In parallel efficiency (Figure 2(**d**)), LAGraph's implementation displays a steeper decline (from 61.92% to 6.77% in 128 threads) compared to kkernel (from 73.79% to 18.89% in 128 threads).

Effects of Graph Properties for Kokkos-GPU. We examine hierarchical parallelism's impact on networks with varying degree distributions in GPU execution. The degree distribution

influences runtime based on the number of threads within a team, analogous to CUDA thread blocks [3]. We observe that for networks with higher degree skewness (Orkut and Livejournal), increasing team thread sizes beyond 256 leads to runtime degradation due to thread contention for limited thread block registers. Conversely, networks with lower skewness (YouTube or Skitter) benefit from increasing thread sizes to 512 and 1024, as it helps mask latency and saturate bandwidth without significant contention for registers.

Effects of Using Hierarchical Threads. In CPU executions, introducing thread hierarchy does not improve performance compared to single-level parallelism. However, in GPU computations, thread hierarchy significantly impacts performance. Simply parallelizing the outermost loop is not sufficient for fine granularity and full thread utilization on the GPU. To address this in Kokkos, we implement a league structure, associating it with the outer loop's total indices. We then use the `TeamThreadRange` construct with various team sizes to saturate a CUDA block, which can have a maximum of 1024 threads. Optimal team sizes vary across networks: 256 for large social networks like Orkut and Live Journal, 1024 for YouTube and Skitter, and 128 for road networks. Kokkos::AUTO, which automatically selects team size, yields similar results to a team size of 128. Table II compares non-hierarchical and hierarchical executions. Our findings indicate that social networks, with some high-degree nodes, gain from hierarchical parallelism. Conversely, bounded-degree graphs like road networks do not benefit from hierarchical parallelism due to insufficient work to offset the added thread layer overhead.

TABLE II: Effect of thread hierarchy in GPU computations using diverse domains of graph networks. (Hierarchical implementation uses Kokkos::AUTO as team thread size). Better speedups are highlighted.

Network	Non-Hierarchical (sec)	Hierarchical (sec)
Facebook	0.058755	**0.000973**
DBLP	0.008681	**0.001244**
YouTube	34.442723	**0.385194**
Skitter	148.80952	**0.591226**
LiveJournal	14.065877	**0.153994**
Orkut	168.381133	**0.802101**
Road-CA	**0.000196**	0.00435
Road-PA	**0.000131**	0.002539
Road-TX	**0.000146**	0.00322
Road-USA	**0.003106**	0.046229
Road-EU	**0.004185**	0.097537

C. Major Takeaways

Our findings recommend employing binary search over hash-based methods for triangle counting, showing up to a $2.66\times$ performance enhancement between sequential algorithms and up to $5.06\times$ between parallel algorithms. Additionally, dynamic scheduling outperforms static scheduling, yielding up to a $9.80\times$ improvement in CPU implementations. Utilizing GPUs with recommended configurations exhibits up to a $6.18\times$ performance boost compared to CPUs. Hierarchical parallelism proves superior to non-hierarchical parallelism in GPUs for power-law graphs, with up to a $306\times$ performance improvement observed. Optimal team size selection is also crucial for GPU implementations, showing up to a $6.66\times$ performance enhancement when the appropriate team size is utilized.

Acknowledgments. This work has been partially supported by National Science Foundation (NSF) under Award Number 2323533 and 2023 Sustainable Research Pathways (SRP) summer program at Berkeley Lab.

REFERENCES

[1] "Execution policies." [Online]. Available: https://kokkos.org/kokkos-core-wiki/API/core/Execution-Policies.html
[2] "Parallel kokkos construct." [Online]. Available: https://kokkos.org/kokkos-core-wiki/API/core/parallel-dispatch/parallel_for.htmlexamples
[3] "Teampolicy." [Online]. Available: https://kokkos.github.io/kokkos-core-wiki/API/core/policies/TeamPolicy.html
[4] "OpenMP sepcification," https://www.openmp.org/, 2023.
[5] "National energy research scientific computing center (nersc)," https://www.nersc.gov/, 2024.
[6] S. Arifuzzaman, M. Khan, and M. Marathe, "Patric: a parallel algorithm for counting triangles in massive networks," in *Proceedings of the 22nd ACM international conference on Information & Knowledge Management*, 2013, pp. 529–538.
[7] ——, "A fast parallel algorithm for counting triangles in graphs using dynamic load balancing," in *2015 IEEE International Conference on Big Data (Big Data)*. IEEE, 2015, pp. 1839–1847.
[8] ——, "Fast parallel algorithms for counting and listing triangles in big graphs," *ACM Transactions on Knowledge Discovery from Data (TKDD)*, vol. 14, no. 1, pp. 1–34, 2019.
[9] A. Azad, M. M. Aznaveh, S. Beamer, M. Blanco, J. Chen, L. D'Alessandro, R. Dathathri, T. Davis, K. Deweese, J. Firoz *et al.*, "Evaluation of graph analytics frameworks using the gap benchmark suite," in *2020 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2020, pp. 216–227.
[10] S. Beamer, K. Asanović, and D. Patterson, "The gap benchmark suite."
[11] G. Daiß, S. Y. Singanaboina, P. Diehl, H. Kaiser, and D. Pflüger, "From merging frameworks to merging stars: Experiences using hpx, kokkos and simd types," in *2022 IEEE/ACM 7th International Workshop on Extreme Scale Programming Models and Middleware (ESPM2)*. IEEE, 2022, pp. 10–19.
[12] T. A. Davis, "Graph algorithms via suitesparse: Graphblas: triangle counting and k-truss," in *2018 IEEE High Performance extreme Computing Conference (HPEC)*. IEEE, 2018, pp. 1–6.
[13] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, dec 2011. [Online]. Available: https://doi.org/10.1145/2049662.2049663
[14] A. S. Dufek, R. Gayatri, N. Mehta, D. Doerfler, B. Cook, Y. Ghadar, and C. DeTar, "Case study of using kokkos and sycl as performance-portable frameworks for milc-dslash benchmark on nvidia, amd and intel gpus," in *2021 International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. IEEE, 2021, pp. 57–67.
[15] K. Z. Ibrahim, C. Yang, and P. Maris, "Performance portability of sparse block diagonal matrix multiple vector multiplications on gpus," in *2022 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. IEEE, 2022, pp. 58–67.
[16] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, Jun. 2014.
[17] P. Pindl, "Performance portability for hpc applications through the raja abstraction layer," 2022.
[18] C. Trott, L. Berger-Vergiat, D. Poliakoff, S. Rajamanickam, D. Lebrun-Grandie, J. Madsen, N. Al Awar, M. Gligoric, G. Shipman, and G. Womeldorff, "The kokkos ecosystem: Comprehensive performance portability for high performance computing," *Computing in Science & Engineering*, vol. 23, no. 5, pp. 10–18, 2021.
[19] 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)*. IEEE, 2017, pp. 1–7.