ISSN 1995-0802, Lobachevskii Journal of Mathematics, 2020, Vol. 41, No. 8, pp. 1417–1426. © Pleiades Publishing, Ltd., 2020.

Developing Efficient Implementations of Connected Component Algorithms for NEC SX-Aurora TSUBASA

I. V. Afanasyev[1*] and Vl. V. Voevodin[1]

(Submitted by E. E. Tyrtyshnikov)

[1] *Research Computing Center of Moscow State University, Moscow, 119234 Russia*

Received March 13, 2020; revised April 2, 2020; accepted April 16, 2020

Abstract—Modern vector architectures are tend to be equipped with high-bandwidth memory, what makes them an interesting candidate for solving large-scale graph processing problems. However, highly irregular structure of real-world graphs makes it extremely challenging to map fundamental graph-processing problems on vector systems. This paper describes the world-first attempt, aimed to create efficient vector- friendly implementations of various connected components algorithms for modern NEC SX-Aurora TSUBASA architecture, which provides high performance computational power together with a world-highest bandwidth memory. In order to develop fast implementations, supercomputer co-design principles are used, including: the selection of vector-friendly graph algorithms, adapting these algorithms for target architecture, selecting vectorized graph storage format and applying various optimisations aimed to improve the efficiency of using memory hierarchy of target platform. In addition, current paper analyses if similar implementation approaches can be used for modern NVIDIA GPU architectures, which have many common properties and features with SX-Aurora TSUBASA. Finally, a comprehensive comparative performance analysis is presented for all algorithms, architectures and optimisations, discussed in the paper.

DOI: 10.1134/S1995080220080028

Keywords and phrases: *graph algorithms, NEC SX-Aurora TSUBASA, connected components, NVIDIA GPU, HPC, large-scale graph processing.*

1. INTRODUCTION

Developing efficient graph algorithms implementations is an extremely important problem of modern computer science, since graphs are used to model various real-world objects from different application areas. For example, graphs are used for social networks and web-graphs analysis, solving infrastructure and biological problems, social-economic modelling and many others. In all listed areas objects, represented with graphs, may consisting of millions and billions of vertices and edges. In order to accelerate computations and store such object it is necessary to use supercomputing for large-scale graph processing.

Two types of supercomputers are frequently used for solving graph problems: shared memory and distributed memory systems. Despite the fact that distributed memory systems are capable of processing much larger graphs, shared memory systems are generally significantly more efficient on a per core, per dollar, and per joule basis [1]. In addition, shared-memory systems can be used for solving many real-world problems [2]. Moreover, shared memory systems are often equipped with GPUs and other co-processors, providing even higher performance and energy efficiency values. Thus, current work mainly focuses on developing graph algorithms efficient implementations for shared memory systems.

However, not all shared memory systems are capable of solving graph processing problems equally fast and efficiently. Graph algorithms usually belong to the data-intensive class of programs, which heavily stress memory subsystem of target platforms. As a result, systems with high-bandwidth memory allow to significantly accelerate graph computations. At the moment of this writing high-bandwidth

*E-mail: afanasiev_ilya@icloud.com

memory is installed mainly in dedicated vector architectures (for example NEC SX-architectures), systems with vector extensions (AVX-512, AltiVec instruction sets) and NVIDIA GPUs which also rely on vector computing principles (warps). vectorized data-processing allows these architectures to utilise high memory throughput due to collective accesses to memory subsystem.

Approaches, which allow to develop efficient graph algorithms implementations for modern vector systems are poorly studied, since due to the highly irregular structure of real-world graphs it is extremely challenging to use vector processing features efficiently. Current paper proposes multiple efficient implementations of connected components algorithms for modern vector systems with high-bandwidth memory. NEC SX-Aurora TSUBASA vector architecture, which at the moment of this writing has world's fastest memory, is used as the main target platform. However, the same approaches can be used for developing and optimising graph algorithms for other vector architectures, for example, Intel KNL processors or NVIDIA GPUs.

2. NEC SX-AURORA TSUBASA ARCHITECTURE

NEC SX-Aurora TSUBASA is the latest SX vector supercomputer with dedicated vector processors [3, 4]. SX-Aurora TSUBASA inherits the design concepts of the vector supercomputer and enhances its advantages to achieve higher sustained performance and higher usability. Different from its predecessors in the SX supercomputer series [5, 6], the system architecture of SX-Aurora TSUBASA mainly consists of vector engines (VEs), equipped with a vector processor and a vector host (VH) of an x86 node. The VE is used as a primary processor for executing applications while the VH is used as a secondary processor for executing basic operating system (OS) functions that are offloaded from the VE. The VE has eight powerful vector cores. As each core provides 537.6 GFlop/s of single-precision performance with 1.40 GHz frequency, the peak performance of the VE reaches 4.3 TFlop/s.

Each SX-Aurora vector core consists of three components: scalar processing unit (SPU), vector processing unit (VPU), and memory subsystem. Most computations are performed by VPUs, while SPUs provide functionality of typical CPU. Since SX-Aurora is not just a typical accelerator, but rather a self-sufficient processor, SPUs are designed to provide relatively high performance on scalar computations. VPU of each vector core has its own relatively simple instruction pipeline aimed for decoding and reordering vector instructions incoming from SPU. Decoded instructions are executed on vector-parallel pipelines (VPP). In order to store the results of intermediate calculations, each vector core is equipped with 64 vector registers with a total register capacity equal to 128 KB. Each register is designed to store a vector of 256 double precision elements (DP). On the memory subsystem side, six HBM modules in the vector processor can deliver the 1.22 TB/s world's highest memory bandwidth [5]. This high memory bandwidth contributes to higher sustained performance, especially in memory-bound applications.

3. INPUT GRAPHS

This paper presents a comprehensive comparative performance and efficiency analysis of the developed implementations using synthetic and real-world graphs with different characteristics. Synthetic graphs allow to easily scale various input graphs parameters (such as graph size), while real-world graphs allow to more accurately evaluate the performance on real problems. Synthetic RMAT [7] and uniform-random [8] graphs are used in this work, while real-world graphs are taken from KONEKT [9] and SNAP [10] collections. Table 1 demonstrate main properties of the graphs used in this paper.

Graph size is determined by the total amount of its vertices and edges. The total number of edges determines whether it is possible to process the graph using a single SX-Aurora Vector Engine or an NVIDIA GPU card. The total number of vertices and vertex connectivity degree distribution determines the possibility of using cache memory in order to store indirectly accessed arrays. Graph diameter determines the performance of algorithms, based on iterative graph traversing: important examples of such algorithms include BFS and shortest paths.

Table 1. Properties and main characteristics of graphs used for the performance evaluation

Name	Type	Number of veritces	Number of edges	Graph size (in CSR format)	Size of indirectly accessed arrays	Vertex-degree distribution	Number of non-trivial connected components	Maximum size of connected component
rmat_20_16	Synthetic	$(2^{20}) = 1$m	$(2^{25}) = 33$m	271 MB	4 MB	Power-low	510	550 390 (52%)
rmat_26_16	Synthetic	$(2^{26}) = 67$m	$(2^{31}) = -$m	– MB	– MB	Power-low	14	550 390 (52%)
ru_20_16	Synthetic	$(2^{20}) = 1$m	$(2^{25}) = 33$m	280 MB	4 MB	Uniform	1	1 048 576 (100%)
ru_24_16	Synthetic	$(2^{24}) = 16$m	$(2^{29}) = 536$m	4 GB	67 MB	Uniform	1	16 777 216 (100%)
Livejournal	Social	5.2m	49m	450 MB	20 MB	Power-low	1	3 306 800 (63%)
Orkut	Social	3m	117m	900 MB	20 MB	Power-low	1	3 072 441 (100%)
Pokec	Social	1.6m	30m	260 MB	6 MB	Power-low	7434	1 427 459 (87%)
Flickr	Social	2.3m	33m	290 MB	9.2 MB	Power-low	64033	1 692 185 (73%)
Youtube	Social	3.2m	9.3m	110 MB	12 MB	Power-low	4	1 522 935 (47%)
Wikipedia_en	Web	12m	378m	3.1 GB	28 MB	Power-low	678	12 114 017 (99%)
Web_trackers	Web	27m	140m	1.4 GB	110 MB	Power-low	1	13 892 605 (50%)
Baidu	Web	2.1m	17.6m	160 MB	8.5 MB	Power-low	41896	1 600 909 (74%)
Wikipedia_ru	Web	2.8m	82m	690 MB	11 MB	Power-low	130	2 852 338 (99%)
RoadNet_CA	Infrastructure	1.9m	5.5m	60 MB	7 MB	Uniform	2638	1 957 027 (99%)
Us_patents	Other	3.7m	16m	170 MB	15 MB	Uniform	91014	2 085 264 (55%)

4. DESCRIPTION OF ALGORITHMS AND STATE OF THE ART

4.1. Connected Components Problem Description

An undirected non-weighted graph $G(V, E)$ is given, where is V the set of vertices, and E is the set of edges. The following notation will be used further in the paper: $|V|$ is the total number of vertices, $|E|$ is the total number of edges in G. The edge $(u, v) \in E$ connects vertices u and v. A path from vertex s to an arbitrary vertex $v \in V$ in a graph G is a finite sequence of edges and vertices $S(s, v) = (a_0, E_0, a_1, E_1, ..., a_{n-1}, E_{n-1}, a_n)$, such that each two adjacent edges E_i and E_{i-1} have a common vertex a_I, and $s = a_0, v = a_n$. A connected component of an undirected graph $G(V, E)$ is a subgraph $G'(V', E')$ in which any two vertices $u, v \in V'$, are connected to each other by path $S(u, v)$.

4.2. Existing Algorithms Overview

Various algorithms for searching connected components have been proposed. First group include BFS-based and DFS-based algorithms [11], which utilize breadth-first and depth-first searches respectively. These algorithms execute a breadth(depth)-first search from each vertex v of graph G, which does not have an assigned component yet. All vertices reached during the search are marked with the same number of connected component. This process is repeated until all graph vertices are labelled with connected components. Both algorithms have $O(|V|)$ complexity, since each graph vertex is visited exactly once.

Shiloach–Vishkin algorithm [12] is based on calculating trees, which correspond to different connected components using " hook" and "jump" operations, such that at the end of the algorithm vertices in the same component will belong to the same tree. The proposed implementation of algorithm allow to avoid the occurrence of cycles, thus algorithm always terminates after $O(\log |V|)$ steps, each of which has a maximum $O(|E|)$ complexity.

Random-mate algorithm [13] is based on the process of randomly assigning each vertex "child" or "parent" statuses. After that, "child" nodes are added to "parent" ones, forming new vertices. With high probability, this algorithm terminates after $O(\log |V|)$ steps.

Several hybrid algorithms, for example Awerbuch–Shiloach [11] also exist based on the previous ones.

4.3. Existing Implementations Overview

At the time of writing, no known attempts of developing vectorized implementations of any connected components for NEC SX-Aurora TSUBASA architecture have been made. Research [14] describes implementations of several graph algorithms, including strongly connected components, for previous SX-ACE generation of the NEC vector architecture. However, only comparable per-socket and slightly better per-core performance has been achieved compared to Intel Skylake processors. Several other graphs algorithms implementations (page rank, shortest paths) have been proposed for NEC SX-Aurora TSUBASA architecture [15]. Optimisations techniques described in this research, such as improving gather and scatter accesses for power-low graphs, utilising vectorized graph storage format and balancing vectorized workloads can be effectively used during connected components algorithms implementation.

As demonstrated in [16], NEC SX-Aurora TSUBASA and NVIDIA GPU architectures have many common computational features, which allow to use similar optimisation approaches for both architectures. Similarities and differences of NEC SX-Aurora TSUBASA and NVIDIA GPU computational models is demonstrated on Fig. 1 (right). Multiple implementations of connected components algorithms have been proposed for NVIDIA GPU architecture [17, 18]. Additionally, connected components implementation is included in Gunrock [20] framework.

Finally, connected components implementations are present in multiple multi-core CPU graph processing frameworks and libraries, including Ligra [1] and Gap Benchmark Suite [21]. However, these multi-core CPU implementations can not be executed efficiently on vector platform like SX-Aurora TSUBASA, but can be also used for comparative performance evaluation.

5. CODESIGN OF VECTOR-FRIENDLY IMPLEMENTATIONS

In order to develop fast implementations, supercomputer co-design principles are used, which include: the selection of vector-friendly graph algorithms, adapting these algorithms for target architecture, selecting vectorized graph storage format and applying various optimisations aimed to improve the efficiency of using memory hierarchy of target platform.

5.1. Algorithm Properties Analysis and Selection

Many graph algorithms can not be effectively vectorized, since SIMD computation model imposes significant restrictions on the control flow of the program. Proposes co-design approach helps to chose suitable for vectorization algorithms based on the analysis of:

- fundamental properties of algorithms (e.g. sequential and parallel complexity, computing power),
- properties and sample templates of information graphs of algorithms [19].

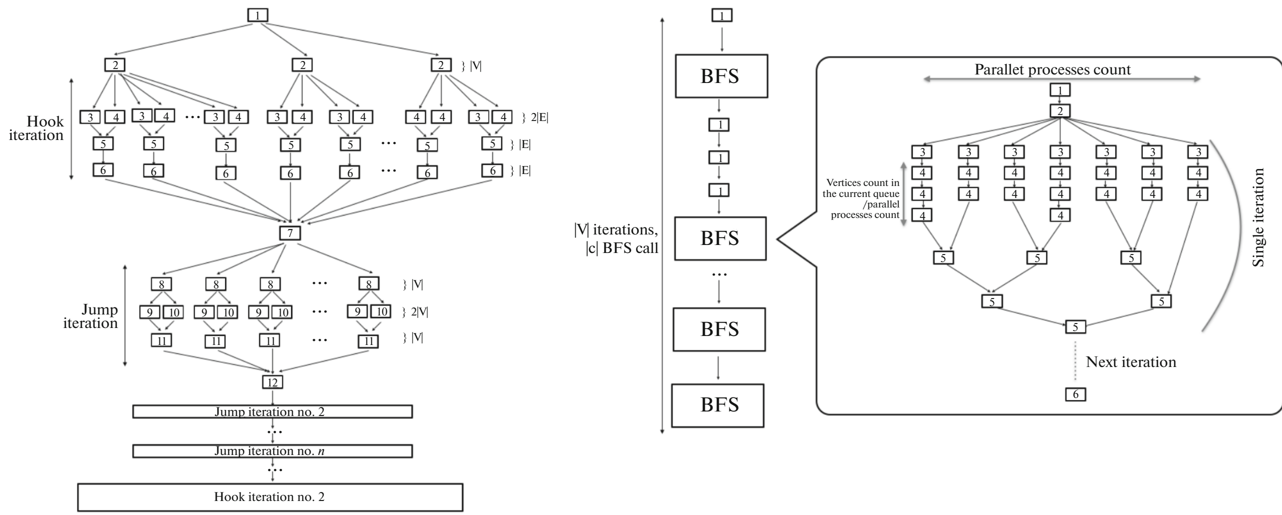

Fig. 1. Informational graphs of Shiloach–Vishkin (left) and BFS-based (right) algorithms.

Since many modern vector architectures are massively-parallel, the ratio of serial to parallel complexity should be as large as possible in order to efficiently provide enough work for all the computing devices of target architecture. The computational power of the algorithm (equal to the ratio of the operations executed to the total amount of input and output data) is also important, since target architectures (NEC SX-Aurora TSUBASA and NVIDIA GPU) are installed as coprocessors, and moving data through PCI or NVLINK bus can quickly become the bottleneck of algorithms with low computing power. General properties of informational graphs (for example, the width of a tiered-parallel form), as well as typical patterns, which allow or prevent efficient vectorization, are also need to be carefully analysed. In the remaining part of the section we will analyse properties of three main connected components algorithms.

5.1.1. Shiloach–Vishkin algorithm. Informational graph of Shiloach–Vishkin algorithm is demonstrated on Fig. 1 (left). The sequential complexity of the algorithm is equal to $|E| * O(\log |V|)$, while parallel complexity—to $O(\log |V|)$. The computational power of the algorithm is equal to $O(\log |V|)$. The width of the parallel form is equal to $O(|E|)$ or $O(|V|)$ depending on the type of iteration (hook or jump), with the same operations being performed on each level, which allows efficient vectorization. At the same time, all levels of the informational graph contain a sufficient amount of operations, and there are no indirect data dependencies between the levels.

5.1.2. BFS-based algorithm. Figure 1 (right) presents informational graph of BFS-based algorithm, which uses top-down BFS as a subroutine. Sequential complexity of this algorithm is equal to $O(|E|)$, while parallel complexity highly depends on the structure of input graph and is equal to $O(d * c)$, where d is graph diameter, c is the number of connected components. Computational power of BFS-based algorithm is equal to 1. The width of layered-parallel form of this algorithm also highly depends on the structure of input graph, since it is equal to the frontiers sizes inside BFS algorithm. Informational graph of BFS algorithm demonstrates, that the algorithm can be efficenlty vectorized (each parallel layer contains sufficient amount of similar independent operations). However, since algorithms contains a large sequential part, when BFS operations are launched from different graphs source vertices one by one, this algorithms may have a large unvectorizable part if many connected components are present in graphs.

5.1.3. Random-mate algorithm. Informational graph of Random-mate algorithm is demonstrated on Fig. 2 (left). It has many similarities to informational graph of Shiloach–Vishkin algorithm, since on each iteration it also traverses all graph edges. However, after each iteration Random-mate algorithm requires creating new representation of input graph, since different vertices have to be merged into supervertices. This subroutine is highly depends from graph storage format, and for many vector-friendly graph storage format can not be vectorized efficiently. In addition, this part of algorithms requires at least $O(|E|)$ operations, which is comparable to the first algorithm part.

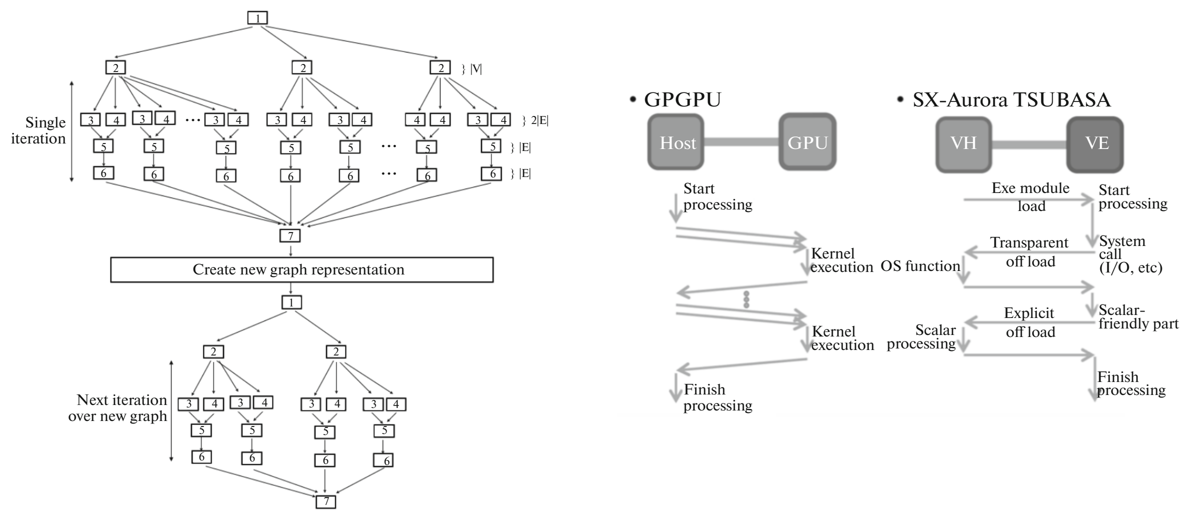

Fig. 2. Informational graphs of Random-mate algorithm (left), differences between execution models of NEC SX-Aurora TSUBASA and NVIDIA GPU (right).

Thus, all three reviewed algorithms can be implemented on vector architectures to some extent. However, Shiloach–Vishkin and BFS-based algorithms are more suitable for vectorization, since they lack large sequential parts, which quickly can become a significant bottleneck. Therefore, it is important to develop efficient vector-friendly implementations of Shiloach–Vishkin and BFS-based algorithms, and to conduct a comprehensive comparative performance analysis for various algorithms and types of input graphs.

5.2. Graph Storage Format

Graph storage format largely determines memory access patterns of graph application, as well as the principles of parallel workload balancing between different computational units (different cores and vector units). Memory access patterns for NEC SX-Aurora TSUBASA architecture ideally should have the following structure: different vector cores should access independent non-overlapping regions of memory, while memory access of each vector core should have sequential (or at least localized) pattern within the same vector instruction. Ideal memory access patterns for NVIDIA GPU architecture are very similar: data accessed from threads of the same warp should be sequential or at least localized, while accesses from different warps don't have any strict requirements.

CSR graph storage format allows to efficiently store information about the adjacency of vertices in graph, which is required for many graph algorithms (for example BFS). However, a straightforward implementation of CSR graph storage format on vector architectures in many cases causes additional indirect memory accesses on the stage of loading graph edges data. Since these accesses significantly decrease the efficiency of vector computations, it is necessary to develop and use vector extensions of CSR storage format. On the other hand, existing specialised vector formats for sparse-matrix operations (COO, SELLPACK for GPU, SELLPACK for KNC) don't support efficient graph traversals for partial-active graph algorithms, when only a small part of graph elements (selected vertices or edges) have to be loaded from memory and processed, leading to significant decrease of effective bandwidth for sparse matrix-based graph algorithms.

A vector extension to the CSR storage format for NEC SX-Aurora TSUBASA architecture has been proposed in [15]. The proposed format is based on preliminary sorting of graph vertices with a key equal to the average degree of each vertex of the graph. This allows to collectively process vertices with low connectivity degree in portions of 256 vertices, which is equal to the length of SX-Aurora vector instruction. This allows to use optimal memory access patterns for vector architectures and GPUs when using CSR vector extension. However, this format has a significant drawback: it can not effectively support algorithms, which require traversing only a subset of graph vertices (partial-active graph algorithms), since the format forces either implementing an ineffective memory access pattern, or loading a large amount of redundant edges data from memory.

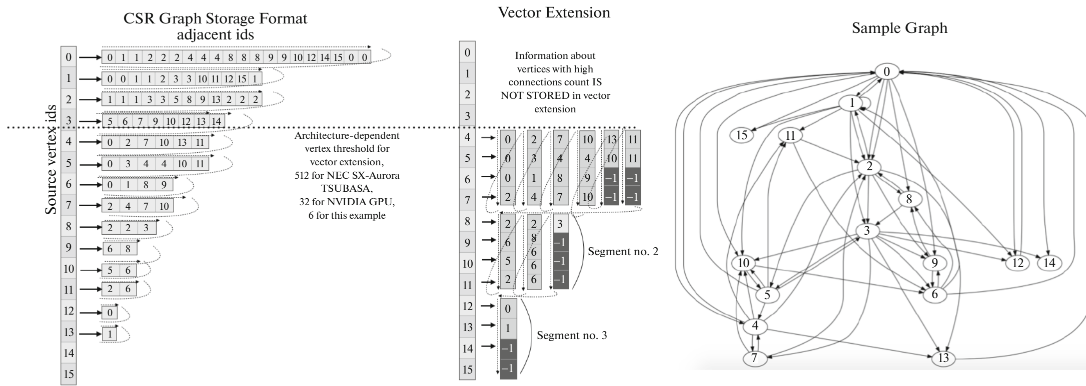

Fig. 3. Sample graph (right) and its CSR graph storage format with vector extension (left).

Since algorithms implemented in this paper belong both to all-active (Shilloah–Vishkin) and partial-active (BFS-based) groups of algorithms, it is advantageous to use a combination of traditional CSR format and the proposed vector extension. This way an entire graph is initially stored in traditional CSR format, with vertices being pre-sorted in descending order of connectivity degree. Starting from a certain degree threshold (512 for SX-Aurora TSUBASA, 32 for NVIDIA GPU), vertices with small degree connectivity degree are also additionally stored inside the vector extension. This way CSR vector extension is used in the case when algorithm iteration requires to traverse either all or a significant part of graph vertices, while the traditional CSR storage format is used in the sparse cases, when only a small number of vertices has to be traversed. Figure 3 demonstrates the final scheme of formats, used in the current paper.

The main disadvantage of combining two formats is the requirement to store a significant amount of additional data. However, for many real-world power-law graphs, the amount of additional data stored is not large, since a large part of graph edges is concentrated at vertices with a high connectivity degree, which are not stored inside vector extension.

5.3. Optimisation and Implementation Details

Graph algorithm optimisations used in this paper can be divided into four main groups:

- optimising memory access pattern when loading information about graph edges,
- increasing the locality of accesses to vertices arrays (indirect memory accesses),
- load balancing between different vector cores and elements of vector instructions,
- optimising the number of traversed graph vertices and edges.

Accesses to the arrays with information about graph edges are implemented efficiently based on using CSR vector extension, described in the previous section. In addition the proposed format is based on sorting graph vertices based on connectivity degree, what allows to increase spatial locality of accesses to per-vertex data for power-law graphs. Efficient parallel workload balancing is implemented by splitting graph vertices into three separate groups based on their connectivity degree. Each vertex from the first group (with high connectivity degree) is processed using all vector cores, vertices from the second group (with medium connectivity degree)—using one vector core, while vertices from the third group (with low connectivity degree) are collectively processed in portions of 256. A work distribution between different vector cores inside each group of vertices is implemented using OpenMP static(8) loop iterations distribution mode.

Shiloah–Vishkin belongs to the group of iterative all-active graph algorithms, which traverse all graph vertices and edges at each iteration using CSR format vector extension. BFS-based algorithm uses optimised version of direction-optimising BFS [21], which belongs to the class of iterative partial-active graph algorithms, when on each iteration a frontier of active vertices exists. The frontier of active

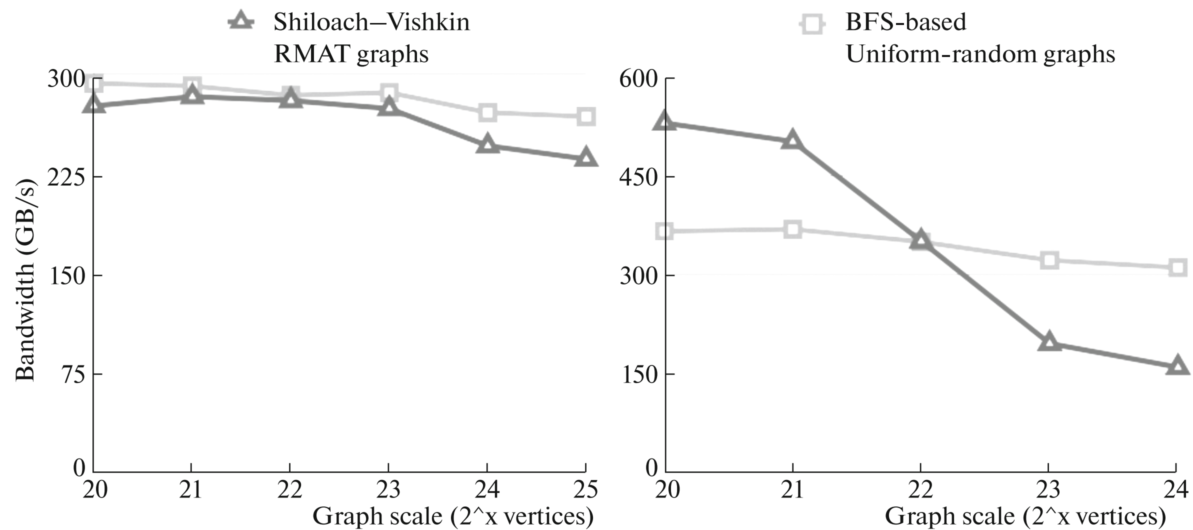

Fig. 4. Bandwidth (in GB/s), achieved by the developed implementations for NEC SX-Aurora TSUBASA architecture.

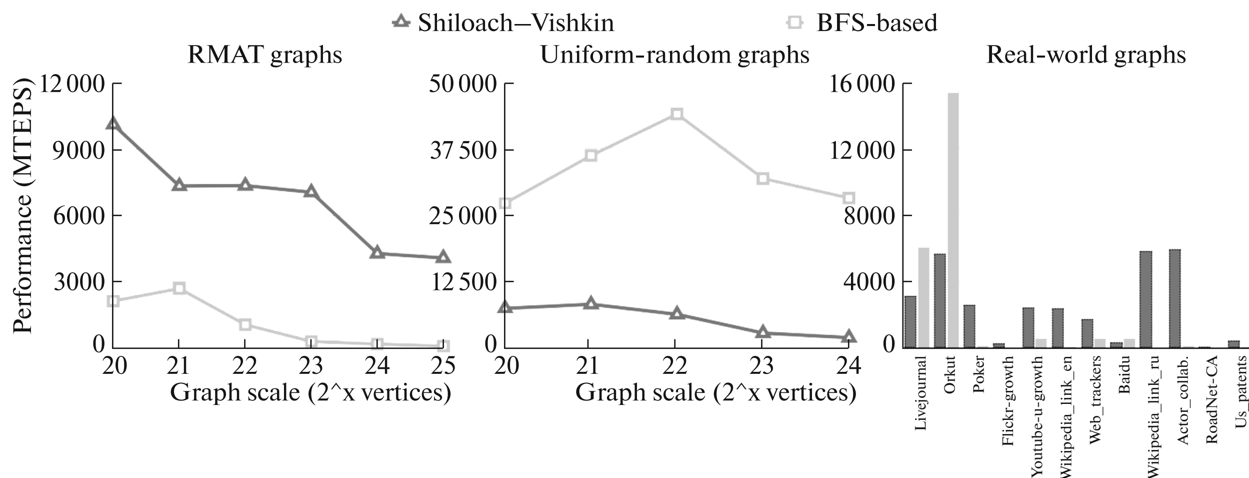

Fig. 5. Performance (in MTEPS) comparison for different types of input graphs and developed algorithms on NEC SX-Aurora TSUBASA architecture.

vertices can be either dense (implemented via presences flags) and sparse (implemented via queue). Switching from dense to sparse frontier is implemented via optimised parallel prefix sum operation, while the decision if the switch is required is implemented by estimating the percentage of active vertices based on the reduction operation, which can be implemented on SX-Aurora TSUBASA architecture very efficiently [21]. On each iteration of BFS algorithms sparse frontier operates with graph in traditional CSR format, while dense frontier operates with CSR vector extension.

The same optimisations with only minor changes can be applied for other vector architectures, including NVIDIA GPU. For example, vector length for the vector extension must be set to 32 (equal to GPU warp size), and thresholds together with principle of parallel workload balancing must be changed based on CUDA block and warp size.

6. PERFORMANCE EVALUATION

The overall effect from the optimisations proposed in the previous section can be estimated by calculating the effective bandwidth of the developed implementations, and then comparing it to the peak bandwidth values for target architecture. When working with 4-byte indirectly accessed data, peak

bandwidth of NEC SX-Aurora TSUBASA architecture is equal to approximately 600 GB/s, since it is effectively halved based on the implementation of gather and scatter instructions. As demonstrated on Fig. 4, the developed implementations achieve 300–550 GB/s: the bandwidth is higher for more regular (in terms of degree distribution) small-sized and medium-sized uniform-random graphs, and is lower for less-regular RMAT graphs, where load-balancing and non-uniform memory accesses to per-vertex arrays significantly bottleneck the performance.

Figure 5 present the performance comparison of the developed connected components algorithms for SX-Aurora TSUBASA on various types of input graphs. All the performance values are measured in TEPS (Traversed Edges Per Second), which can be defined as the number of edges in a graph, divided by the execution time of the implementation.

Based on the presented performance comparison, the following conclusions can be made. First, BFS-based algorithm demonstrates significantly higher performance on graphs, which have only few non-trivial connected components (especially uniform-random graphs). In addition, BFS-based algorithm scales significantly better with the increase of graph size for graphs with uniform degree distribution, since it is less dependent on the locality of indirect accesses to per-vertex data arrays. However, for the many real-world graphs, which have many non-trivial connected components 1, BFS-based algorithm demonstrates significantly lower performance due to the sequential non-vectorized nature of launching BFS subroutines.

7. CONCLUSION

In this paper efficient vector-friendly implementations of two fundamental graph algorithms for connected component computation have been proposed for NEC SX-Aurora TSUBASA architecture. Optimisation techniques applied aimed to maximize the performance include using a specialised vector-friendly graph format, efficient workload-balancing between different vector cores and elements of vector instructions, improvement of memory access patter during edge traversals and locality during the accesses to indirectly accessed arrays. The conducted comparative performance analysis demonstrates that each algorithm is capable of processing different types graph more efficiently: for example, BFS-based algorithm works significantly faster for uniform-random graphs, while Shiloach–Vishkin is better suited for RMAT and many real-world graphs with multiple non-trivial connected components.

FUNDING

The reported study was funded by RFBR, project number 19-37-90002. The research is carried out using the equipment of the shared research facilities of HPC computing resources at Lomonosov Moscow State University.

REFERENCES

1. J. Shun and G. E. Blelloch, "Ligra: A lightweight graph processing framework for shared memory," ACM Sigplan Notices, 135–136 (2013).
2. S. Beamer, K. Asanovic, and D. Patterson, "Locality exists in graph processing: Workload characterization on an ivy bridge server," in *Proceedings of the 2015 IEEE International Symposium on Workload Characterization, 2015,* pp. 56–65.
3. K. Komatsu, S. Momose, Y. Isobe, O. Watanabe, A. Musa, M. Yokokawa, T. Aoyama, M. Sato, and H. Kobayashi, "Performance evaluation of a vector supercomputer SX-aurora TSUBASA," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, 2018,* pp. 54:1–54:12.
4. Y. Yamada and S. Momose, "Vector engine processor of NEC's brand-new supercomputer SX-Aurora TSUBASA," in *Proceedings of the International Symposium on High Performance Chips* (2018).
5. R. Egawa, K. Komatsu, S. Momose, Y. Isobe, A. Musa, H. Takizawa, and H. Kobayashi, "Potential of a modern vector supercomputer for practical applications: Performance evaluation of SX-ACE," J. Supercomput. **73**, 3948–3976 (2017).
6. K. Komatsu, R. Egawa, Y. Isobe, R. Ogata, H. Takizawa, and H. Kobayashi, "An approach to the highest efficiency of the HPCG benchmark on the SX-ACE supercompute," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis* (2015), pp. 1–2.

7. D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-MAT: A recursive model for graph mining," in *Proceedings of the 2004 SIAM International Conference on Data Mining* (2004), pp. 442–446.
8. P. Erdos and A. Renyi, "On random graphs I," Publ. I. Math. Debrecen, No. 6, 18 (1959).
9. The Koblenz Network Collection—KONECT. http://konect.uni-koblenz.de.
10. Stanford Large Network Dataset Collection—SNAP. https://snap.stanford.edu/data/.
11. J. Greiner, "A comparison of data-parallel algorithms for connected components," in *Proceedings Symposium on Parallel Algorithms and Architectures, 1994,* pp. 16–25.
12. Y. Shiloach and U. Vishkin, "An O(log(n)) parallel connectivity algorithm," CS Tech. Report CS0178 (1980).
13. H. Gazit,"An optimal randomized parallel algorithm for finding connected components in a graph," SIAM J. Comput. **20**, 1046 (1991). https://doi.org/10.1137/0220066
14. I. V. Afanasyev, A. S. Antonov, D. A. Nikitenko, V. V. Voevodin, V. V. Voevodin, K. Komatsu, O. Watanabe, A. Musa, and H. Kobayashi, "Developing efficient implementations of Bellman-Ford and forward-backward graph algorithms for NEC SX-ACE," Supercomput. Front. Innov., No. 5, 65 (2018). https://doi.org/10.14529/jsfi180311
15. I. Afanasyev, V. V. Voevodin, V. V. Voevodin, K. Komatsu, and H. Kobayashi, "Developing efficient implementations of shortest paths and page rank algorithms for NEC SX-aurora TSUBASA architecture," Lobachevskii J. Math. **40** (11), 1753–1762 (2019). https://doi.org/10.1134/S1995080219110039
16. I. V. Afanasyev, V. V. Voevodin, V. V. Voevodin, K. Komatsu, and H. Kobayashi, "Analysis of relationship between simd-processing features used in nvidia gpus and nec sx-aurora tsubasa vector processors," in *Proceedings of the International Conference on Parallel Computing Technologies* (Springer, 2019), pp. 125–139. https://doi.org/10.1007/978-3-030-25636-4_10
17. J. Soman, K. Kothapalli, and P. Narayanan, "Some GPU algorithms for graph connected components and spanning tree," Parallel Process. Lett. **20**, 325–339 (2010).
18. I. Y. Jungand and C. S. Jeong, "Parallel connected-component labeling algorithm for GPGPU applications," in *Proceedings of the 10th International Symposium on Communications and Information Technologies, 2010,* pp. 1149–1153.
19. V. Voevodin, *Parallel Computing* (BHV-Peterburg, St. Petersburg, 2002) [in Russian].
20. Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. Owens, "Gunrock: A high-performance graph processing library on the GPU," in *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2016,* pp. 1–12.
21. S. Beamer, K. Asanovic, and D. Patterson, "The GAP benchmark suite," arXiv: 1508.03619 (2015).