DOI: 10.14529/jsfi200404

Developing an Architecture-independent Graph Framework for Modern Vector Processors and NVIDIA GPUs

Ilya V. Afanasyev[1]

This paper describes the first-in-the-world attempt to develop an architectural-independent graph framework named VGL, designed for different modern architectures with high-bandwidth memory. Currently VGL supports two classes of architectures: NEC SX-Aurora TSUBASA vector processors and NVIDIA GPUs. However, VGL can be easily extended to other architectures due to its flexible software structure. VGL is designed to provide users with the possibility of selecting the most suitable architecture for solving a specific graph problem on a given input data, which, in return, allows to significantly outperform existing frameworks and libraries, developed for modern multicore CPUs and NVIDIA GPUs. Since VGL uses an identical set of computational and data abstractions for all architectures, its users can easily port graph algorithms between different target architectures without any source code modifications. Additionally, in this paper we show how graph algorithms should be implemented and optimised for NVIDIA GPU and NEC SX-Aurora TSUBASA architectures, demonstrating that both architectures have multiple similar properties and hardware features.

Keywords: vector computers, NVIDIA GPUs, graph algorithms, graph framework, VGL, CUDA, optimisation.

Introduction

Developing efficient implementations of graph algorithms is an extremely important problem of modern computer science, since graphs are heavily used in many applications fields: social networks and web graphs analysis, navigation, solving infrastructural problems, and many others. Supercomputing architectures with high-bandwidth memory (HBM) are able to significantly speed up solving various graph problems, which belong to the data-intensive class and thus potentially benefit from faster memory hierarchy. Nowadays, high-bandwidth memory is installed either into GPUs or systems with vector processing features (vector processors or CPUs with vector extensions). Efficiently implementing graph algorithms on systems with high-bandwidth memory is difficult, since implementation approaches significantly differ from those used for traditional multicore CPUs, mainly because these systems utilize SIMD-processing (Single Instruction Multiple Data) features.

Various graph libraries and frameworks have been developed for various modern architectures, mainly multicore CPUs and NVIDIA GPUs. Graph libraries usually provide highly-optimised implementations of several fundamental graph algorithms, while graph frameworks typically include optimised computational and data abstractions, which can be used to easily express different graph algorithms variations. However, existing graph-libraries and frameworks have the following drawbacks:

1. none of the existing frameworks and libraries support efficient graph processing on vector systems (such as NEC SX-Aurora TSUBASA);
2. existing frameworks typically target only a specific architecture, forcing its users to completely rework the implementation when using a different architecture is required; and
3. existing frameworks in many cases can be further optimized for their target architectures (including NVIDIA GPUs).

[1]Moscow Center of Fundamental and Applied Mathematics, Moscow, Russia

To approach the first problem we have previously developed a VGL (Vector Graph Library)[2] [2, 5] framework for the NEC SX-Aurora TSUBASA vector architecture. VGL significantly outperforms many existing graph-processing frameworks, developed for modern multicore CPUs and NVIDIA GPUs.

As shown in [3], NVIDIA GPUs and NEC SX Aurora TSUBASA vector architecture have many common hardware features. This means that **many graph algorithms can be implemented on these architectures with similar optimisation and implementation approaches**. However, at the moment of this writing no research has been carried out to confirm (or refute) this thesis. In order to approach this problem, we ported our VGL framework (originally developed for NEC SX-Aurora TSUBASA vector architecture) to the latest NVIDIA GPUs, what allowed us to compare implementation and optimisation approaches, which should be used for both architectures.

As a result we have developed a first-in-the-world architecture independent framework, which simultaneously targets multiple architectures with high-bandwidth memory: modern NEC SX-Aurora TSUBASA vector system and NVIDIA GPUs. This is achieved by using a unified set of computational and data abstractions, identical for all architectures. Moreover, our framework has flexible software structures, which allow to easily extend it to different architectures (for example multicore CPUs). This property of VGL allows its users to select the most suitable architecture for solving a specific graph problem on a given input data, thus solving it considerably faster compared to the existing frameworks and libraries, developed for a single particular architecture.

The article is organized as follows. Section 1 describes primary target architectures of the VGL framework: NEC SX-Aurora TSUBASA vector processors and modern NVIDIA GPUs. Section 2 describes existing state-of-the-art frameworks, developed for modern NVIDIA GPUs and multicore CPUs. In addition, Section 2 provides the description of computational and data abstractions, used in the VGL framework. Section 3 describes program structure of the VGL framework, which allows it to operate on different architectures, and thus to easily port graph algorithms implementations between them. Section 4 provides a detailed comparison of implementation and optimisation approaches, which are used to implement VGL computational abstractions on NEC SX-Aurora TSUBASA and NVIDIA GPU architectures. In particular, effects of different optimisations is compared for these two architectures. Section 5 evaluates the performance of multiple graph algorithms implemented via VGL framework, as well as VGL performance against existing graph-processing frameworks for other architectures. Conclusion summarizes the study and points directions for further work.

1. Target Architectures Overview

1.1. NEC SX-Aurora TSUBASA

The NEC SX-Aurora TSUBASA vector architecture [15, 20] consists of multiple vector engines (VE), installed into a vector host (VH), which is a typical x86 node. Vector engines are used as a primary processors for executing vectorised applications, while vector host is used as a secondary processor for executing basic operating system (OS) functions, as well as some scalar computations offloaded from the VE. The VE has eight powerful vector cores, each one operating with vector instructions of 256 length. Each vector core consists of two computational

[2]VGL is available for free download at vgl.parallel.ru

components: a scalar processing unit (SPU) and a vector processing unit (VPU). All vector computations are performed by VPUs, while SPUs are designed to provide a relatively high performance on scalar computations, without the need to explicitly offload them to the vector host (thus significantly reducing the amount of transfers through interconnect). In the following subsections the most important hardware characteristics of the NEC SX-Aurora TSUBASA architecture will be provided, related to graph processing.

1.2. NVIDIA GPU

NVIDIA GPUs [14] are also installed into the system as coprocessors, similar to SX-Aurora vector engines. Modern NVIDIA GPUs have thousands of CUDA cores, which are grouped into streaming multiprocessors (SM). Streaming multiprocessors execute instructions based on the warp concept: a group of 32 threads running on CUDA cores perform exactly the same instruction at every given moment of time. This computing model has many common features with vector computing, since they both belong to SIMD [9] class. This particular feature determines the fact that various graph algorithms may potentially be implemented similarly on these two classes of architectures. Further in the paper warps and vector instructions will be sometimes refereed as "SIMD instructions". In this paper two most recent GPUs of Tesla family are used: V100 and A100, hardware characteristics of which will be also provided in the following subsection and simultaneously compared with vector engines.

1.3. Hardware Characteristics Comparison

The main hardware characteristics of the two latest generations of SX-Aurora Vector Engines and NVIDIA GPUs are listed in Tab. 1. These hardware characteristics most noticeably affect the performance of graph processing. For example, peak memory bandwidth determines how fast information about graph edges can be loaded from memory, while the memory capacity determines graph of which size can be processed using GPU or VE, etc.

Table 1. The comparison between main hardware characteristics of modern NVIDIA GPUs and NEC SX-Aurora TSUBASA vector engines

Hardware Characteristic	**NEC SX-Aurora TSUBASA (1st generation)**	**NEC SX-Aurora TSUBASA (2nd generation)**	**NVIDIA V100 GPU**	**NVIDIA A100 GPU**
Peak memory bandwidth	1200 GB/s	1500 GB/s	900 GB/s	1500 GB/s
Memory capacity	48 GB	48 GB	16-32 GB	40-80 GB
LLC size	16 MB	16 MB	6 MB	40 MB
Prefetching support	yes	yes	no	yes
LLC bandwidth	3000 GB/s	3000 GB/s	N/a	N/a
SIMD size	256	256	32	32
Cores number	8	8	5120	6912
Interconnect bandwidth	up to 32 GB/s	up to 32 GB/s	up to 300 GB/s	up to 600 GB/s

Based on the provided in Tab. 1 information, the following conclusions can be made. The first generation of SX-Aurora vector engines have comparable characteristics to V100 GPUs, while the second generation – to A100 GPUs. However, several differences exist: for example, GPUs typically have interconnect with higher bandwidth,which allows to copy input graphs into GPU memory much faster. At the same time vector engines have significantly less resource of inner parallelism: they use only 8 cores, each one operating with vectors of 256 length, while modern GPU have thousands of cores, which require even more threads running in order to efficiently hide memory latency. This potentially allows vector engines to process small-sized and medium-sized graphs more efficiently compared to GPUs.

2. State of the Art

2.1. Existing Graph-Processing Frameworks

Several graph libraries and frameworks have been recently developed for modern multicore CPUs and NVIDIA GPUs. Ligra [18], Galois [17] and GAPBS [6] are the most well-known examples of multicore CPUs frameworks and libraries, while Gunrock [19] CuSHA [13], Medusa [22], and Enterprise [16] frameworks and libraries target modern NVIDIA GPUs. However, the following factors determine the relevance of developing VGL framework:

- none of the existing frameworks target modern vector systems, such as NEC SX-Aurora TSUBASA;
- none of the existing frameworks are capable of operating with relatively high performance on different architectures, such as NVIDIA GPUs, multicore CPUs and vector processors; and
- performance of almost all existing frameworks and libraries for NVIDIA GPU architectures can be further improved by applying additional optimisations, discussed in this paper.

2.2. VGL Abstractions

In [2, 5] we have proposed a set of four computational abstractions and four data abstractions, which can be efficiently implemented on the NEC SX-Aurora TSUBASA architecture. Further in this paper we will describe how these abstractions can be ported to the NVIDIA GPU architecture. However, first it is necessary to describe the main functionality of each abstraction and discuss why these abstractions are suitable for both classes of architectures.

Graph. A graph is the main data-abstraction of the VGL framework. Graphs in the VGL are stored in optimized and preprocessed VectCSR format [4]. The VGL framework provides a convenient interface for working with both directed and undirected graphs. For directed graphs, outgoing and incoming edges are stored for each vertex, while for undirected graphs all edges are stored as outgoing. This allows VGL users to easily implement pull-based and push-based algorithms [7].

Frontier. Frontier is a specific subset of graph vertices and the second important data-abstraction of the VGL. Frontiers in the VGL allows to control which vertices and edges need to be processed inside computational abstractions. For example, the advance abstraction processes all vertices from the input frontier, as well as all their adjacent edges. Frontiers in the VGL have different types: sparse, dense, and all-active (last one includes all graph vertices). Sparse

frontiers are represented via lists of indices, while dense – via an array of flags, where each flag corresponds to the presence of vertex inside the frontier.

Vertices array. Vertices arrays allow storing information about graph vertices, for example current level of each vertex in the BFS (Breadth-First Search) algorithm, or distances to each vertex in the shortest paths algorithms. Vertices arrays have a straightforward implementation using aligned arrays, allocated either in vector engine or unified memory of GPUs.

Edges array. Edges arrays allow storing information about graph edges, which is required when working with weighted graphs. Weighted edges are stored as a structure of arrays, providing better memory access pattern for vector instructions and warps.

Advance. The advance abstraction is the main tool of traversing graphs in the VGL. The advance input consists of a graph, an input frontier, and several user-defined handler functions: *vertex preprocess o*, *edge op*, *vertex postprocess op*. The advance applies *vertex preprocess op* to each vertex of its input frontier, *edge op* to each of its adjacent edges, and then *vertex postprocess op* to each vertex again. It is guaranteed that the execution of *vertex preprocess op*, edge-processing, and *vertex postprocess op* operations for each vertex are serialized. However, all *edge op* operations for each adjacent edge are executed in parallel. The computational workflow of the advance abstraction (as well as three others) is illustrated in Fig. 1. The advance abstraction is used in all situations, when processing graph edges is required.

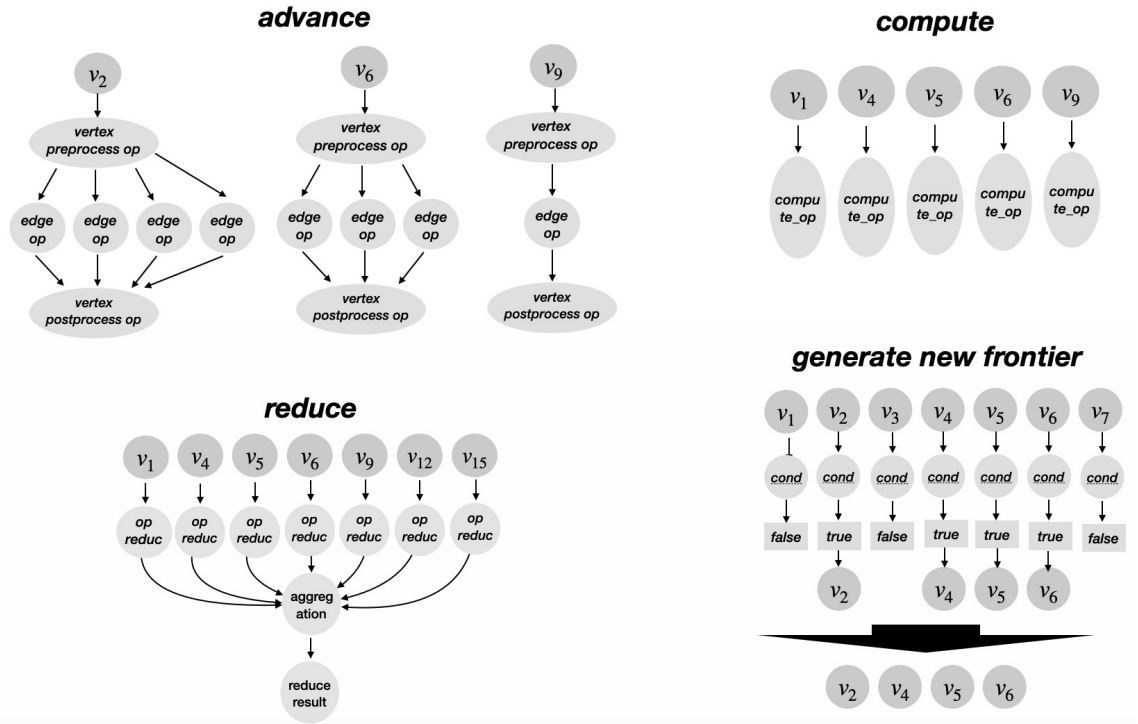


Figure 1. The computational workflow of VGL abstractions

Compute. The compute abstraction applies a user-defined operation to each vertex of the given input frontier. Typicality, this abstraction is used for a wide range of operations over graph vertices: initializing distances in shortest paths, implementing the "hook" phase in connected component algorithms, and many others.

Reduce. The reduce abstraction applies a user-defined operation (which returns some value) to each vertex of a given input frontier. The returned values are reduced using additionally specified reduction operation (SUM, MAX, MIN, AVG). This abstraction can be used for a

large number of applications: estimating future frontier size in BFS, calculating dangling nodes inputs in page rank, etc.

Generate New Frontier. The generate new frontier abstraction allows to create a new frontier of graph vertices, using a specified condition. This condition can be based on vertex id, its degree, or some data from user-defined vertices arrays.

The described computational abstractions are suitable for both NVIDIA GPUs and NEC SX-Aurora TSUBASA architectures for the following two reasons. First, each abstraction has a large resource of so-called data-driven parallelism, since they all execute the same operations over different data (graph vertices and edges). This allows to efficiently use vector instructions and warps, which is crucial for both architectures. In addition, each abstraction has a large resource of inner parallelism, since all graph vertices and edges can be processed in parallel, what is important since both architectures can be classified as massively-parallel. Thus, the described abstractions can be implemented on both architectures with approximately the same level of efficiency, in the case when correct optimisations are applied. These optimisations and implementation approaches will be described in details further in the paper.

3. Program Structure of VGL Framework

The software structure of the developed framework is illustrated in Fig. 2. This software structure allows VGL to operate on various target platforms, since all the abstractions have identical interfaces for all platforms. Each computational abstraction is implemented as a method of a base class, and has a basic OpenMP parallel implementation. For each architecture, a separate derived class can be created, where these methods are overloaded to contain architecture-specific implementations. Implementation and optimisation approaches inside overloaded methods are not limited to any extend; for example, abstractions for NVIDIA GPU can be implemented via CUDA, while for NEC SX-Aurora TSUBASA – by using special vector instructions and directives.

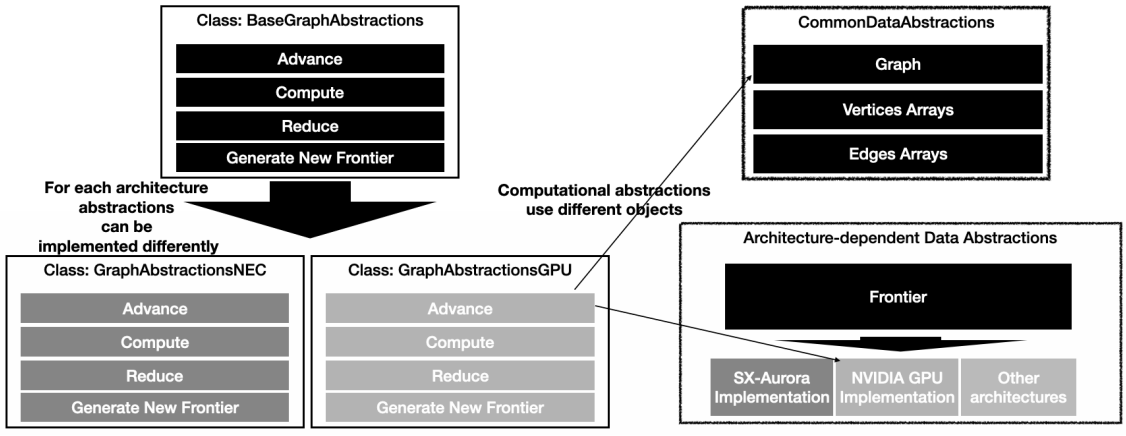


Figure 2. The software structure of the VGL framework. Third-party users are allowed to extend abstractions for different architecture by implementing derived classes

Data abstractions are split into two categories: architecture-dependent and architecture-independent. Graph, vertices array and edges array belong to architecture-independent category, since they have exactly the same implementation for each architecture currently supported in VGL. The frontier belongs to the architecture dependent category, since our experi-

ments demonstrated that frontiers require significantly different implementation approaches for NVIDIA GPU, SX-Aurora TSUBASA and multicore CPUs architectures.

The described software structure allows third-party users to extend VGL on the new architectures by implementing derived classes and changing several implementations of abstractions if necessary. Since interfaces of all abstractions remain exactly the same, graph algorithms implementations will also remain the same for different architectures, which makes VGL an architectural-independent framework.

4. Porting VGL Abstractions to NVIDIA GPU: a Detailed Comparison of Implementation and Optimisation Approaches

Despite the fact that VGL computational and data abstractions for different architectures can be implemented independently, for NVIDIA GPUs and NEC SX-Aurora TSUBASA architectures similar implementation and optimisation approaches can be used in many situations. Further in this section we will discuss which optimisations have been used for both architectures when implementing different computational abstraction. In the most important cases we will also demonstrate which acceleration has been achieved on each platform by applying each optimisation.

Advance. Implementing the advance abstraction on NVIDIA GPUs and vector architectures is difficult since it has highly-irregular computational workflow caused by (1) the irregular distribution of vertex degrees and (2) a large number of indirect memory accesses performed during edge traversals. Moreover, the advance abstraction contributes from 50% to 95% of execution time for many graph algorithms. The main optimisations used in the advance abstraction are listed in Tab. 2 together with the acceleration values obtained by implementing each of the optimisations.

Inter-core workload balancing in graph algorithms is typically implemented via splitting graph vertices into groups based on their degrees. Vertices from different groups are processed using different amount of hardware resources (cores). Dividing vertices into groups can be implemented either based on graph preprocessing (preliminary sorting graph vertices based on their degree), or dynamically during graph algorithm execution (using different vertex queues). According to our experiments both these approaches allow to achieve a comparable acceleration on NVIDIA GPUs, while for the NEC SX-Aurora TSUBASA architecture only the first approach (preprocessing) can be efficiently used. In order to provide better hardware utilisation, different groups of vertices can be simultaneously processed on the same GPU or vector engine. This optimisation can be implemented using CUDA-streams for NVIDIA GPUs, while OpenMP nested parallelism can be used for the NEC SX-Aurora TSUBASA architecture. This optimisation allows to achieve higher acceleration on NVIDIA GPUs, since modern GPUs have significantly larger resource of parallelism, and thus require more graph vertices and edges to be processed in parallel.

Low-degree vertices should be accurately processed on both architectures, since it is hard to efficiently use vector instructions or warps when loading information about their adjacent edges. On both architectures loading information about graph edges with load/store instructions must have sequential memory access pattern in order to maximise the sustained bandwidth. To achieve this goal we used two different techniques: constructing graph vector extension [4], or using

Table 2. The effect of different optimisations applied to the implementation of the advance abstraction

Optimisation	V100 GPU, acceleration (times)	NEC SX-Aurora, TSUBASA 1st generation, acceleration (times)
Inter-core workload balancing based on graph preprocessing	6.4	3.1
Dynamic inter-core workload balancing	5.5	0.93
Concurrent processing of different group of vertices	1.9	0.8
SIMD-processing of low-degree vertices based on vector extension	1.4–5.2	2.1–8.1
SIMD-processing of low-degree vertices based on virtual warp	4.8	0.5
Graph clusterisation	5	3.8
Prefetching most frequently accessed vertices into LLC	–	1.2
Switching from push to pull-based graph traversal	0.9	1.7
Packing indirectly accessed 4-byte into 8-byte	1.1	1.3

"virtual warp" concept [12]. Vector extension allows to process groups of *VECTOR_LENGTH* vertices by simultaneously processing first edges of all vertices (using a single SIMD instruction), then second, and etc. Virtual warps concept is based on splitting SIMD instruction in separate parts, with each part processing vertices in a fixed range, as shown in Fig. 3. Vector extension allows to achieve a very significant and comparable acceleration on both architectures, while applying "virtual warp" concept to vector instructions leads to program slowdown on the NEC SX-Aurora TSUBASA architecture.

In order to efficiently load information about indirectly accessed graph vertices, the clusterisation [21] should be used for both architectures. The clusterisation is based on grouping information about most frequently accessed graph vertices in the adjacent regions of memory, which can be later prefetched into LLC cache (which allows to obtain an additional acceleration on the NEC SX-Aurora TSUBASA).

The performance of the advance abstraction also depends on the direction, in which graph edges are traversed. During pull traversal [7] in VGL the incoming edges are processed, while during push – the outgoing. According to our experiments, pull-direction is preferable for NEC SX-Aurora TSUBASA architecture, while push – on GPUs. Finally, for the NEC SX-Aurora TSUBASA multiple indirectly accessed values can be packed into 8-byte values, since gather and scatter instructions to 8-byte values are approximately 2 times faster compared to 4-byte values for this architecture. On NVIDIA GPUs, such optimisation does not provide any significant acceleration.

Compute. The implementation of the compute abstraction on both architectures is almost identical and straightforward, since all its operations can be performed independently in parallel.

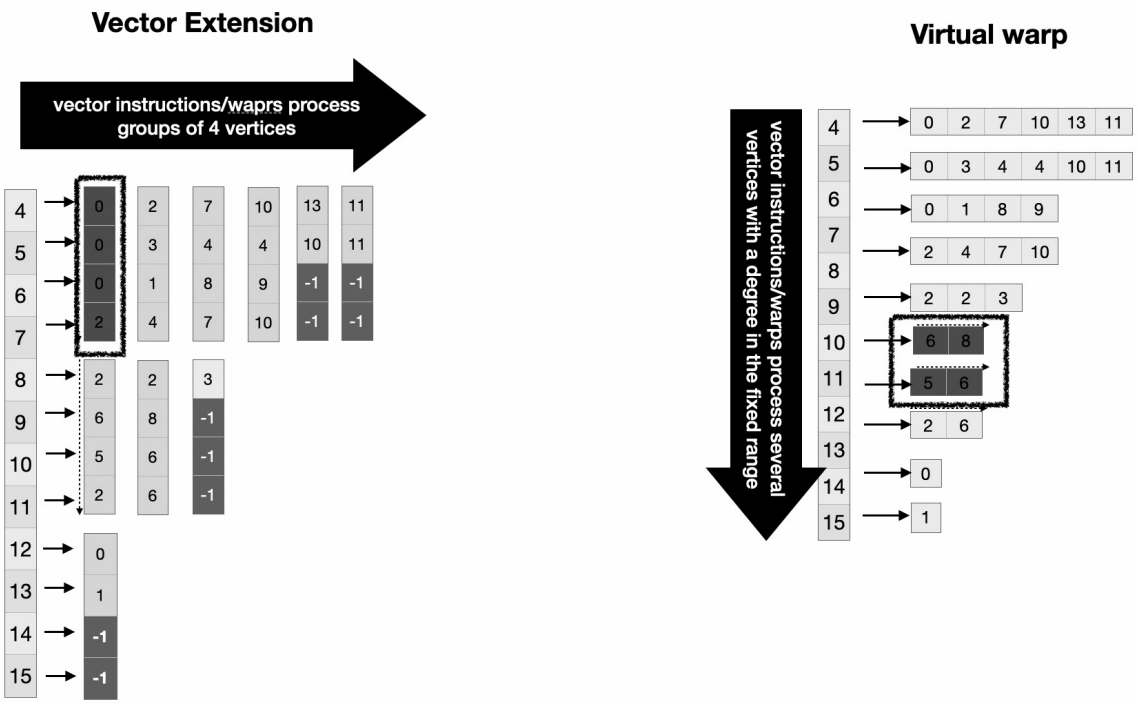


Figure 3. Vector extension against virtual warps comparison. Red edges are simultaneously processed using a single vector instruction of length 4

On NVIDIA GPUs the compute abstraction is implemented via a parallel CUDA kernel, while on NEC SX-Aurora TSUBASA – via a vectorized parallel loop.

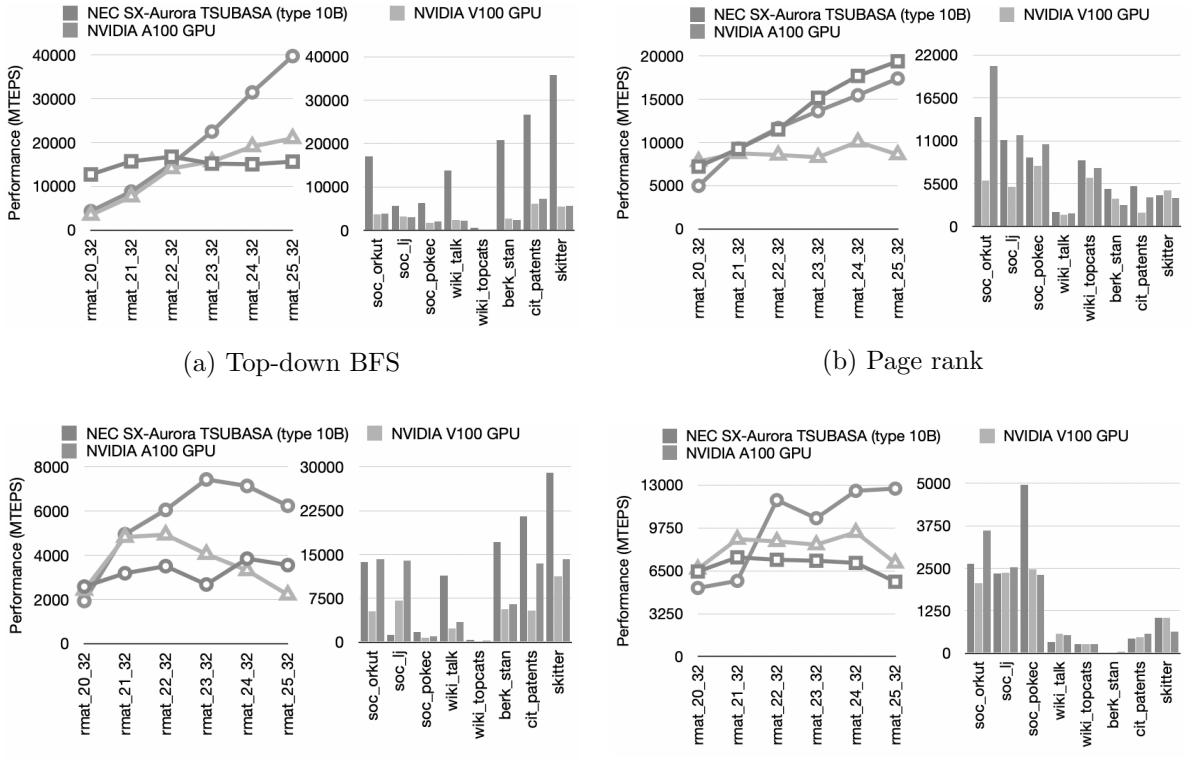
Reduce. For the NEC SX-Aurora TSUBASA architecture, the reduce abstraction is implemented via a vectorized and parallelized loop, where each vector core accumulates the reduced values on vector registers. This way the reduction vector instructions are executed only on the last stage of algorithm, when the obtained on vector registers values are reduced into the scalars. On the NVIDIA GPU architecture, the reduction is implemented based on using parallel reduction inside shared memory [10]. However, the reduce is implemented on NVIDIA GPUs less efficiently since shared memory has a higher latency compared to vector registers.

Generate New Frontier. The generate new frontier abstraction on GPUs is implemented based on parallel prefix sum algorithm [11], which generates indexes of vertices from the output frontier. On the NEC SX-Aurora TSUBASA a different algorithm is implemented [5], which generates lists of frontier indexes using special vector buffers, later unrolling them into a linear list. Both these approaches demonstrate approximately the same performance.

5. Performance Evaluation

The performance of the VGL framework has been evaluated on cluster equipped with (1) 12-core Intel (R) Xeon (R) Gold 6126 processors, (2) NVIDIA V100 and (3) A100 GPUs, and (4) SX-Aurora TSUBASA Type 10B (First Generation) vector engines. Unfortunately, at the moment of this writing we do not have an access to the second generation of SX-Aurora TSUBASA architecture. As input graphs we used synthetic RMAT [8] and several real-world graphs from the SNAP [1] collection.

The performance evaluation is split into two stages. On the first stage we compared the performance of VGL-based implementations launched on SX-Aurora TSUBASA, V100 and A100 GPUs. This comparison is demonstrate in Fig. 4 for different graph problems and algorithms.



(a) Top-down BFS

(b) Page rank

(c) Generalized Bellman–Ford shortest paths

(d) Shilloah–Vishkin connected components algorithm

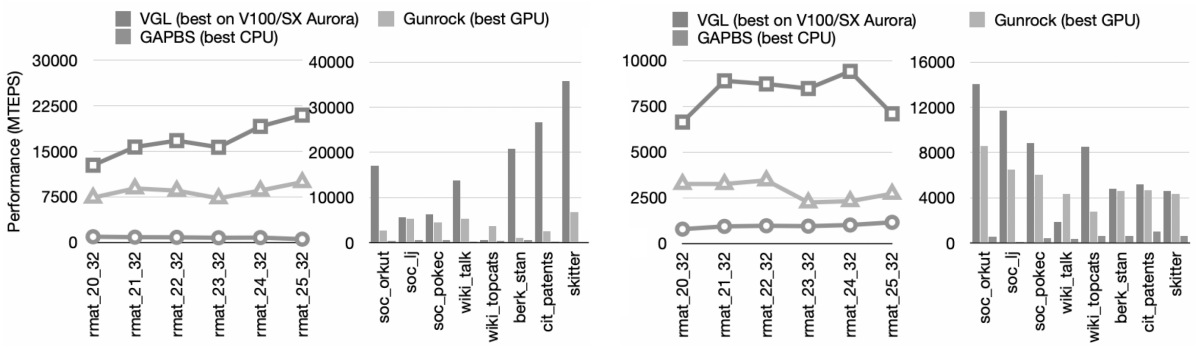
Figure 4. The comparison of VGL-based implementations of different graph problems and algorithms

The following conclusions can be made based on the provided performance data. The first generation NEC SX-Aurora TSUBASA vector engines have a comparable with V100 performance on medium-sized and large graphs, which once again confirms the thesis about the similarity of these architectures. At the same time, SX-Aurora implementations are significantly faster on small-sized RMAT and most of real-world graphs, which have relatively small size. This can be explained by the fact that SX-Aurora requires significantly less vertices and edges, which have to be processed in parallel in order to efficiently utilize hardware resources.

At the second stage, the importance of developing of an architectural-independent framework is demonstrated in Fig. 5. For a specified graph problem and input graph we selected the fastest VGL-based implementation (among SX-Aurora and V100 GPU architectures), and compared it to the fastest available among CPU-based and GPU-based frameworks and libraries, listed in Section 2. The necessity of selecting different architectures can be explained by the fact that different architectures are faster at processing different input graphs, as shown in Fig. 4. According to our experiments, the Gunrock framework and NVGRAPH library provide the highest performance on NVIDIA GPUs, while GAPBS library has the highest performance among single-socket multicore CPU implementations. As shown in Fig. 5, VGL outperforms these frameworks and libraries from 3 to 15 times on different input graphs.

Conclusion

In this paper we have described the first-in-the-world attempt to develop an architecture-independent graph framework VGL, which targets multiple modern systems with high-



(a) Top-down BFS

(b) Shilloah–Vishkin connected components algorithm

Figure 5. The comparison of VGL-based implementations to best available multicore CPU and NVIDIA GPU frameworks

bandwidth memory: NEC SX-Aurora TSUBASA and NVIDIA GPUs. VGL has from 3 to 15 times better performance compared to the existing frameworks, developed for modern multicore CPUs and NVIDIA GPUs. Moreover, due its flexible software structure, the VLG framework can be easily extended to other massively parallel architectures, such as the A64FX, AMD EPYC Rome and Intel KNL, which is an important direction of future research.

Finally, in this paper we have compared optimisation approaches, which should be used in order to efficiently implement graph algorithms on NEC SX-Aurora TSUBASA vector processors and NVIDIA GPUs. Applying various optimisations, such as graph clusterisation or constructing graph vector extension, allowed to achieve similar acceleration on both these architectures, which emphasizes the similarity of these architectures in the context of implementing various graph algorithms.

Acknowledgements

The reported study was funded by RFBR and JSPS, project number 21-57-50002.

References

1. Stanford Large Network Dataset Collection – SNAP. `https://snap.stanford.edu/data/` (2020), accessed: 2020-12-29

2. Afanasyev, I.V.: Developing a prototype of high-performance graph-processing framework for NEC SX–Aurora TSUBASA vector architecture. Numerical methods and programming 21, 290–305 (2020), DOI: 10.26089/NumMet.v21r325

3. Afanasyev, I.V., Voevodin, V.V., Kobayashi, H., et al.: Analysis of relationship between SIMD-processing features used in NVIDIA GPUs and NEC SX-Aurora TSUBASA vector processors. In: Malyshkin, V. (ed.) International Conference on Parallel Computing Technologies, PaCT 2019. Lecture Notes in Computer Science, vol. 11657, pp. 125–139. Springer (2019), DOI: 10.1007/978-3-030-25636-4_10

4. Afanasyev, I.V., Voevodin, V.V., Kobayashi, H., et al.: Developing efficient implementations of shortest paths and page rank algorithms for NEC SX-Aurora TSUBASA architecture. Lobachevskii Journal of Mathematics 40(11), 1753–1762 (2019), DOI: 10.1134/S1995080219110039

5. Afanasyev, I.V., Voevodin, V.V., Komatsu, K., et al.: VGL: a high-performance graph processing framework for the NEC SX-Aurora TSUBASA vector architecture. The Journal of Supercomputing (2021), DOI: 10.1007/s11227-020-03564-9

6. Azad, A., Aznaveh, M.M., Beamer, S., et al.: Evaluation of graph analytics frameworks using the GAP benchmark suite. In: IEEE International Symposium on Workload Characterization, IISWC 2020, 27-30 October 2020, Beijing, China. pp. 216–227. IEEE (2020), DOI: 10.1109/IISWC50251.2020.00029

7. Besta, M., Podstawski, M., Groner, L., et al.: To push or to pull: On reducing communication and synchronization in graph computations. In: Huang, H.H., Weissman, J.B., Iamnitchi, A., et al. (eds.) Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing, HPDC 2017, 26-30 June 2017, Washington, DC, USA. pp. 93–104. ACM (2017), DOI: 10.1145/3078597.3078616

8. Chakrabarti, D., Zhan, Y., Faloutsos, C.: R-MAT: A recursive model for graph mining. In: Berry, M.W., Dayal, U., Kamath, C., et al. (eds.) Proceedings of the Fourth SIAM International Conference on Data Mining, 22-24 April 2004, Lake Buena Vista, Florida, USA. pp. 442–446. SIAM (2004), DOI: 10.1137/1.9781611972740.43

9. Flynn, M.J.: Very high-speed computing systems. Proceedings of the IEEE 54(12), 1901–1909 (1966), DOI: 10.1109/PROC.1966.5273

10. Harris, M., et al.: Optimizing parallel reduction in CUDA. NVIDIA Developer Technology 2(4), 70 (2007)

11. Hillis, W.D., Steele Jr., G.L.: Data parallel algorithms. Commun. ACM 29(12), 1170–1183 (1986), DOI: 10.1145/7902.7903

12. Hong, S., Kim, S.K., Oguntebi, T., et al.: Accelerating CUDA graph algorithms at maximum warp. SIGPLAN Not. 46(8), 267–276 (2011), DOI: 10.1145/2038037.1941590

13. Khorasani, F., Vora, K., Gupta, R., et al.: CuSha: vertex-centric graph processing on GPUs. In: Plale, B., Ripeanu, M., Cappello, F., et al. (eds.) The 23rd International Symposium on High-Performance Parallel and Distributed Computing, HPDC'14, 23-27 June 2014, Vancouver, BC, Canada. pp. 239–252. ACM (2014), DOI: 10.1145/2600212.2600227

14. Kirk, D.: NVIDIA CUDA software and GPU parallel computing architecture. In: Morrisett, G., Sagiv, M. (eds.) Proceedings of the 6th International Symposium on Memory Management, ISMM 2007, 21-22 October 2007, Montreal, Quebec, Canada. pp. 103–104. ACM (2007), DOI: 10.1145/1296907.1296909

15. Komatsu, K., Watanabe, O., Musa, A., et al.: Performance evaluation of a vector supercomputer SX-Aurora TSUBASA. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, SC 2018, 11-16 November 2018, Dallas, TX, USA. pp. 54:1–54:12. IEEE Press (2018)

16. Liu, H., Huang, H.H.: Enterprise: breadth-first graph traversal on GPUs. In: Kern, J., Vetter, J.S. (eds.) Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2015, 15-20 November 2015, Austin, TX, USA. pp. 68:1–68:12. ACM (2015), DOI: 10.1145/2807591.2807594

17. Nguyen, D., Lenharth, A., Pingali, K.: A lightweight infrastructure for graph analytics. In: Kaminsky, M., Dahlin, M. (eds.) ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, 3-6 November 2013, Farmington, PA, USA. pp. 456–471. ACM (2013), DOI: 10.1145/2517349.2522739

18. Shun, J., Blelloch, G.E.: Ligra: a lightweight graph processing framework for shared memory. SIGPLAN Not. 48(8), 135–146 (2013), DOI: 10.1145/2517327.2442530

19. Wang, Y., Davidson, A.A., Pan, Y., et al.: Gunrock: a high-performance graph processing library on the GPU. In: Asenjo, R., Harris, T. (eds.) Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2016, 12-16 March 2016, Barcelona, Spain. pp. 11:1–11:12. ACM (2016), DOI: 10.1145/2851141.2851145

20. Yamada, Y., Momose, S.: Vector Engine Processor of NEC Brand-New supercomputer SX-Aurora TSUBASA. In: International symposium on High Performance Chips, Hot Chips 2018, August 2018, Cupertino, USA (2018)

21. Zhang, Y., Kiriansky, V., Mendis, C., et al.: Making caches work for graph analytics. In: Nie, J., Obradovic, Z., Suzumura, T., et al. (eds.) 2017 IEEE International Conference on Big Data, BigData 2017, 11-14 December 2017, Boston, MA, USA. pp. 293–302. IEEE Computer Society (2017), DOI: 10.1109/BigData.2017.8257937

22. Zhong, J., He, B.: Medusa: Simplified graph processing on GPUs. IEEE Trans. Parallel Distributed Syst. 25(6), 1543–1552 (2014), DOI: 10.1109/TPDS.2013.111