



VGL: a high-performance graph processing framework for the NEC SX-Aurora TSUBASA vector architecture

Ilya V. Afanasyev, et al. *[full author details at the end of the article]*

Accepted: 10 December 2020 / Published online: 26 January 2021

© The Author(s), under exclusive licence to Springer Science+Business Media, LLC part of Springer Nature 2021

Abstract

Developing efficient graph algorithms implementations is an extremely important problem of modern computer science, since graphs are frequently used in various real-world applications. Graph algorithms typically belong to the data-intensive class, and thus using architectures with high-bandwidth memory potentially allows to solve many graph problems significantly faster compared to modern multicore CPUs. Among other supercomputer architectures, vector systems, such as the SX family of NEC vector supercomputers, are equipped with high-bandwidth memory. However, the highly irregular structure of many real-world graphs makes it extremely challenging to implement graph algorithms on vector systems, since these implementations are usually bulky and complicated, and a deep understanding of vector architectures hardware features is required. This paper presents the world first attempt to develop an efficient and simultaneously simple graph processing framework for modern vector systems. Our vector graph library (VGL) framework targets NEC SX-Aurora TSUBASA as a primary vector architecture and provides relatively simple computational and data abstractions. These abstractions incorporate many vector-oriented optimization strategies into a high-level programming model, allowing quick implementation of new graph algorithms with a small amount of code and minimal knowledge about features of vector systems. In this paper, we evaluate the VGL performance on four widely used graph processing problems: breadth-first search, single source shortest paths, connected components, and page rank. The provided comparative performance analysis demonstrates that the VGL-based implementations achieve significant acceleration over the existing high-performance frameworks and libraries: up to 14 times speedup over multicore CPUs (Ligra, Galois, GAPBS) and up to 3 times speedup compared to NVIDIA GPU (Gunrock, NVGRAPH) implementations.

Keywords NEC SX-Aurora TSUBASA · Graph algorithms · Graph frameworks · VGL · Vector processing · High-performance computing

1 Introduction

Developing efficient implementations of graph algorithms is an extremely important problem in modern computer science, since graphs successfully model various real-world objects from different areas. Graph processing is frequently used in social networks and web graphs analysis, solving infrastructure and biological problems, socioeconomic modeling, and many others. For example, implemented in this paper shortest paths algorithms allow to solve navigation problems, where graphs represent road networks, while page rank algorithm (and its modifications) is frequently used in search engines to rank webgraph.

Graph algorithms belong to the data-intensive class; thus they usually heavily stress the memory subsystem of target platforms. Using systems with HBM (high-bandwidth memory) can significantly accelerate many graph algorithms. Currently, HBM is installed mainly in GPUs, vector processors (NEC SX family of supercomputers) and processors with vector extensions (A64FX), since computational principles of these architectures allow to efficiently utilize available memory throughput using collective accesses to the memory subsystem performed by GPU cores or vector units.

Though there are many high-performance implementations of various graph algorithms for modern NVIDIA GPUs and multicore CPUs, the possibility of efficiently implementing graph algorithms for modern vector systems is poorly studied. Due to the highly irregular structure of many real-world graphs, it is extremely complicated to efficiently use vector data processing, operating over regular data structures: In order to achieve maximum performance, vector units have to perform operations of the same type over adjacent memory cells using vector instructions with a fixed vector length. Moreover, even when graph algorithms can be efficiently vectorized, their implementations tend to be bulky and extremely complicated. For example, each implementation described in [1, 2] includes up to 1000 lines of code and a large number of micro-architectural optimizations.

Developing high-level graph processing frameworks allows application developers (framework users) to significantly simplify implementation of efficient vector-oriented graph algorithms even for those developers, who are not familiar with concepts of vectorization and parallel programming.

2 Target architecture: NEC SX-Aurora TSUBASA vector processors

The NEC SX-Aurora TSUBASA architecture with dedicated vector processors [14, 26] is the primary target architecture of the VGL framework. This architecture inherits the design concepts of vector supercomputer and enhances its advantages to achieve higher sustained performance and higher usability. Different from its predecessors in the SX supercomputer series [7, 13], 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 a 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 at 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 the functionality of a 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 at 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 (DP) elements. On the memory subsystem side, six HBM modules in the vector processor can deliver the 1.22 TB/s memory bandwidth [7] with up to 48 GB total capacity.

Parallel programs for the NEC SX-Aurora architecture are implemented using the OpenMP programming model, while vectorization is performed by the NEC compiler: A developer inserts compiler-specific directives, which help the compiler to perform automatic vectorization. Although NEC SX-Aurora TSUBASA has many similar hardware and architecture features with NVIDIA GPUs [3], special programming models such as CUDA (Compute Unified Device Architecture) or OpenCL cannot be used to port the existing graph processing frameworks to SX-Aurora. At the same time, the existing frameworks for multicore CPUs demonstrate poor performance on vector systems, since they do not take into an account vector computational feature.

3 Related work: graph algorithms and vector systems

Graph processing frameworks typically include several highly optimized computational and data abstractions, which allow to express various graph algorithms for different target architectures. Such abstractions usually represent some graph-related objects (graphs, vertices, and edges) and computational operations on them. Framework user implements the desired graph algorithms as a combination of these abstractions and uses the developed implementations to process real-world graphs from a specific application area. When using graph frameworks, the user saves a lot of time and efforts compared to the situation, when he needs to manually implement and optimize a specific graph algorithm for a specified target architecture.

Many high-level graph processing frameworks have been proposed for different supercomputing architectures. Ligra [21] and Galois [18] are the most well-known multicore CPUs frameworks, while Gunrock [24], CuSHA [12], Medusa [28], and Enterprise [15] frameworks target NVIDIA GPU architecture. However, no existing

frameworks are aimed at operating on modern vector systems, such as NEC SX-Aurora TSUBASA.

3.1 Conventional high-level GPU frameworks

Medusa [28] is one of the first high-level parallel graph processing system for NVIDIA GPUs, which is based on using a message-passing model. CuSha [12] uses different Gather Apply Scatter (GAS) abstraction by using parallel sliding window to avoid non-coalesced memory access and the efficiency of using shared memory. MapGraph [8] also adopts the GAS abstraction. Gunrock [24] implements a very different data-driven abstraction, which is based on representing subsets of graph vertices and edges called "frontiers." Frontiers are used to advance (traverse) graphs, similar to how breadth-first search algorithm operate. Gunrock currently achieves one of the best performance among programmable single-node GPU graph computations. Unfortunately, all these frameworks cannot be used for vector systems such as SX-Aurora, since their implementation is based on the CUDA programming model.

3.2 Conventional high-level CPU frameworks

Ligra [21] is a multicore CPU-based framework for shared-memory systems. Its abstraction is based on using vertex and edge maps—objects, similar to the Gunrock frontiers. Galois [18] is another high-level abstraction for shared-memory systems, which supports priority scheduling, dynamic graphs, and working with subsets of vertices called active elements. However, the Galois model does not abstract the internal details of the loop. Graph application developers have to generate the active elements set directly for different graph algorithms. Cagra [27] is another interesting framework, which aims to improve the utilization of CPU caches using so-called segmentation and clusterization. Unfortunately, these frameworks also cannot be used for vector systems, since they do not include any vector-oriented optimizations.

3.3 Evaluating the performance of graph applications

Roofline [25] and cache-aware roofline [11] models are frequently used to estimate the efficiency of executions of various applications, including graph algorithms. Since most of graph algorithms have low computational complexity, it is possible to avoid generating a full roofline model for estimating efficiency of graph applications. Instead, the efficiency can be calculated as $\frac{SB}{TB}$, where SB is the sustained bandwidth of a graph application, equal to the amount of bytes requested divided by the execution time of the algorithm. TB is the theoretical bandwidth of a target platform that is 1.2 TB/s for NEC SX-Aurora TSUBASA. This approach to calculating the efficiency in the VGL framework will be frequently used further in the paper.

4 VGL high-level graph processing abstractions

The VGL framework is designed to easily implement various iterative graph algorithms including breadth-first search (BFS) graph traversals, calculating shortest paths (SSSP), (strongly)connected components (CC), minimal spanning tree (MST), maximum flow (MF), page ranking (PR), and many others. The iterative graph algorithms process a certain subset of graph vertices and edges at each iteration, when some computational operations are performed over vertices and edges of the input graph. The iterative graph algorithms usually provide enough data-driven parallelism: Vertices and edges can be processed at each iteration independently (and therefore in parallel) using the same compute operations over each vertex or edge, which allows to efficiently utilize vector-processing features of NEC SX-Aurora Tsubasa architecture.

The frontier (a subset of graph vertices) is a central data abstraction of the VGL framework, similar to Gunrock and Ligra. Typically, users of the VGL framework define various subsets of graph vertices based on some criteria. Afterwards, the user applies various computational operations (further in the paper referred as “abstractions” or “computational abstractions”) to these frontiers. Thus, the VGL framework implements a bulk synchronous parallel (BSP) computational model: The user describes an algorithm as multiple steps, each of which involves applying some computational abstraction to a specific subset of graph vertices and edges. While data dependencies between different steps may exist, vertices and edges in the frontier (or subset) can be processed in parallel. Unlike the conventional frameworks such as Gunrock and Ligra, VGL does not provide a concept of edge frontiers. All VGL computational abstractions are arranged around processing a specific subset of graph vertices and all their adjacent edges.

VGL provides four basic computational abstractions: *advance*, *generate_new_frontier*, *compute*, and *reduce*.

4.1 Data abstractions

4.1.1 Graph

A graph is the main data abstraction of the VGL framework. Graphs in VGL are stored in the optimized and preprocessed format (VectCSR), which is based on the compressed sparse row (CSR) representation. 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. For undirected graphs, all edges are stored as outgoing. This allows VGL users to easily implement pull-based, push-based [5], and mixed [4] algorithms for any type of graph

4.1.2 Frontier

Frontier of graph vertices is the second important data abstraction in VGL. Frontier in VGL can be one of three types: “all-active,” “dense,” or “mixed.” “All-active”

frontiers include all graph vertices and allow us to significantly reduce an overhead for the frontier maintenance. “All-active” frontiers are beneficial for implementing graph algorithms, in which all graph vertices and edges have to be processed at each algorithm iteration (e.g., Bellman–Ford). “Dense” and “mixed” frontiers, respectively, represent cases when most/few graph vertices belong to the frontier. A detailed description of VGL frontier representations will be provided in the following sections.

4.1.3 Vertices array

The *VerticesArray* abstraction allows storing information about graph vertices, for example current level of each vertex (in BFS algorithms) or current distances to each vertex (in shortest paths algorithms). *VerticesArray* abstraction has a straightforward implementation using aligned arrays.

4.1.4 Edges array

The *EdgesArray* abstraction allows storing information about graph edges. Using *EdgesArray* VGL supports storing weighted graphs. This way weighted edges are stored as a structure of arrays, providing better memory access pattern for each vector instruction, which loads information about graph edges.

4.2 Computational abstractions

4.2.1 Advance

The *advance* abstraction is the main tool of traversing graph in VGL. For many real-world graphs, the *advance* has highly irregular computational workflow due to the irregular distribution of vertex degrees and a large number of indirect memory accesses. The *advance* input consists of a graph, an input frontier, and several user-defined handler functions: *vertex_preprocess_op*, *edge_op*, *vertex_postprocess_op* as shown in listing 1. During its execution, the *advance* applies *vertex_preprocess_op* to each vertex of input frontier, *edge_op* to each of its adjacent edges, and then *vertex_postprocess_op* to the vertex again. It is guaranteed that the execution of *vertex_preprocess_op*, edge-processing, and *vertex_postprocess_op* operations for each vertex is serialized. However, all *edge_op* operations for each adjacent edge are executed in parallel. In addition, all frontier vertices can also be processed in parallel. A computational scheme of the advance abstraction is provided in Fig. 1. Typically, vertex preprocess (or postprocess) operations are used to initialize (or finalize) some data processing, to be performed during edge traversals. For example, in the page rank algorithm, the *edge_op* operation can be defined to accumulate new ranks, while postprocess operations can be implemented to save the result using the input of dangling nodes and the coefficients of the page rank.

VGL also provides an additional version of the *advance* abstraction, which generates a new frontier as an output. This version of *advance* in addition requires a

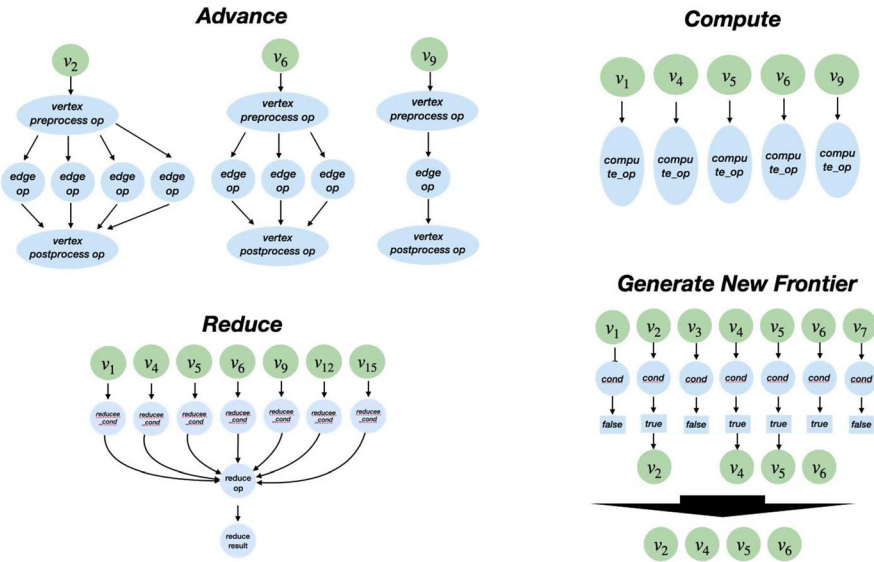


Fig. 1 Main VGL computational abstractions: *advance*, *compute*, *reduce*, and *generate_new_frontier*

user-defined function used to determine whether a graph vertex belongs to the new frontier or not. The output frontier of the *advance* abstraction may only include vertices, adjacent to the input frontier, which significantly reduces the computational complexity required for generating the frontier.

For processing directed graphs, VGL provides two wrappers over the *advance* abstraction: *gather* and *scatter*. The *scatter* executes the *advance* over outgoing edges of each vertex, while *gather*—over incoming. When working with undirected graphs, *advance*, *gather*, and *scatter* can be used, providing the same results.

Listing 1: A part of the VGL API set. User-defined *edge_op* and *vertex_op* operations declare computational logic of an algorithm, receiving various graph objects (for example, source and destination indexes) and special API-objects, aimed at performance tuning (*delayed_write*). *Advance*, *generate_new_frontier*, and *compute* abstractions receive graph, frontiers and user-defined C++ lambda functions.

```

1 auto EDGE_OP = [] (int src_id, int dst_id, int local_edge_pos, long long int global_edge_pos, int vector_index,
2                 DelayedWriteNEC &delayed_write) {};
3 auto VERTEX_OP = [] (int src_id, int connections_count, int vector_index, DelayedWriteNEC &delayed_write){};
4
5 template <typename TVertexValue, typename TEdgeWeight, typename EdgeOperation, typename VertexPreprocessOperation,
6          typename VertexPostprocessOperation>
7 void advance(ExtendedCSRGraph<TVertexValue, TEdgeWeight> &_graph,
8            FrontierNEC &_frontier, EdgeOperation &&edge_op,
9            VertexPreprocessOperation &&vertex_preprocess_op,
10           VertexPostprocessOperation &&vertex_postprocess_op);
11
12 template <typename TVertexValue, typename TEdgeWeight, typename Condition>
13 void generate_new_frontier(ExtendedCSRGraph<TVertexValue, TEdgeWeight> &_graph,
14                          FrontierNEC &_frontier,
15                          Condition &&cond);
16
17 template <typename TVertexValue, typename TEdgeWeight, typename ComputeOperation>
18 void compute(ExtendedCSRGraph<TVertexValue, TEdgeWeight> &_graph,
19            FrontierNEC &_frontier,
20            ComputeOperation &&compute_op);

```

4.2.2 Generate new frontier

The *generate_new_frontier* abstraction allows a user to create a new frontier of graph vertices. As an input, this abstraction receives a graph and user-defined condition. Then, this abstraction generates a new frontier of vertices for which the provided condition returns *IN_FRONTIER* (true) flag.

4.2.3 Compute

The *compute* abstraction applies a user-defined *compute_op* operation to each vertex of the given input frontier. Since all *compute_op* operations can be independently executed, the *compute* abstraction can be implemented in a straightforward way on NEC SX-Aurora TSUBASA architecture. Typicality, this abstraction is used for wide range of operations over graph vertices: initializing distances in shortest paths, implementing the "hook" phase in connected component algorithms, and many others.

4.2.4 Reduce

The *reduce* abstraction applies a user-defined *reduce_op* 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. The conventional frameworks, such as Gunrock, implement functionality of the *reduce* abstraction using a combination of the *compute* abstraction (or its analogues) and user-defined operations with atomic instructions. However, atomic operations on the SX-Aurora architecture easily becomes a performance bottleneck compared to GPUs and multicore CPUs. Thus, the atomic operation must be avoided in the VGL framework. Furthermore, the reduction implementation for the SX-Aurora architecture is much more efficient compared to highly optimized thrust and "modernGPU" GPU libraries [3], which is the main reason behind implementing the *reduce* as a separate abstraction in the VGL.

5 VGL implementation details

5.1 The Graph Storage Format

The VGL framework extends the vector-oriented graph storage format called "VectCSR" proposed in [1], which is effective for page rank and shortest paths algorithms implementations on the NEC SX-Aurora TSUBASA.

In VGL, graphs are always preprocessed as shown in Fig. 2. All graph vertices are sorted and then renumbered in the descending order with a sort key equal to the number of adjacent edges for each vertex. This sorting implements clustering

optimization [27], which allows us to use cache memory more efficiently by storing only the most frequently accessed graph vertices into the cache.

In addition, in the VectCSR format, graph vertices are split into three groups based on the number of adjacent edges for each vertex. The first group stores vertices with the "large" number of adjacent edges ($\geq 8 * 256$), second—with "medium" ($\geq 256, < 8 * 256$), third—with "small" (< 256). Afterward, we will denote these groups with "large-degree," "medium-degree", and "small-degree" groups, or as "first," "second," and "third." Computational abstractions process vertices of each group differently: Vertices from the "medium-degree" and "large-degree" groups generally allow vector processing of their adjacent edges using vector instructions with the maximum length, while vertices from the "small" group—do not. In order to improve memory access pattern when processing "small-degree" vertices, their adjacent edges are stored separately in a vector extension (Fig. 2, right). Both vector extension and CSR representations are used inside computational abstractions: Vector extension is used when working with dense frontiers, while the CSR representation is used when working with mixed (sparse).

As it was mentioned earlier in the paper, VGL framework supports working with both directed and undirected graphs. In directed graphs, incoming and outgoing edges are stored in two separate VectCSR data structures. Since the same vertex can possibly have drastically different incoming and outgoing degree in directed graph, graph vertices usually have different numbers and positions inside two VectCSR formats, used to store incoming and outgoing edges. To allow switching between different traversal directions (*gather* and *scatter*), the VGL stores multiple index reordering arrays, as well as provides special API functions for reordering *VertexArrays* and *frontiers*.

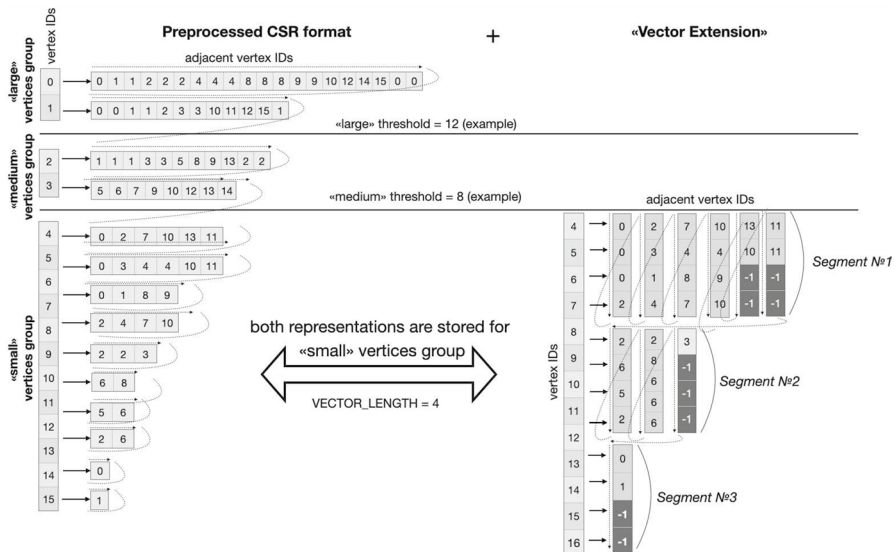


Fig. 2 VectCSR graph storage format: preprocessed CSR part (left) and its vector extension (right). Graph is split on three sample groups based on degrees of vertices

5.2 Frontier representation

The frontier representation in VGL largely depends on the number of vertices composing it. Each frontier tracks the number of vertices placed inside it during each frontier modification, while not allowing duplicate vertices (with the same ID) to be stored inside.

“All-active” frontiers are represented in VGL as a single flag, which reflects whether all graph vertices belong to the frontier or not. A frontier can be either directly initialized as “all-active” (using special class method or user-defined criteria), or frontiers of the other types can become all-active during computations, which is very easy to check since each frontier is tracking the number of its vertices and also removes duplicate vertices.

The dense frontier in VGL is represented as an array of flags and is used when most of graph vertices belong to it. Each flag has the 4-byte integer type, since SX-Aurora does not allow vectorized processing of smaller data types (char, short, bool). Each flag is equal to the result of user-defined conditional operation, passed to some computational abstraction capable of updating the frontier (*advance* and *generate_new_frontier*).

When the frontier consists only of a small number of vertices, it has a “mixed” type. Vertices of the “mixed” frontier are divided into three groups, based on the number of adjacent edges for each vertex—exactly with the same thresholds used in the VectCSR graph representation. Each group of vertices can be either dense or sparse independently from others: Dense groups are represented via arrays of flags, while sparse groups are represented via lists of vertex IDs. Splitting vertices of “mixed” frontiers into three groups are motivated by two reasons. First, many graph algorithms usually visit more “high-degree” vertices in the first iterations, while more “small-degree” vertices on the later iterations for various real-world graphs. In addition, the operation of generating a list of vertex IDs on the SX-Aurora architecture is not vector friendly (to be shown in next section) and thus should be avoided if possible.

Computational abstractions, which are allowed to create new frontiers (*advance* and *generate_new_frontier*) automatically, determine the output frontier type based on build-in criteria of the frontier sparsity. The standard criteria are based on ratios of vertices currently stored in each group to the maximum possible number of vertices in each group. For example, a dense “small-degree” group of vertices becomes sparse when it contains less than 30% of total “small-degree” graph vertices. Each criterion can be altered by user. For example, one may choose the criteria for a group, depending on the total number of outgoing frontier edges, as it implemented in Ligra [21].

5.3 Generating a new frontier

A user of VGL framework generates a new frontier by either using the *generate_new_frontier* abstraction or obtaining it as a result of the *advance* abstraction (variation which returns a frontier). Further in this section, we will discuss in detail

algorithm used in the *generate_new_frontier* abstraction. First, an array of flag is generated, which determines whether each vertex belongs to the generated frontier or not. Each flag can be easily obtained by checking the return value of the conditional operator passed into the *generate_new_frontier* abstraction, and this flag will be further used in dense and mixed frontier representations. When generating an array of flags, the number of vertices in each of three vertex groups (“large,” “medium,” “small”) is calculated, which can be implemented without any overhead by accumulating the desired numbers in vector registers and using reduction OpenMP directives. The total number of vertices inside the frontier is calculated as a sum of 3 obtained values for different groups. Consequently, each group is classified as sparse or dense based on the obtained numbers. If all three groups are dense, the frontier is considered to have a “dense” type, otherwise—“mixed.” Finally, for each sparse group a parallel *copy_if* operation is invoked, which generates a list of vertex IDs inside the group.

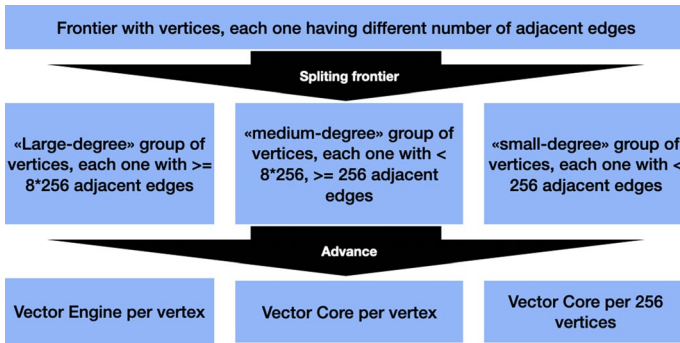
Efficiently implementing the *copy_if* operation on NEC SX-Aurora TSUBASA poses a significant challenge. For the NVIDIA GPU architecture, this operation is usually based on parallel prefix sum algorithm [10], which, however, is poorly suitable for the SX-Aurora architecture. Therefore, the following *copy_if* algorithm is used in VGL. First, each vector core allocates temporary buffers for each element of vector instruction. Each buffer has a $\frac{|V|}{256*8}$ size, and totally $256 * 8$ buffers are allocated. Next, all vector indices with nonzero flags are copied into the buffers in a single traversal through array of flags. Data are copied to buffers via scatter operations to the desired positions inside buffers, stored in a special vector register. Next, each vector core calculates the number of elements inside its vector buffers, and then all vector cores exchange obtained numbers in order to calculate offsets for each core inside the output array. Finally, each vector core copies elements from vector buffers to the output array starting from previously calculated offsets, thus generating a list of vertex IDs, which belong to the new frontier. The efficiency and the sustained bandwidth of two main parts of the *generate_new_frontier* abstraction are provided in Table 1. For comparison, Table 1 also provides the sustained memory bandwidth and the efficiency of the STREAM benchmark.

5.4 Advance

The *advance* abstraction input consists of a preprocessed graph (in VectCSR format) and a frontier, both of which have their vertices sorted and split into three groups based on their degree. In addition, each group of frontier vertices has its own sparsity characteristic, determined during frontier updates. The *advance* abstraction implements three types of handler functions, each one aimed at processing vertices of each separate group using different techniques for parallel inter-core workload balancing and using vector instructions (Fig. 3). Additionally, each handler function is optimized to work with three different representations of the input frontier: all-active, dense, and sparse.

Table 1 The sustained bandwidth and the efficiency of different parts of the *generate new frontier* abstraction

Algorithm	Sustained bandwidth (GB/s)	Efficiency (% of peak bandwidth)
VGL copy_if	261	21
Generate flags array	667	55
STREAM benchmark	983	81

**Fig. 3** Load balancing strategy used in VGL *advance* abstraction

Each “high-degree” vertex from the first group is processed using entire SX-Aurora vector engine. All eight vector cores process adjacent edges of each vertex using vector instructions of the 256 vector length, applying a user-defined *edge_op* operation to each adjacent edge. This edge traversal is implemented via simultaneously parallelized and vectorized loop. Vertex pre/postprocessing operations are executed on master thread before/after edge processing loop without vectorization.

Each “medium-degree” vertex from the second group is processed using a single vector core, which processes 256 adjacent edges at a time using vector instructions. Edge traversal of each vertex is implemented as a vectorized loop, while parallelization between different cores is implemented among different vertices.

Processing “small-degree” vertices in VGL is implemented in a fundamentally different way. Due to the low degree of each vertex in this group, each vertex is processed by one element of the vector instruction. Thus, each vector core processes 256 consecutive vertices of the input frontier. Due to the fact that frontier vertices are preliminary sorted, for most real-world graphs, each vector instruction processes approximately the same number of adjacent edges. Processing only the vertices, which belong to the input frontier for the dense cases is implemented via vector masking: Each vector instruction processes 256 consecutive vertices of graph (not frontier), deactivating those which have frontier flags set to false value. In addition, vector extension data structure (Fig. 2) is used when working with dense frontiers, which allows to load information about graph edges (destination IDs and weights) with a sequential memory access pattern, using LOAD vector instructions instead of GATHER, thus maximizing effective memory bandwidth during graph traversal. When working with sparse frontiers, the information about adjacent edges is loaded

using vector GATHER instructions from the CSR graph representation. This pattern is significantly less effective; however, it allows to load only the information required for processing specific frontier vertices, while masked vector loads on SX-Aurora are implemented so they load a significant amount of excessive data from memory if input frontier is sparse. The efficiency and the sustained bandwidth of the *advance* are provided in Table 2.

In the *advance* abstraction, the following optimizations have been used.

Parallel workload balancing Parallel workload balancing between different vector cores in VGL is implemented via the OpenMP *schedule (static, 8)* clause. Using static workload balancing in VGL is possible due to the preliminary applied clusterization [27] (vertices in VectCSR graph storage format are sorted according to their degree). Thus, 8 SX-Aurora vector cores always process 64 consecutive graph vertices, which usually have approximately the same number of adjacent edges, since all graph vertices are sorted. This allows all vector cores to process approximately the same number of edges even for highly irregular real-world graphs, as shown in Table 3. Consequently, all OpenMP threads have similar execution times, as shown in Table 4 (“min/max time among all threads”).

Scheduling mode theoretically could be changed to *(static, 1)* in order to even further improve parallel efficiency. However, in this case vector cores start processing edges, located in consecutive memory regions. Such memory access pattern is significantly less efficient for the NEC SX architecture, which results in *(static, 1)* demonstrating significantly lower performance. However, *(static, 1)* is still used in VGL when working with very sparse frontiers. In this case, vertices processed by different vector cores are usually located far away from each other (since the frontier is sparse), increasing the efficiency of memory access pattern.

Using vector instructions with the maximum vector length The performance (including the effective memory bandwidth) of SX-Aurora vector instructions decreases in proportion to their length; thus, using vector instructions with a vector length of 256 is highly desired. Processing graph vertices in separate groups allows using vector instructions of length close to 256 (Table 4 “average vector length”).

Table 2 The sustained bandwidth and the efficiency of different parts of the *advance* abstraction, applied to RMAT graphs with scale 23

Graph problem	“Medium-degree” group		“Small-degree” group	
	Sustained bandwidth (GB/s)	Efficiency (%)	Sustained bandwidth (GB/s)	Efficiency (%)
BFS	321	26	384	32
SSSP	505	42	589	49
PR	524	43	428	35
CC	219	18	543	45

Table 3 Percent of graph edges, processed by each vector core during one iteration of PageRank algorithm

Graph	Core 1 (%)	Core 2 (%)	Core 3 (%)	Core 4 (%)	Core 5 (%)	Core 6 (%)	Core 7 (%)	Core 8 (%)
Rmat (scale 21)	13.16	12.8	12.72	12.68	12.6	12.18	11.93	11.88
Live journal	12.93	12.72	12.61	12.51	12.42	12.34	12.26	12.16
Pokec	13.40	12.94	12.71	12.49	12.33	12.17	12.03	11.89

Each edges is processed only once, all graph edges are processed

Table 4 Main profiling metrics collected for the *advance* abstraction on several real-world graphs

Graph	Min time among all threads (ms)	Max time among all threads (ms)	Average vector length	Vector op. ratio (%)	LLC hit rate (%)
Rmat (scale 21)	172	192	250.2	98.9	56
Wiki_ru	49	50	253.9	99	52
Uniform-random (scale 24)	1106	1110	245.8	99	40
Pokec	78	81	255.7	99.1	61

Efficiently vectorizing different user-defined operations In order to vectorize vertex and edge traversals, specialized nc++ compiler directives are used, which indicate the absence of data dependencies within user-defined operations: *ivdep*, *overtake*, *novob*, *gather_reorder*. These directives allow to avoid partially vectorized loops (Table 4 “vector op. ratio”), as well as enable both types of indirect memory accesses—gather and scatter.

Improving last-level cache (LLC) usage For power-law graphs, the *advance* abstraction efficiently caches most of indirect memory accesses in LLC due to clusterization [27] optimization included into VectCSR graph representation (Table 4 “LLC hit rate”). In addition, VGL provides *advance* interface, which allows to directly prefetch specific indirectly accessed arrays into LLC cache using the nc++ prefetch directives.

Push/pull traversal The *advance* abstraction supports efficient implementations of both push- and pull-based graph algorithms. Generally, pull-based algorithms are more efficient on SX-Aurora, since gather vector instructions, implementing indirect memory loads, have better performance compared to scatter instructions, which implement indirect stores (Fig. 4 left).

Packing indirectly accessed 4-byte values into 8-byte On the SX-Aurora TSUB-ASA architecture, indirect memory accesses to 8-byte values are approximately twice more faster compared to accesses to 4-byte data due to the scatter/gather

vector instructions implementation (Fig. 4 right). Thus, for many graph algorithms it can be beneficial to pack 4-byte values into 8-byte, if two different 4-byte values are loaded per graph edge: levels and parents in BFS algorithm, page ranks, and outgoing degrees for PR algorithm. VGL provides a special API to perform such packing/unpacking operations in *compute* and *advance* abstractions.

6 Implementation of graph problems and algorithms using VGL

Using VGL abstractions, we implemented several algorithms aimed at solving the following fundamental graph problems: shortest paths, page rank, connected components, and breadth-first search (Fig. 5). These problems are typically implemented in most existing graph processing frameworks and libraries and thus can be used for the comparative performance analysis, which will be provided in the next section.

Breadth-first search. The BFS algorithms operate with frontiers of vertices—concept, natively implemented using the VGL data abstractions. The initial frontier is set to contain only a source vertex, and in each iteration, a new frontier of vertices is generated from all unvisited vertices, which are neighbours of the previous frontier. In VGL, both direction-optimizing [4] and top-down BFS algorithms are implemented.

Shortest paths. The single-source shortest paths problem involves finding paths between a given source vertex and all other graph vertices, such that all weights on the path between source and destination vertices are minimized. Multiple parallel shortest paths algorithms exist, including the Bellman–Ford [9] and the delta stepping [16]. In VGL, push-based and pull-based versions of the Bellman–Ford

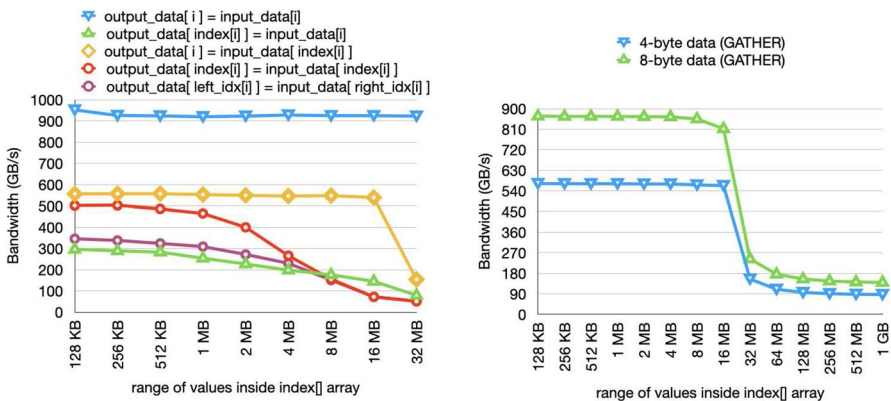


Fig. 4 Comparative bandwidth values for various scatter/gather benchmarks (left) on SX-Aurora architecture. Comparative bandwidth values for benchmarks, indirectly accessing 4-byte and 8-byte data (right)

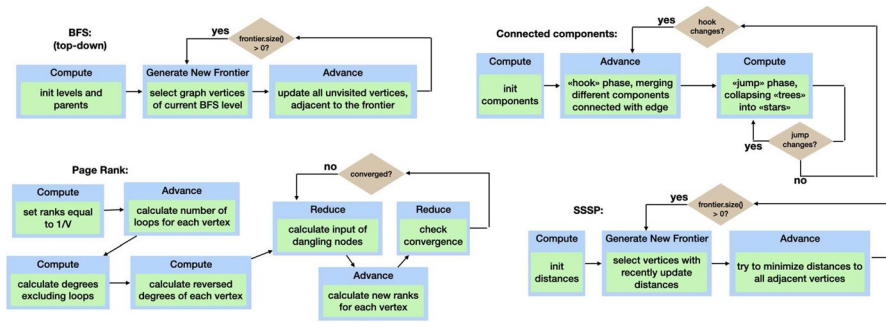


Fig. 5 Operation flowcharts for selected graph problems implemented via VGL abstractions

algorithm are implemented, where in each iteration all graph vertices and their adjacent edges are used for updating paths. In addition, VGL implements a more computationally optimal version of the Bellman–Ford algorithm, when only vertices with recently updated labels participate in computations in each iteration.

Page rank. The page rank [19] algorithm assigns a numerical weighting to each element of a hyperlinked set of documents (e.g., web graph) with the purpose of quantifying its relative importance within the set. In VGL, the pull-based page rank algorithm is implemented, since it allows to avoid using atomic operations, which are very inefficient on the SX-Aurora architecture. During each iteration of the page rank algorithm, all graph vertices participate in calculations; thus, "all-active" frontier API part is used.

Connected Components. The connected component problem involves labeling graph vertices using unique component IDs. VGL connected components implementation is based on Shiloach–Vishkin [20] and bfs-based algorithms.

7 Performance evaluation

The performance of VGL-based implementations has been evaluated in comparison with other existing frameworks and libraries for multicore CPUs and NVIDIA GPUs. We ran all the experiments on the cluster equipped with: (1) 12-core Intel(R) Xeon(R) Gold 6126 CPU of Intel Skylake architecture, (2) NVIDIA V100 GPU of Volta architecture, and (3) vector engine SX-Aurora TSUBASA Type 10B, installed in different cluster partitions. Ligra, Galois, and GAPBS graph libraries have been used in order to evaluate the VGL performance against Intel Xeon, each of which is the latest available version at the moment of this writing. Each CPU library has been compiled using GCC version 8.3, and during execution, a number of threads equal to the number of Intel Xeon cores have been used. Gunrock, cuSHA and Enterprise frameworks have been used in order to evaluate VGL performance against NVIDIA GPUs, as well as NVGRAPH and Lonestar GPU libraries, built using GCC v8.3 and

NVIDIA CUDA Toolkit v10.2. VGL-based implementations have been compiled using nc++ of version 3.0.6.

Graphs used in our experiments include synthetic RMAT [6] and multiple real-world graphs from [22, 23] collections. Main characteristics of several graphs are provided in Table 5. For each implementation, exactly the same synthetic graphs have been used, externally loaded into each framework. Mega Traversed Edges Per Second (MTEPS) [17] has been used as the main performance evaluation metric.

The comparative VGL performance analysis is demonstrated in Figs. 7, 6, 8 and 9. Provided performance results have been obtained using the following methodology: For each graph problem (PR, SSSP, BFS, CC), the VGL-based implementation performance has been compared with two fastest multicore CPU and NVIDIA GPU implementations, available for each problem. This is the main reason why some frameworks (such as cuSHA) are not present in the figures below—their performance is significantly lower compared to other frameworks and libraries. In order to exclude possible differences in performance caused by different computational complexity of algorithms being used in different frameworks, only the same algorithms (or minor variations of the same algorithm) have been compared.

The first important thing to observe VGL achieves up to 14 times acceleration compared to multicore CPU implementations. Such a significant performance difference is caused by different peak memory bandwidth values for these platforms: 90 GB/s for Intel Xeon versus 1.2 TB/s for NEC SX-Aurora TSUBASA. The fact that the performance difference is approximately proportional to bandwidths values proves our observation expressed in the beginning of the paper about significant potential of using systems with high-bandwidth memory for graph applications.

The comparison of the VGL performance with implementations targeting V100 GPU is more fair, since both systems have approximately the same theoretical memory bandwidth: 900 GB/s for V100 GPU versus 1.2 TB/s for NEC SX-Aurora TSUBASA. However, in most cases (BFS, PR, SSSP problems) we can observe up to 3x better performance of the VGL implementations. Such acceleration can be explained by the combination of the following factors. First, SX-Aurora TSUBASA has a slightly higher theoretical memory bandwidth, which, of course, contributes to the performance difference. Second, VGL uses graph preprocessing techniques (clusterization), which allows to significantly increase LLC hit rate when processing indirect memory accesses and, in addition, improves parallel workload balancing.

Table 5 Main properties and characteristics of graphs used during performance evaluation

Graph name	Vertices count	Edges count	Type	Degree-distribution	Maximum vertex degree
Rmat (scale 21)	2 M	67 M	Synthetic	Power law	137610
Rmat (scale 26)	67 M	2.14 bn	Synthetic	Power law	1.1 M
Friendster	67 M	2.5 bn	Web	Power law	5214
Twitter	41 M	1.4 bn	Social	Power law	2.9 M
Wiki_en	12 M	378 M	Web	Power law	7888

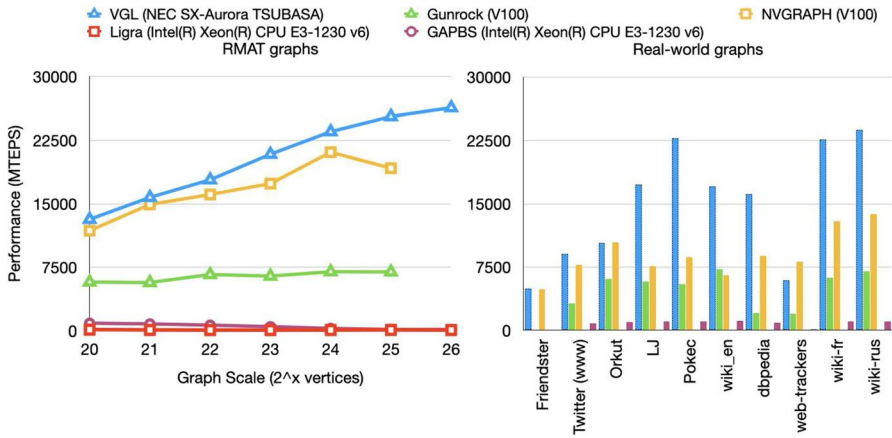


Fig. 6 The performance (per iteration) of VGL page rank implementation compared to four fastest multicore CPU and NVIDIA GPU frameworks and libraries: NVGRAPH, Ligra, GAPBS, Gunrock. Other frameworks and their fastest available page rank method (including push-based)

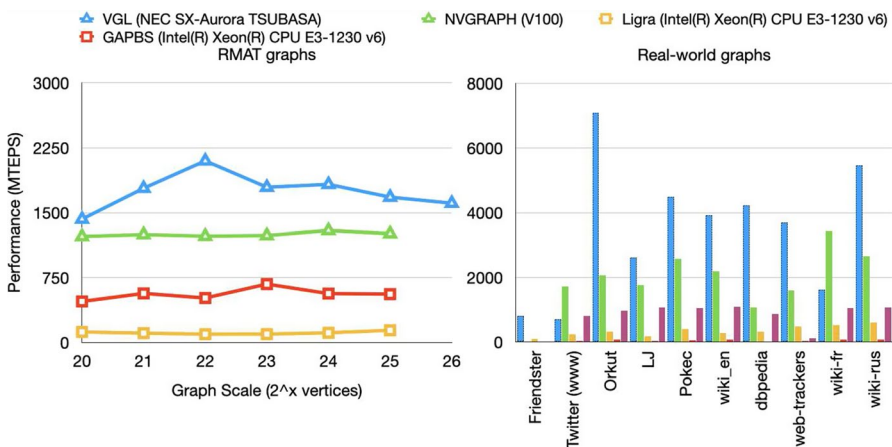


Fig. 7 The performance of VGL shortest paths implementation compared to three fastest multicore CPU and NVIDIA GPU frameworks and libraries: NVGRAPH, Ligra, GAPBS

Other frameworks, such as Gunrock, do not use preprocessing-based optimizations. Finally, SX-Aurora TSUBASA has a significantly larger LLC cache (compared to NVIDIA GPUs), which allows to store a significant part of most frequently accessed graph vertices, since all the graphs used for the performance evaluation are scale-free. Unfortunately, it is hard to conclude which factor provides a higher contribution to the performance increase. On the final note, for the connected components problem Gunrock demonstrates better performance for multiple real-world graphs (Fig. 9), most probably due to the harmful for SX-Aurora TSUBASA memory access patterns during the “jump” phase of Shiloach–Vishkin algorithm.

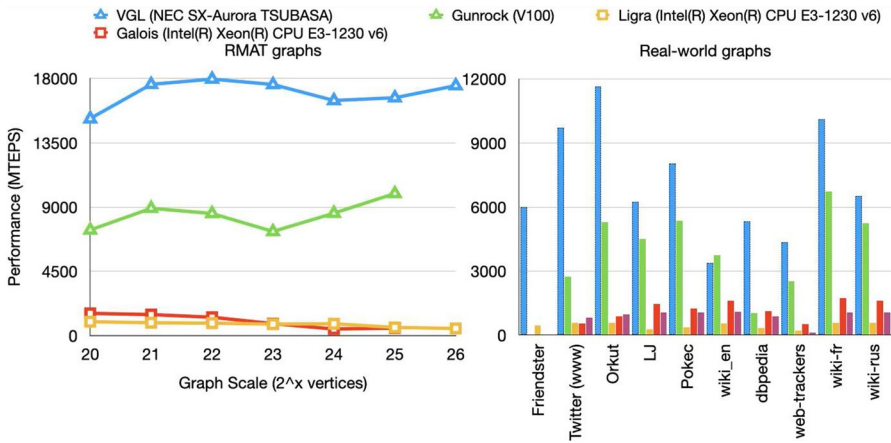


Fig. 8 The performance of VGL breadth-first search implementation compared to three fastest multicore CPU and NVIDIA GPU frameworks and libraries: Gunrock, Ligma, GAPBS

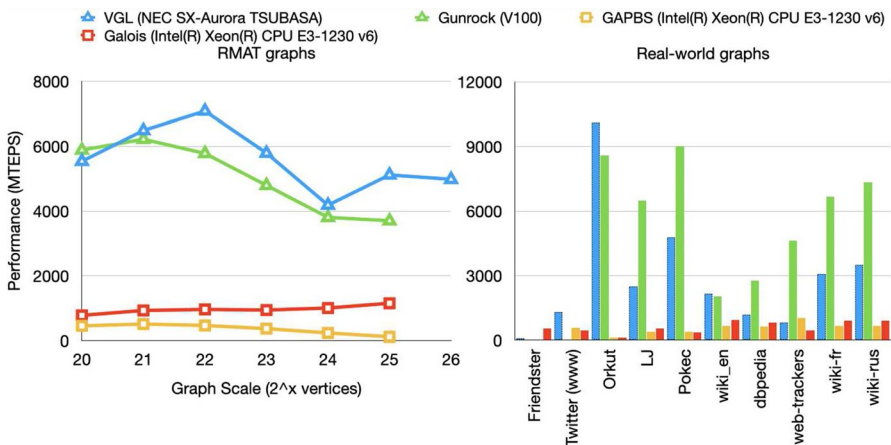


Fig. 9 The performance of VGL connected components implementation compared to three fastest multicore CPU and NVIDIA GPU frameworks and libraries: Gunrock, Ligma, GAPBS

8 Future plans

Our future plans include the following main research and development directions.

1. Extending the list of graph algorithms implemented in the VGL. Currently, in addition the algorithms discussed in this paper, algorithms for solving maximum flow, community detection, widest paths, strongly connected components and all-pairs shortest paths problems are implemented.
2. Implementing the possibility of using multiple SX-Aurora vector engines for graph processing. This direction is motivated by the fact that supercomputer

nodes based on SX-Aurora TSUBASA architecture are equipped with up to eight vector engines, using all of which can potentially even further accelerate graph processing.

3. Extending the VGL framework to other systems with high-bandwidth memory and vector-processing features, such as Intel KNL, A64FX (vector extensions), and NVIDIA GPUs (warps). According to our experience, such systems require similar optimization approaches (using specific memory access patterns, SIMD instructions of maximum available length, workload balancing), which need to be used in order to efficiently utilize high-bandwidth memory.

9 Conclusions

In this paper, we presented the world first attempt to develop a high-level programmable graph processing framework for modern NEC SX-Aurora TSUBASA vector architecture. NEC SX-Aurora TSUBASA is equipped with memory of a 1.2TB/s bandwidth, which allows to drastically accelerate various graph algorithms, if they are accurately implemented. In this paper, we discussed the VGL computational and data abstractions, as well as their implementation details and possible applications.

As was shown in this paper, VGL allows to solve a number of fundamental graph problems, including SSSP, BFS, PR, and CC. The VGL-based implementations of these problems demonstrate a significant acceleration compared to existing most advanced frameworks and libraries, developed for other platforms. For example, the VGL-based implementations are up to 14 times faster compared to Ligra, Galois and GAPBS multicore CPU frameworks and libraries, and up to 3 times faster compared to Gunrock and NVGRAPH implementations for various synthetic and real-world graphs. Finally, due to the 48 GB of high-bandwidth memory available in the SX-Aurora architecture, VGL is capable of processing relatively large datasets including Twitter and Friendster social graphs—the largest available in KONECT [23] and SNAP [22] collections.

Acknowledgements The results described in Section 5 were obtained in Lomonosov Moscow State University with the financial support of the Russian Science Foundation (Agreement N 20-11-20194). The reported study was funded by RFBR, Project Number 19-37-90002.

References


1. Afanasyev I, Voevodin VV, Voevodin VV, Komatsu K, Kobayashi H (2019) Developing efficient implementations of shortest paths and page rank algorithms for NEC SX-Aurora TSUBASA architecture. *Lobachevskii J Math* 40(11):1753–1762
2. Afanasyev IV, Antonov AS, Nikitenko DA, Voevodin VV, Voevodin VV, Komatsu K, Watanabe O, Musa A, Kobayashi H (2018) Developing efficient implementations of bellman-ford and forward-backward graph algorithms for nec sx-ace. *Supercomput Front Innov* 5(3):65–69

3. Afanasyev IV, Voevodin VV, Voevodin VV, Komatsu K, Kobayashi H (2019) Analysis of relationship between simd-processing features used in nvidia gpus and nec sx-aurora tsubasa vector processors. In: *International Conference on Parallel Computing Technologies*. Springer, pp 125–139
4. Beamer S, AsanoviÄ‡ K, Patterson D (2013) Direction-optimizing breadth-first search. *Sci Program* 21(3–4):137–148
5. Besta M, Podstawski M, Groner L, Solomonik E, Hoefer T (2017) To push or to pull: On reducing communication and synchronization in graph computations. In: *Proceedings of the 26th international symposium on high-performance parallel and distributed computing*, pp 93–104
6. Chakrabarti D, Zhan Y, Faloutsos C (2004) R-mat: a recursive model for graph mining. In: *Proceedings of the 2004 SIAM International Conference on Data Mining*. SIAM, pp 442–446
7. Egawa R, Komatsu K, Momose S, Isobe Y, Musa A, Takizawa H, Kobayashi H (2017) Potential of a modern vector supercomputer for practical applications: performance evaluation of SX-ACE. pp 3948–3976
8. Fu Z, Personick M, Thompson B (2014) Mapgraph: a high level api for fast development of high performance graph analytics on gpus. In: *Proceedings of workshop on GRAPh data management experiences and systems*, pp 1–6
9. Goldberg A, Radzik T (1993) A heuristic improvement of the bellman-ford algorithm. Stanford Univ CA Dept of Computer Science, Technical report
10. Hillis WD, Steele GL Jr (1986) Data parallel algorithms. *Commun ACM* 29(12):1170–1183
11. Ilic A, Pratas F, Sousa L (2013) Cache-aware roofline model: upgrading the loft. *IEEE Comput Archit Lett* 13(1):21–24
12. Khorasani F, Vora K, Gupta R, Bhuyan LN (2014) Cusha: vertex-centric graph processing on gpus. In: *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*, pp 239–252
13. Komatsu K, Egawa R, Isobe Y, Ogata R, Takizawa H, Kobayashi H (2015) An approach to the highest efficiency of the HPCG benchmark on the SX-ACE supercomputer. In: *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC15)*. Poster, pp 1–2
14. Komatsu K, Momose S, Isobe Y, Watanabe O, Musa A, Yokokawa M, Aoyama T, Sato M, Kobayashi H (2018) Performance evaluation of a vector supercomputer sx-aurora tsubasa. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, SC '18*. IEEE Press, Piscataway, pp 54:1–54:12
15. Liu H, Huang HH (2015) Enterprise: breadth-first graph traversal on gpus. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp 1–12
16. Meyer U, Sanders P (2003) Δ -stepping: a parallelizable shortest path algorithm. *J Algorithms* 49(1):114–152
17. Murphy RC, Wheeler KB, Barrett BW, Ang JA (2010) Introducing the graph 500. *Cray Users Group (CUG)* 19:45–74
18. Nguyen D, Lenharth A, Pingali K (2013) A lightweight infrastructure for graph analytics. In: *Proceedings of the twenty-fourth ACM symposium on operating systems principles*, pp 456–471
19. Page L, Brin S, Motwani R, Winograd T (1999) The pagerank citation ranking: bringing order to the web. Technical report, Stanford InfoLab
20. Shiloach Y, Vishkin U (1980) An $o(\log n)$ parallel connectivity algorithm. Technical report, Computer Science Department, Technion
21. Shun J, Blelloch GE (2013) Ligra: a lightweight graph processing framework for shared memory. In: *ACM sigplan notices*, vol. 48. ACM, pp 135–146
22. Stanford Large Network Dataset Collection-SNAP. <https://snap.stanford.edu/data/>
23. The Koblenz Network Collection-KONECT. <http://konect.uni-koblenz.de>
24. Wang Y, Davidson A, Pan Y, Wu Y, Riffel A, Owens JD (2016) 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*, pp 1–12
25. Williams S, Waterman A, Patterson D (2009) Roofline: an insightful visual performance model for multicore architectures. *Commun ACM* 52(4):65–76

26. Yamada Y, Momose S (2018) Vector engine processor of nec brand-new supercomputer sx-aurora TSUBASA. In: Intentional symposium on high performance chips (Hot Chips2018)
27. Zhang Y, Kiriansky V, Mendis C, Amarasinghe S, Zaharia M (2017) Making caches work for graph analytics. In: 2017 IEEE International Conference on Big Data (Big Data). IEEE, pp 293–302
28. Zhong J, He B (2013) Medusa: simplified graph processing on gpus. *IEEE Trans Parallel Distrib Syst* 25(6):1543–1552

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Affiliations

Ilya V. Afanasyev¹  · Vladimir V. Voevodin² · Kazuhiko Komatsu³ · Hiroaki Kobayashi³

✉ Ilya V. Afanasyev
afanasiev_ilya@icloud.com

¹ Moscow Center of Fundamental and Applied Mathematics, Moscow, Russia 119991

² Research Computing Center of Moscow State University, Moscow, Russia 119234

³ Tohoku University, Sendai, Miyagi 980-8579, Japan