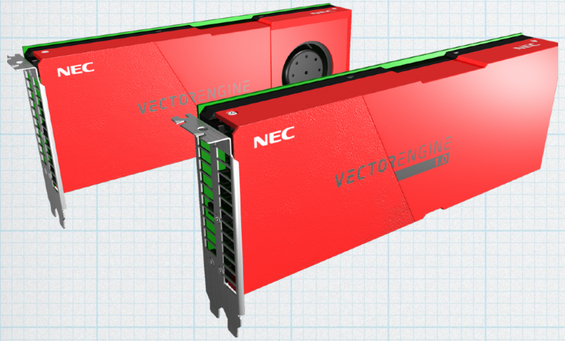
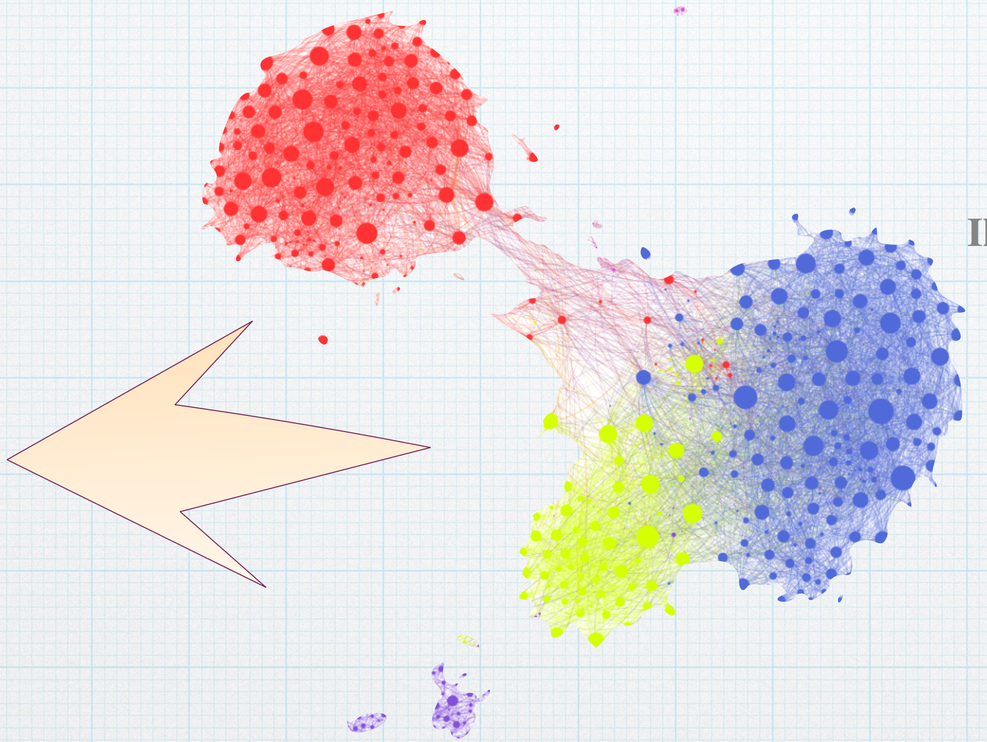
Developing a Prototype of High-Performance Graph-Processing Framework





Ilya Afanasyev, Vladimir Voevodin, Vadim Voevodin [2]

Hiroaki Kobayashi, Kazuhiko Komatsu [1]

[1] Tohoku University (Japan)

[2] Lomonosov Moscow State University (Russia)

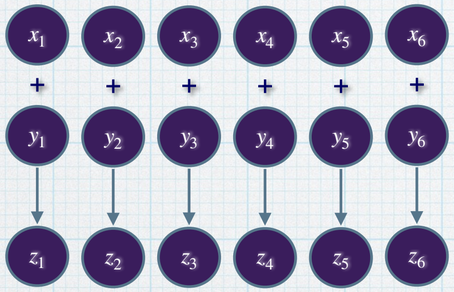
RuSCD 2020

22.09.2020

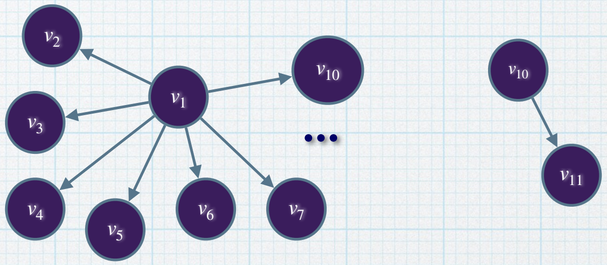
Moscow, Russia

Motivation

- Efficiently implementing large-scale graph-processing algorithms for vector systems is extremely complicated due to the irregularity of many real-world graphs



vector processing is «regular»



graphs are «irregualr»

- SX-Aurora TSUBASA provides 1.2TB/s memory bandwidth, paired with a large LLC cache, which allows to significantly accelerate many **memory-bound** applications

- We want to implement a simple, fast and efficient graph-processing framework for NEC SX-Aurora TSUBASA architecture, similar to well-known framework Gunrock or Ligra

VGL - Graph Processing Framework for vector systems

* We developed VGL (Vector Graph Library) framework — graph-processing system which targets NEC SX-Aurora architecture

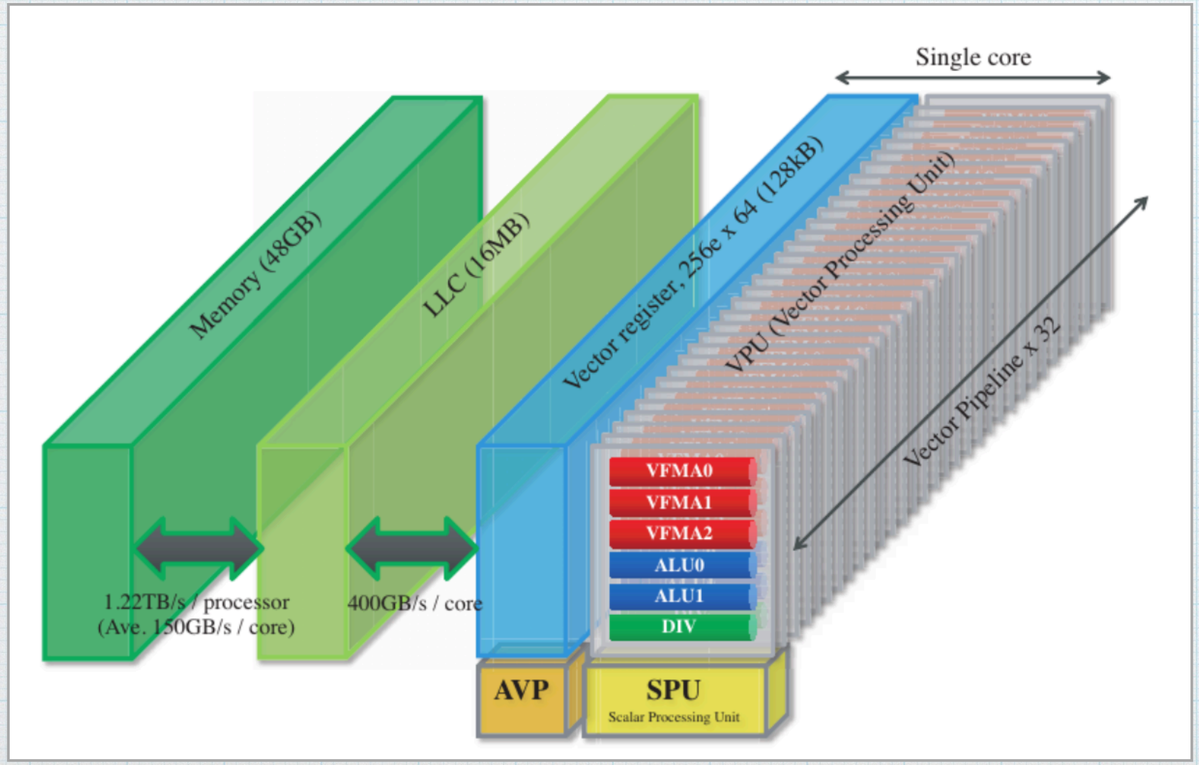
* In the future it will support other vector architectures with HBM memory (A64FX, KNL, NVIDIA GPUs)

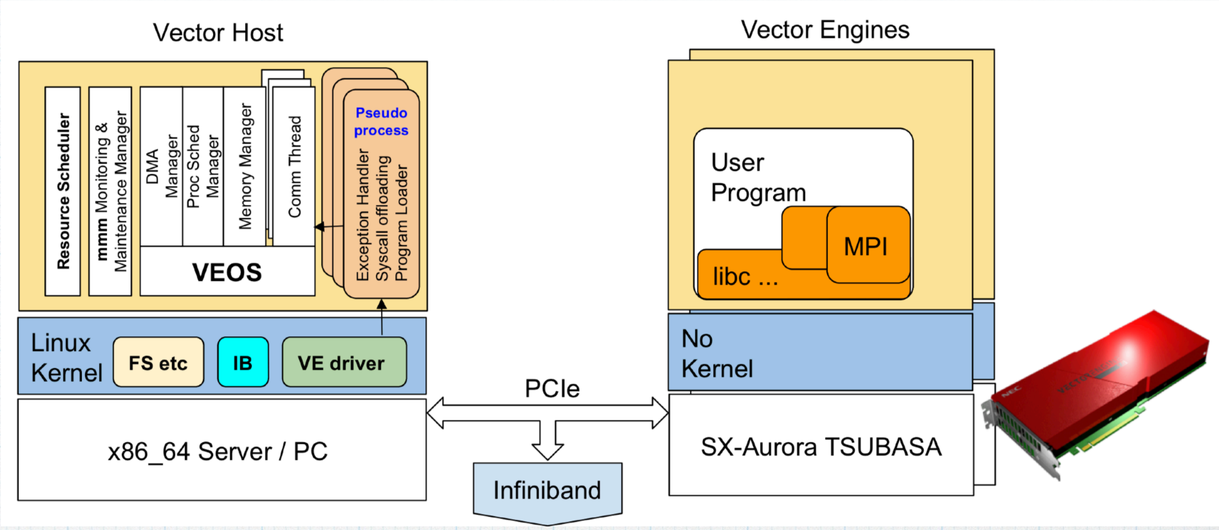
* VGL provides simultaneously efficient and simple to use API, which can be used to express various graph algorithms (similar to other frameworks for modern multicoreCPUs and GPUs)

* VGL will be available on GitHub in this month — https://github.com/afanasyev-ilya/VectorGraphLibrary

NEC SX-Aurora TSUBASA Architecture Overview

* NEC SX-Aurora TSUBASA is a dedicated vector processor of the NEC SX- architecture family
* Unlike the previous SX- computers, the SX-Aurora TSUBASA is provided as a PCIe card, and the whole system consists of vector engines (VEs), equipped with a vector processor and a vector host (VH) of an x86 node.
* VE includes 8 vector cores, 4.3 TFlop/s performance (SP)
* 6 HBM modules, 1.22 TB/s bandwidth





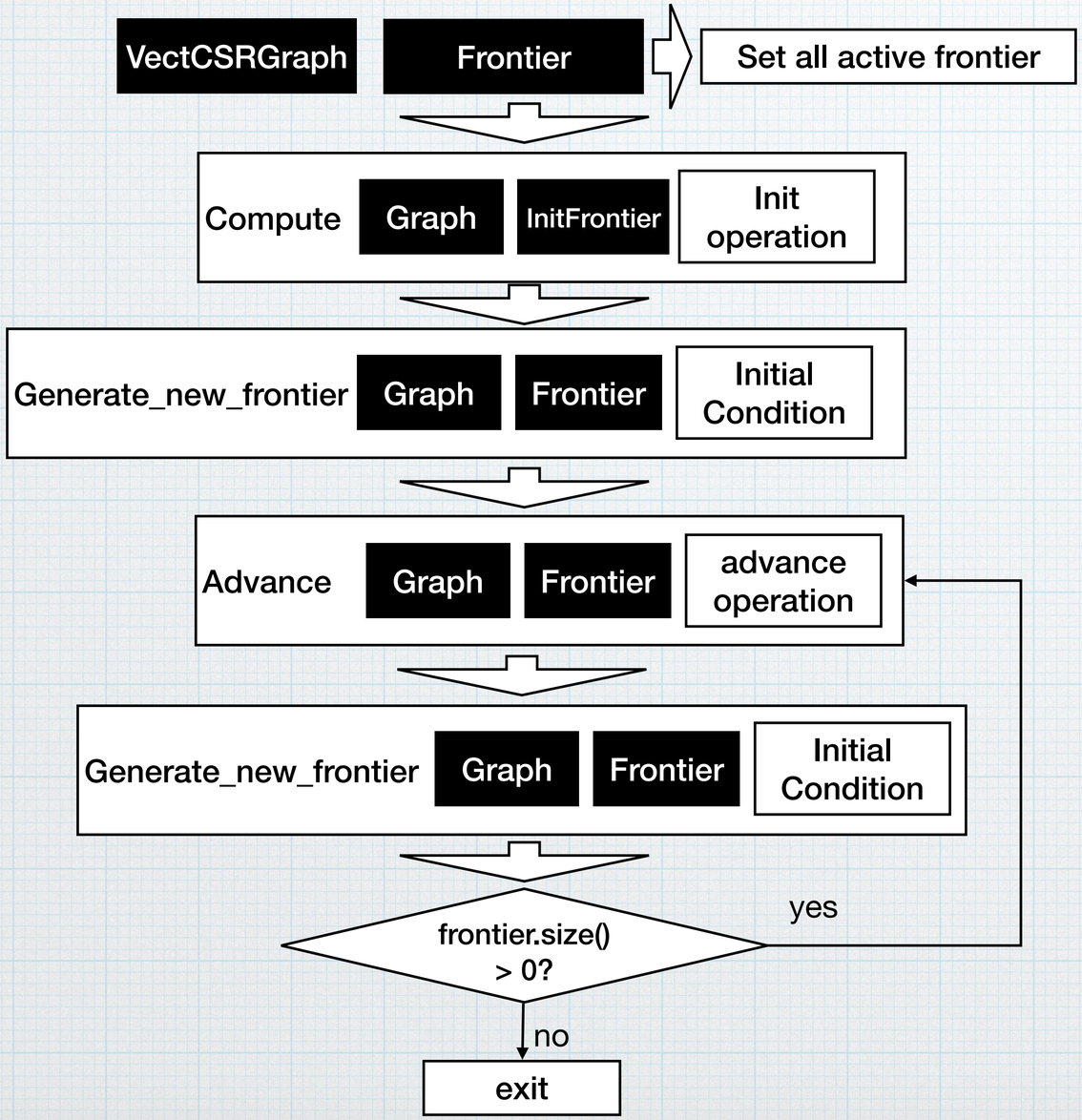
* Each vector core consists of SPU (processing scalar instructions) and VPU (processing vector instructions)
* VPUs operate with vectors of up to 256 length

VGL API

- VGL provides API for large-scale graph processing
- VGL API is **data-driven**, e.g. it defines graph algorithms in terms of computational operations (abstractions) over the following graph data-structures:
  - graph
  - frontier (a subset of graph vertices)
- VGL includes the following computational abstractions:
  - advance
  - generate_new_frontier
  - compute
  - reduce

The most challenging abstraction to implement!

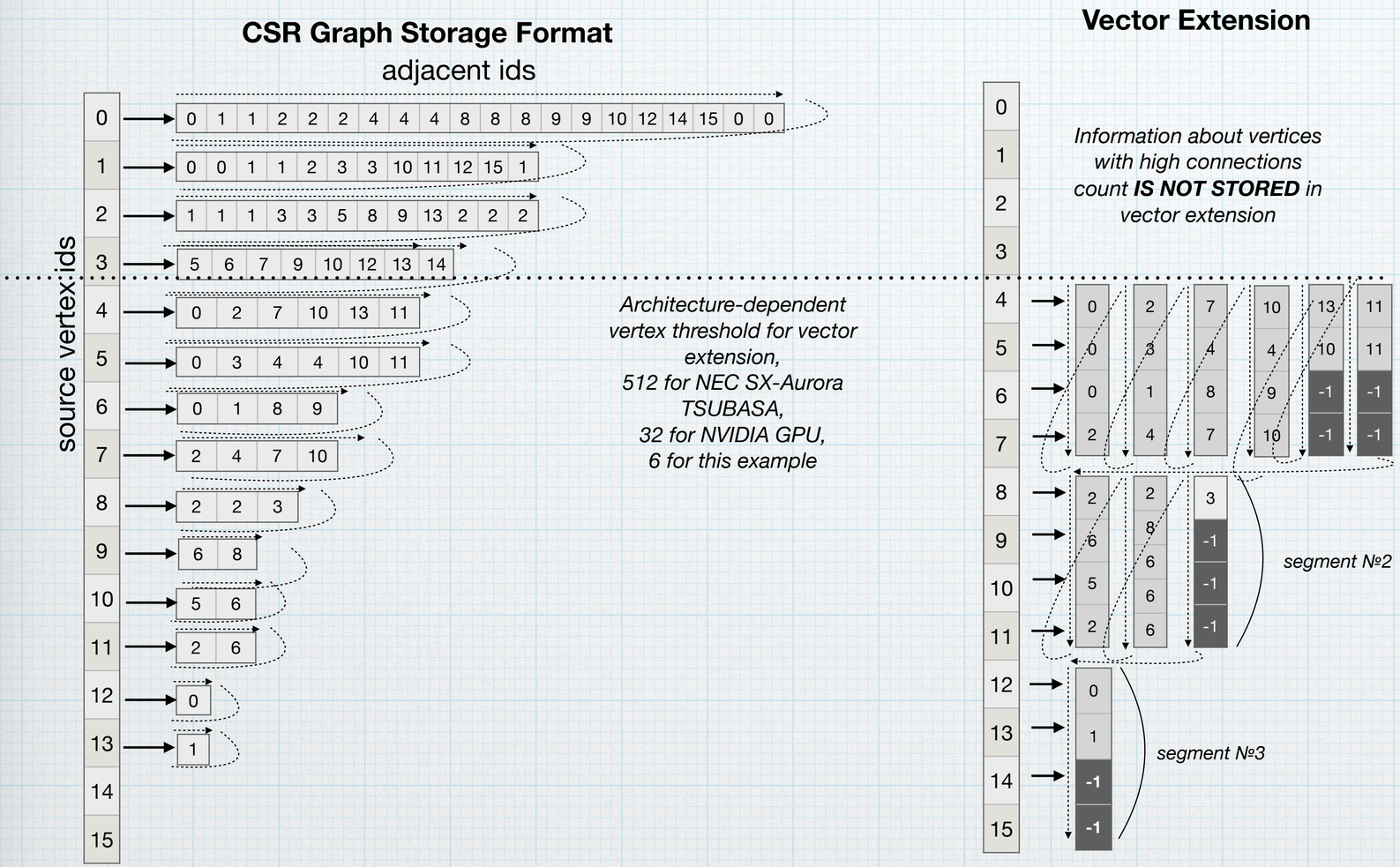
Typical workflow of VGL framework



- VGL is aimed to implement **iterative** graph algorithms (BFS, SSSP, PR, CC, and many others)
- Iterative graph algorithms process graph by iterations; on each iteration a certain **subset of graph vertices and edges** is processed
- Processing vertices and edges means applying some **computational operations**, defined by the implemented algorithm

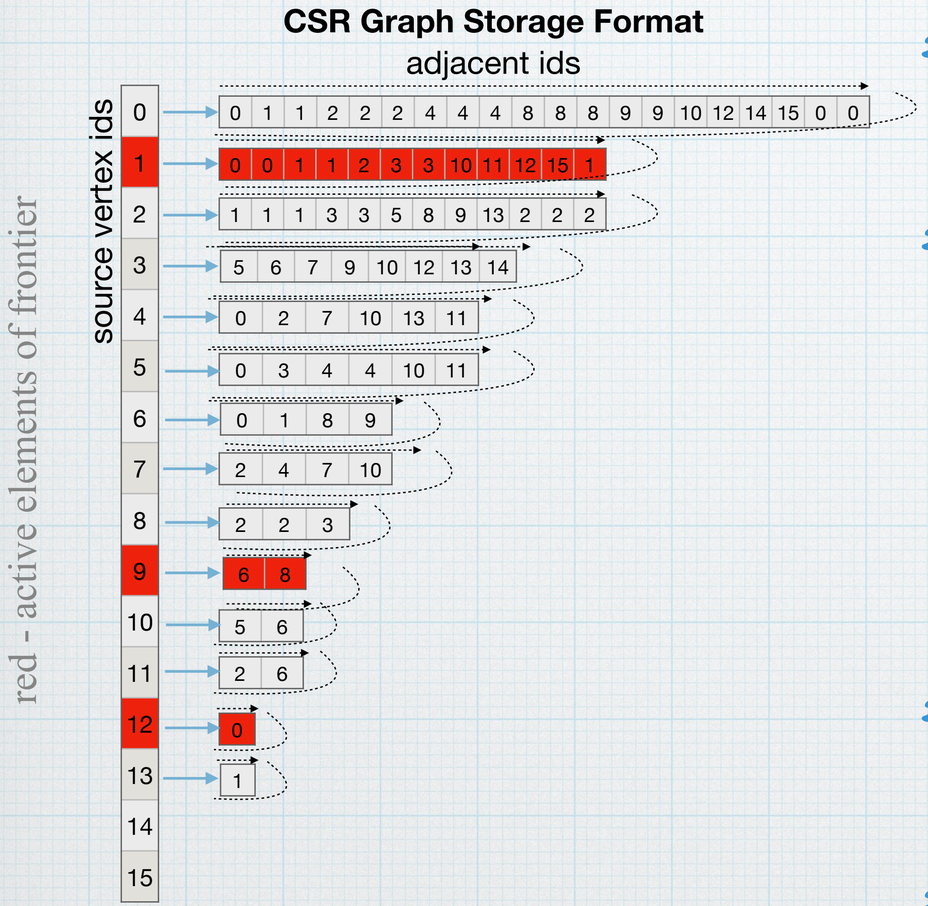
First, let's discuss VGL data-abstractions...

Data abstraction: Graph



- Graph in VGL is stored in **preprocessed** CSR format, paired with **vector extension**, which allows vector-friendly graph traversal
- Graph clusterization (and possibly CSR segmentation) is applied, which allows to efficiently use LLC cache

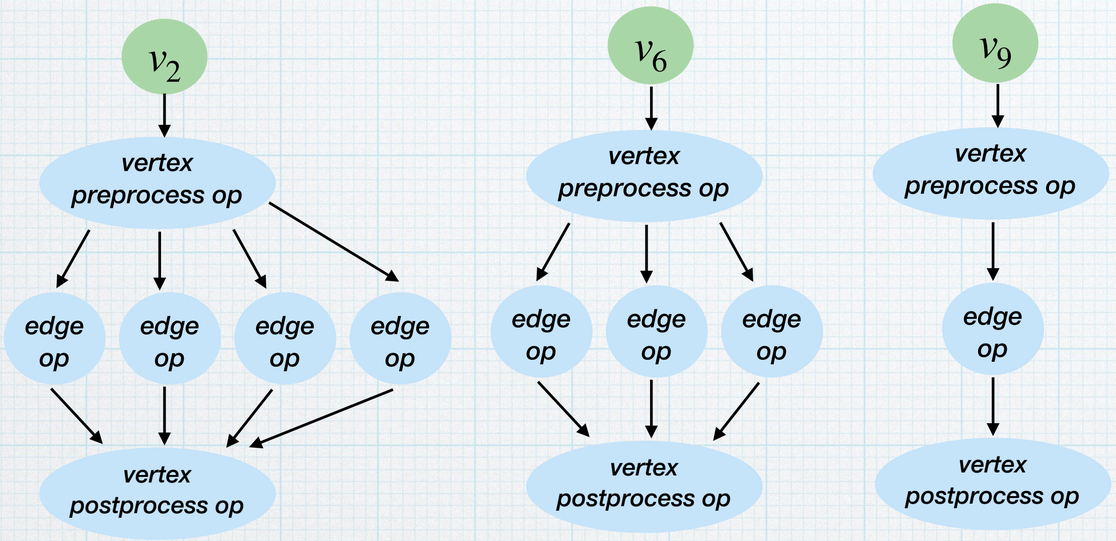
VGL data abstractions: Vertex Frontier



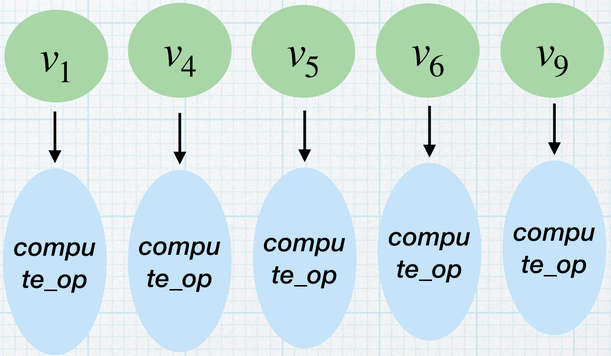
- Frontier is a user-defined subset of graph vertices
- Frontier is used when only a specific part of graph vertices has to participate in computations (for example, BFS algorithm)
- Frontier can be:
  - all-active (all graph vertices participate in computations)
  - dense (many graph vertices participate in computations)
  - mixed (only few graphs vertices participate in computations)
- Parts of «mixed» frontier is implemented as a list of vertex indexes [1, 9, 12]
- «Dense» frontier is implemented as an array of flags [0, 1, 0, 0, 0 ...]
- All-active frontier has it's own simplified implementation

VGL computational abstractions

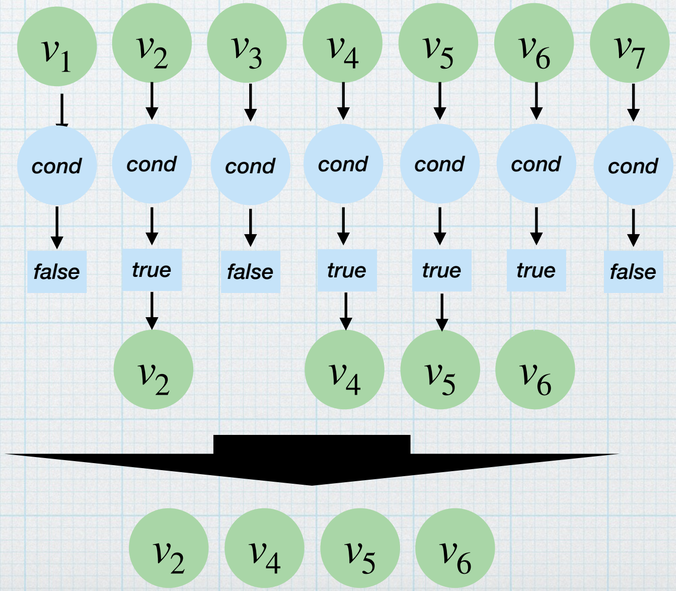
Advance



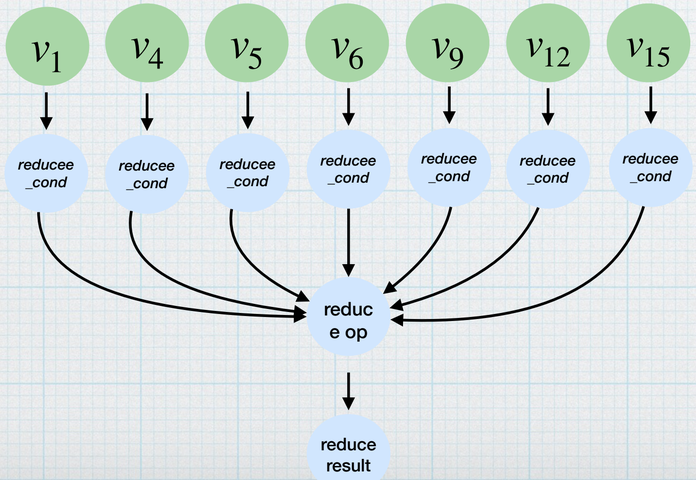
Compute



Generate New Frontier



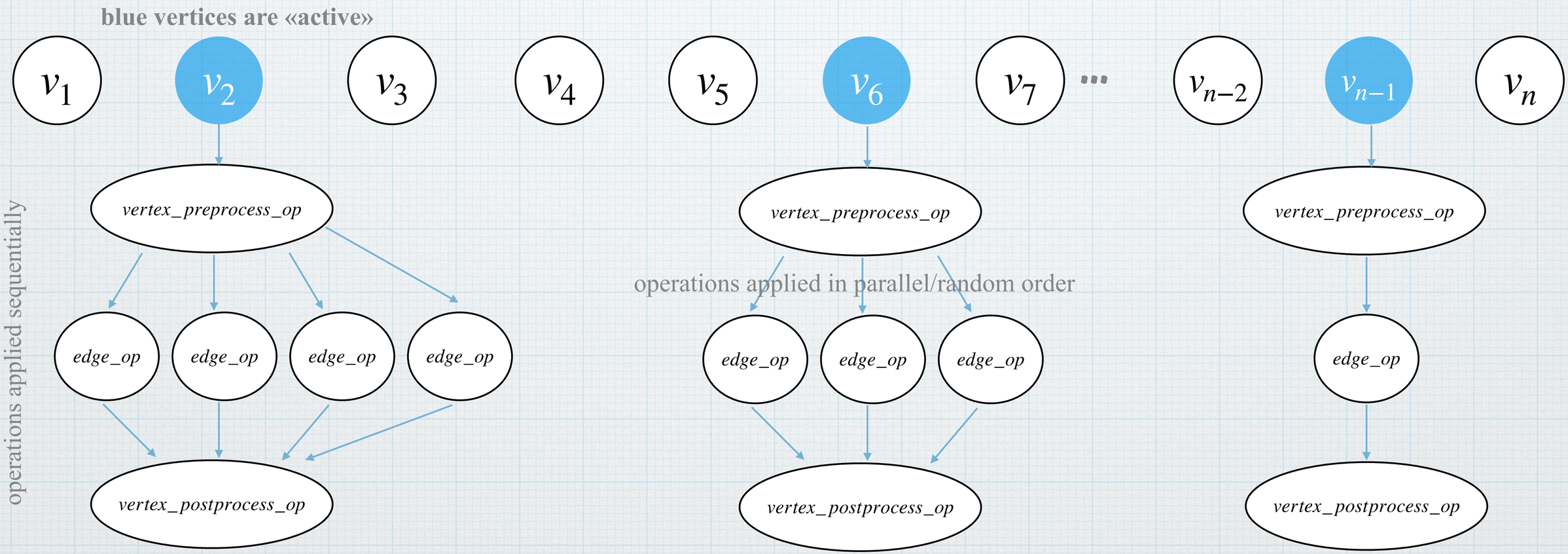
Reduce



VGL: Advance abstraction

- advance(graph, input_frontier, vertex_preprocess_op, edge_op, vertex_postprocess_op)
- user defines operations over vertices and edges using **c++ lambda functions**
- these functions are applied to frontier vertices and edges in the following way:

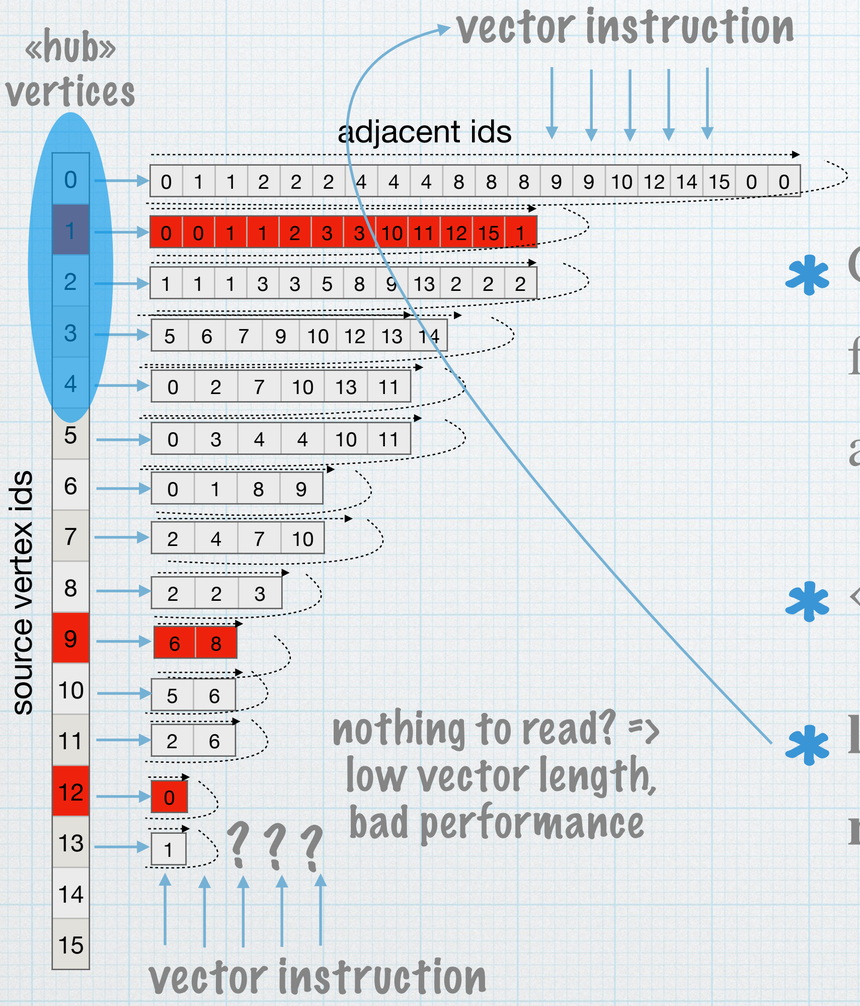
```
auto edge_op = [_levels, current_level] (...)
{
    int dst_level = _levels[dst_id];
    if(dst_level == UNVISITED_VERTEX)
    {
        _levels[dst_id] = current_level + 1;
    }
};
```



Advance implementation challenges

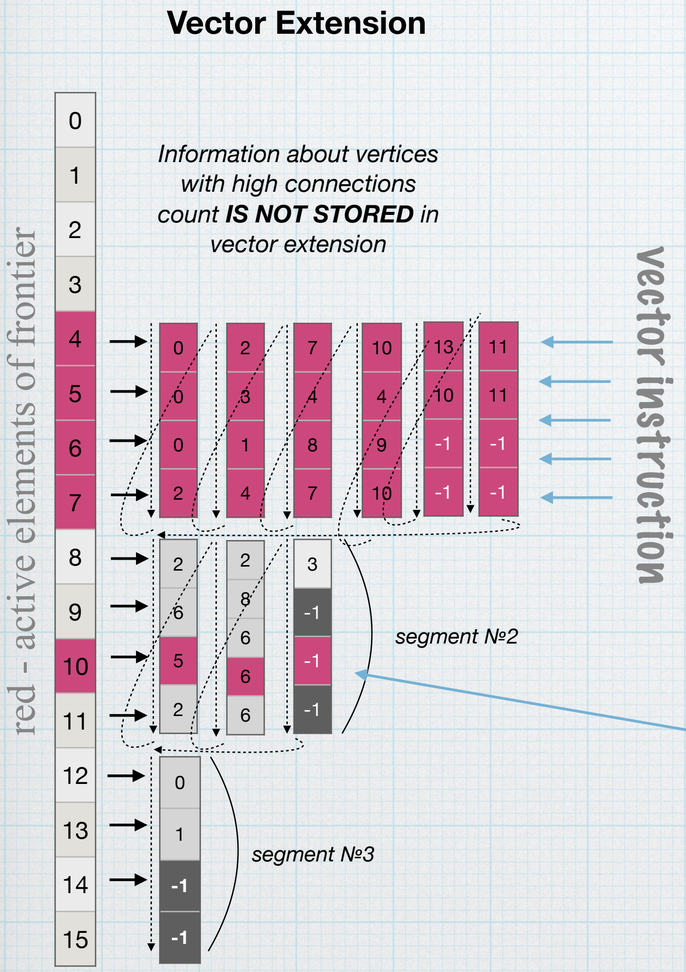
- Mapping Advance abstraction on vector architectures is extremely hard, since:
  - irregular memory accesses inside user-defined operations over edges => bad (random) memory access pattern
  - irregular structure of real-world graphs (vertex connections count) => difficult parallel workload balancing
  - hard to process low-degree vertices with vector instructions
  - necessary to work with sparse frontiers in many cases
- Thus, Advance abstraction requires to implement many vector-oriented optimizations

Advance: memory access pattern



- Graph is stored in **preprocessed** VectCSR format, and most frequently accessed vertices are **stored together (nearby)** in vertex arrays (spatial locality)
- «hub» vertices can be prefetched into LLC cache
- **loading adjacent ids is implemented so that vector instructions read consequent elements from memory**
  - however, for low degree vertices this approach can not be applied (!)

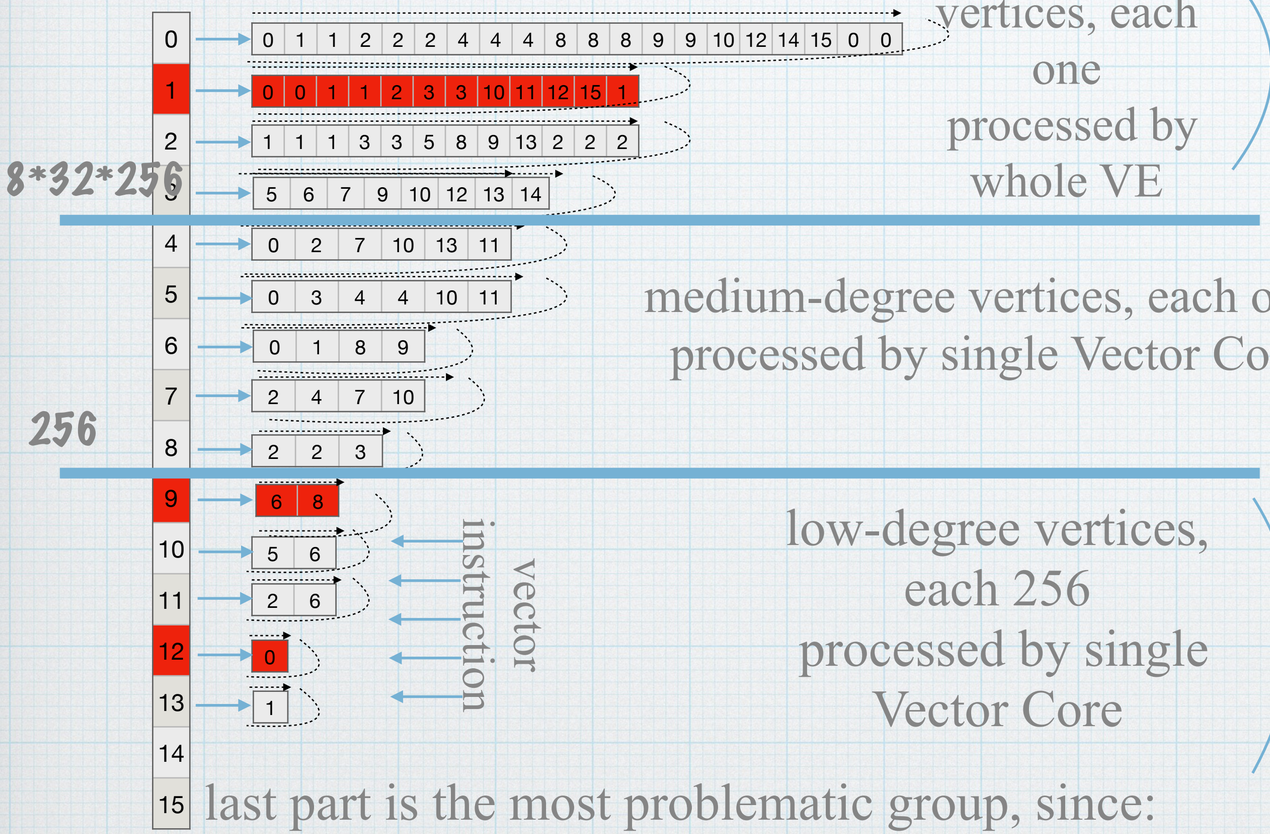
Improving memory access pattern for low-degree vertices



- We solve this problem using **vector extension** provided in VectCSR graph storage format
- main idea of vector extension is providing **good memory access pattern** for loading **information about edges** (adjacnet ids, weights) for vertices with **low-degree**, when 256 of them are processed using vector instructions, thus **maximizing vector length**
- However, this approach has drawbacks:
  - processing sparse frontiers this way is very inefficient (loading unnecessary data from memory)
  - vector extension requires additional space (up to 2x for some graphs)

Advance: parallel workload balancing

Graph is split into 3 parts, based on the number of outgoing degree of each vertex



high-degree vertices, each one processed by whole VE

medium-degree vertices, each one processed by single Vector Core

low-degree vertices, each 256 processed by single Vector Core

last part is the most problematic group, since:

- bad memory access pattern in sparse cases
- requires storing vector extension
- loading excessive data from memory in dense cases

Working with sparse frontier for first 2 parts if fast and easy!

```
if(_frontier_flags[src_id] > 0)
{
    #pragma omp for schedule(static)
    for (int local_edge_pos = 0; local_edge_pos < connections_count; local_edge_pos++)
    {
        const int dst_id = _adjacent_ids[global_edge_pos];
        edge_op(….);
    }
}
```

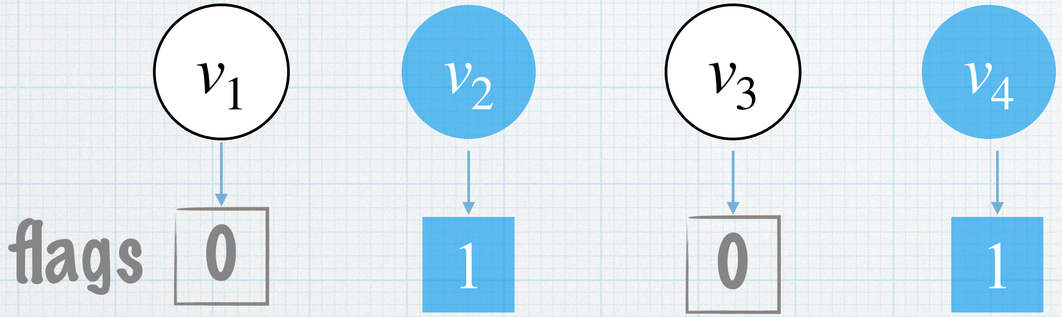
```
if(_frontier_flags[src_id] > 0)
{
    for (int local_edge_pos = 0; local_edge_pos < connections_count; local_edge_pos++)
    {
        const int dst_id = _adjacent_ids[global_edge_pos];
        edge_op(….);
    }
}
```

Working with sparse frontier for the last part is complicated…

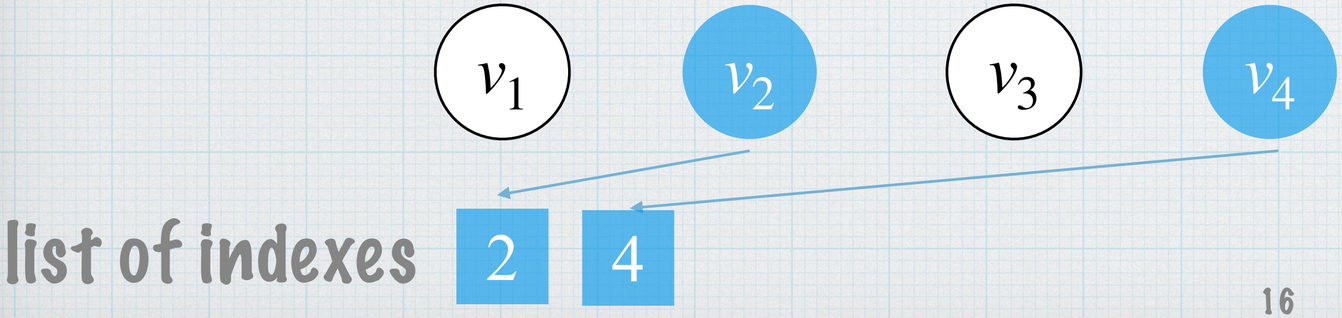
```
for(int edge_pos = _first_edge; edge_pos < segment_connections_count; edge_pos++)
{
    #pragma _NEC gather_reorder
    for (int i = 0; i < VECTOR_LENGTH; i++)
    {
        const int src_id = segment_first_vertex + i;

        if(_frontier_flags[src_id] > 0)
        {
            const int dst_id = _ve_adjacent_ids[global_edge_pos];
            edge_op(….);
        }
    }
}
```

Generate_new_frontier abstraction

* very efficient when generating dense frontier, since only array of flags required to be created (O|V| work, efficiently vectorized)

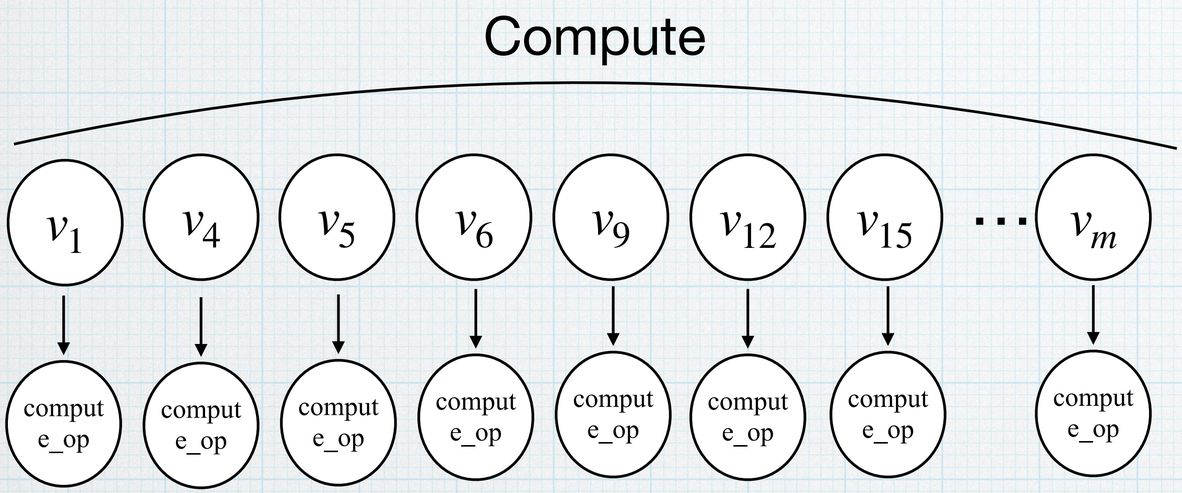


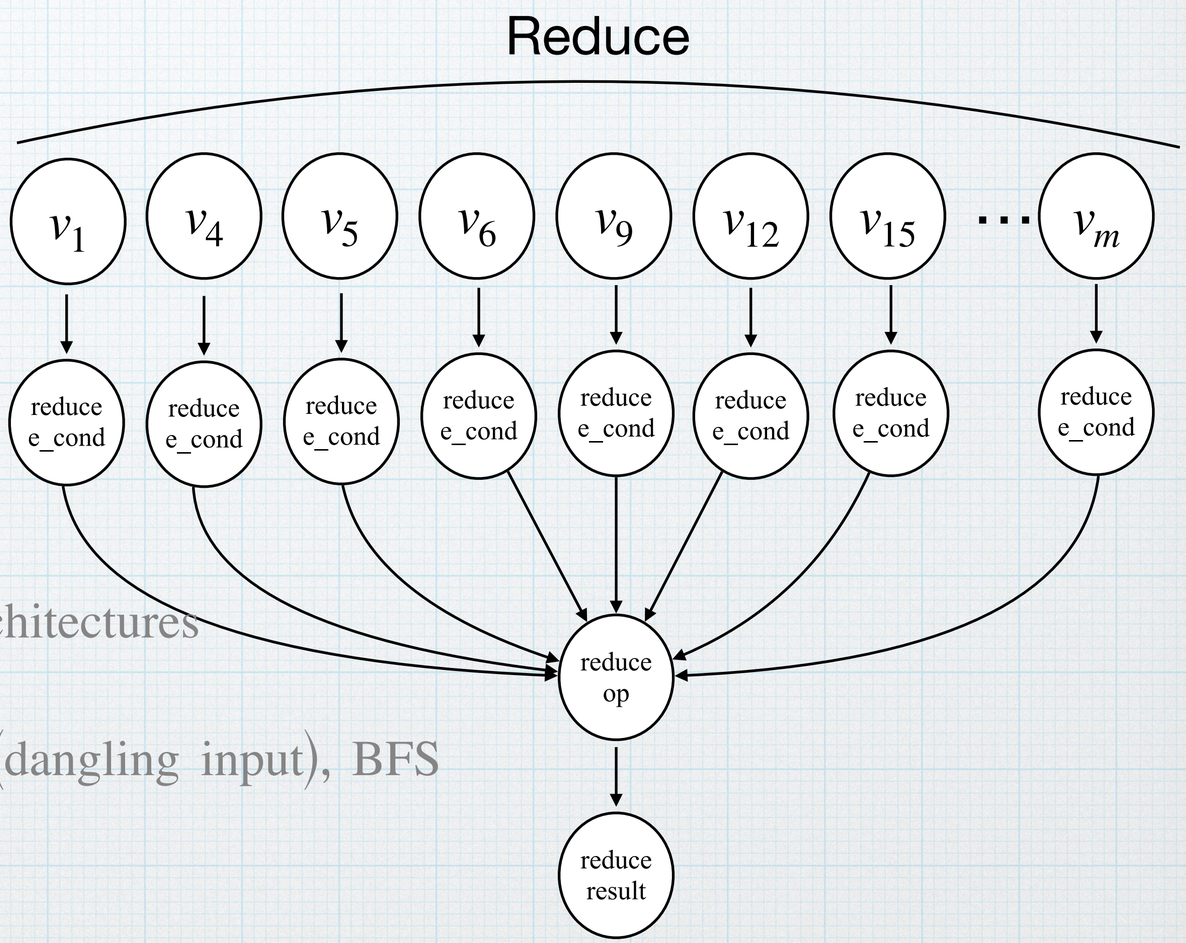
* can be relatively inefficient for sparse frontier generation, since list of vertex indexes is required to be created, which is hard to implement on vector systems



* **however, we have proposed vector-friendly copy-if algorithm, which allows to efficiently process even sparse frontiers!**

Compute and Reduce abstraction

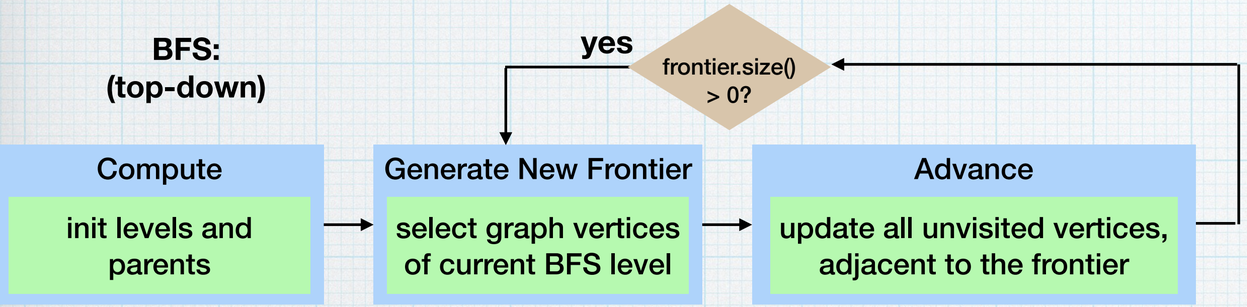




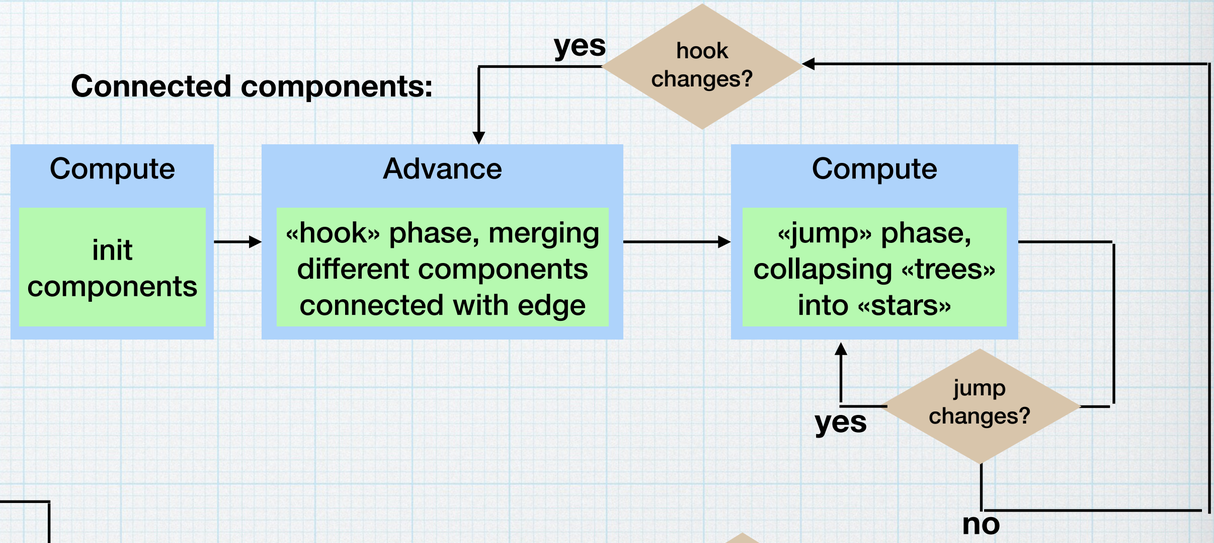
- regular parallelism => simple implementation on vector architectures
- reduce is frequently used in many graph algorithms - PR (dangling input), BFS (calculating outgoing size), etc
- reduction on vector architecture can be implemented much more efficiently compared to GPUs

Implementing graph algorithms via VGL API

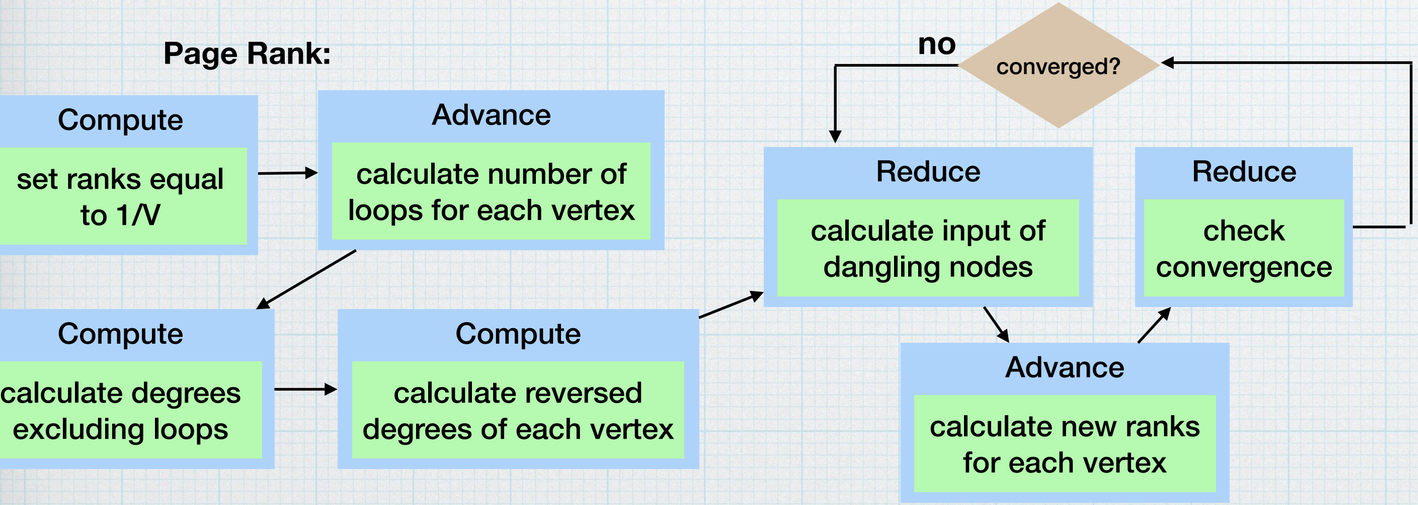
BFS:
(top-down)



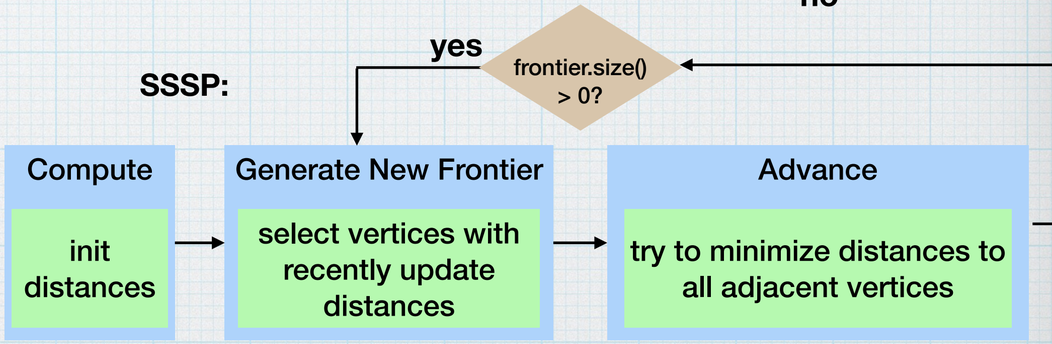
Connected components:



Page Rank:



SSSP:

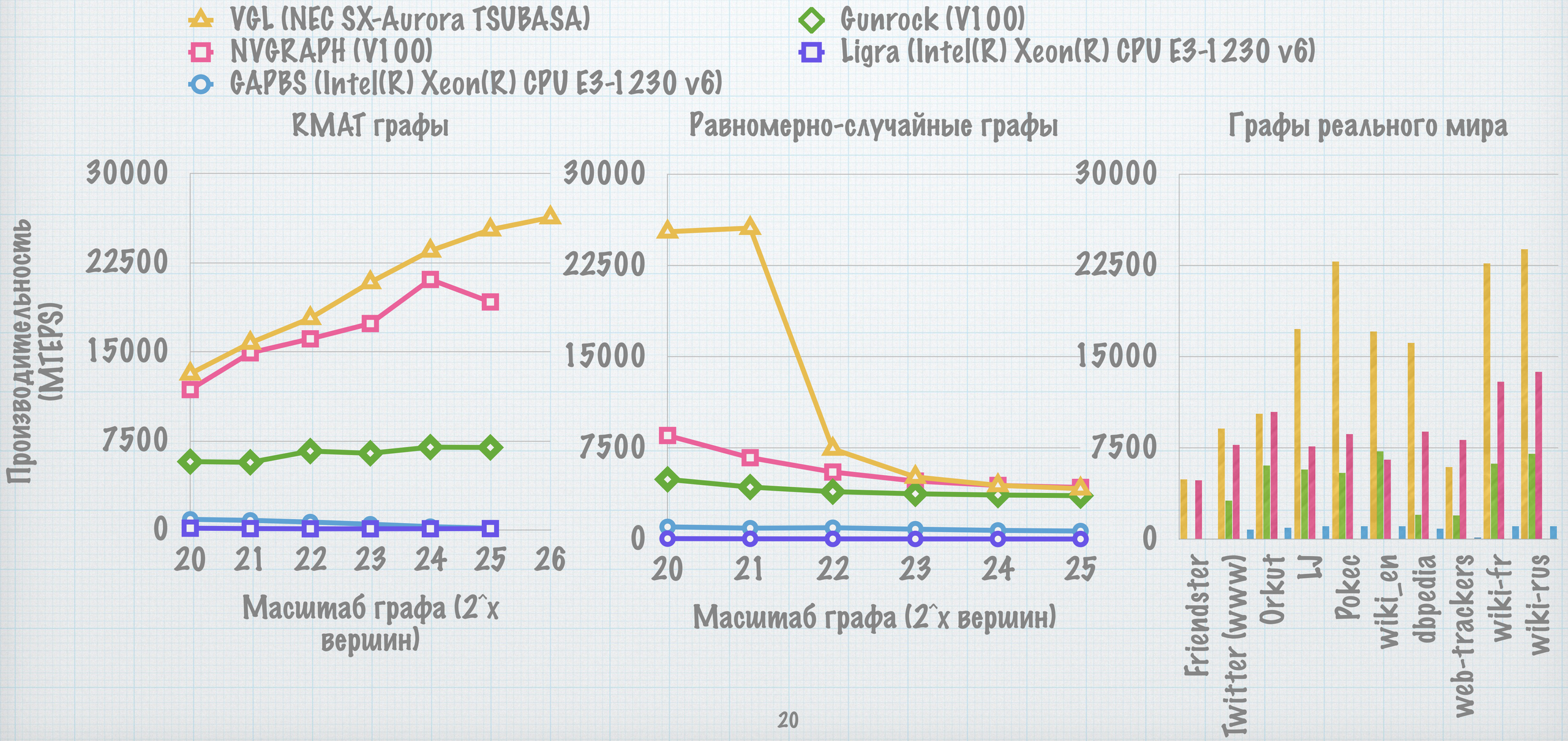


Examples of using VGL framework: top-down BFS

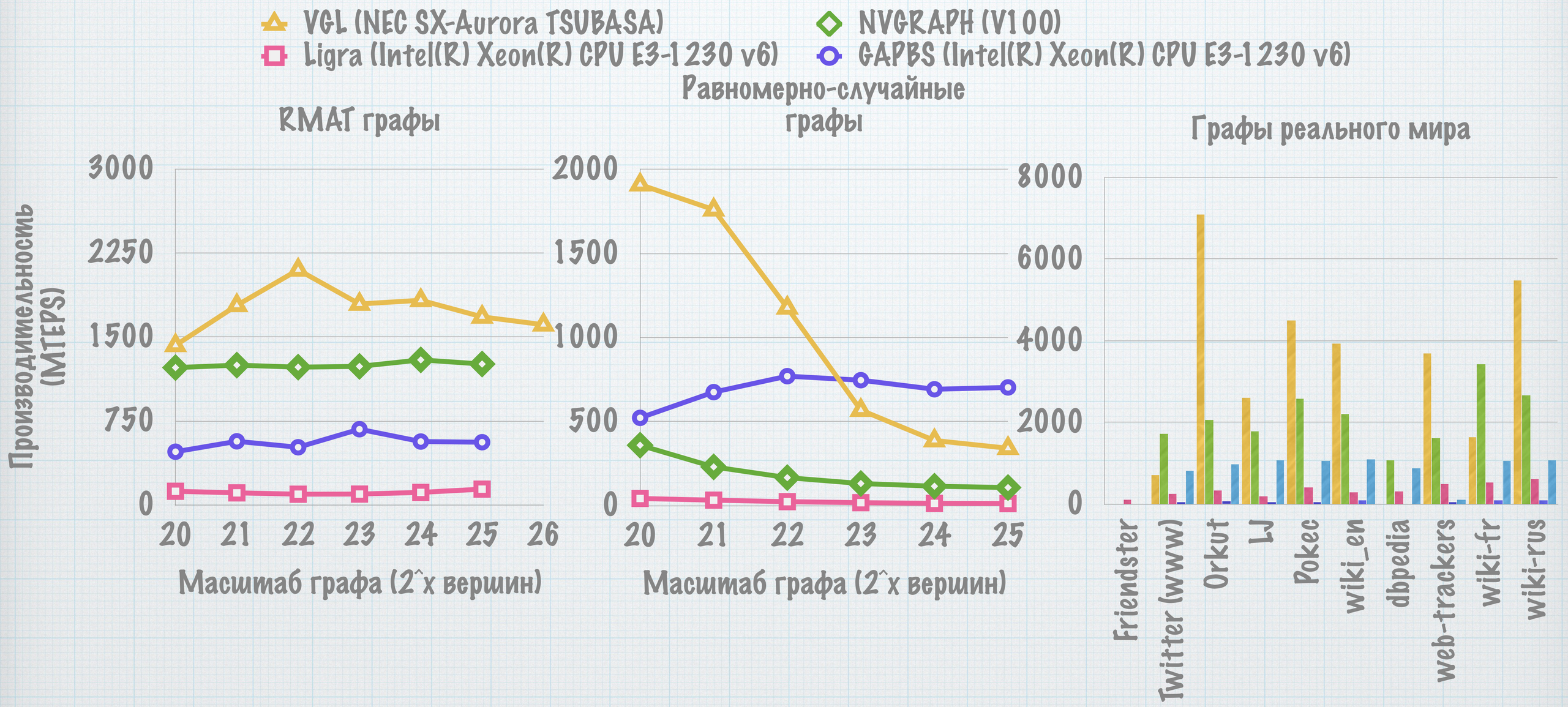
```
1.frontier.set_all_active();

3.auto init_levels = [_levels, _source_vertex] (int src_id,
  int connections_count, int vector_index)
4.{
5.     if(src_id == _source_vertex)
6.          _levels[_source_vertex] = FIRST_LEVEL_VERTEX;
7.     else
8.          _levels[src_id] = UNVISITED_VERTEX;
9.};

11.graph_API.compute(_graph, frontier, init_levels);

13.auto on_first_level = [_levels] (int src_id)->int
14.{
15.     if(_levels[src_id] == FIRST_LEVEL_VERTEX)
16.          return NEC_IN_FRONTIER_FLAG;
17.     else
18.          return NEC_NOT_IN_FRONTIER_FLAG;
19.};
20.graph_API.generate_new_frontier(_graph, frontier,
  on_first_level);
```

```
1.int current_level =  FIRST_LEVEL_VERTEX;
2.while(frontier.size() > 0)
3.{
4.     auto edge_op = [_levels, _current_level](int src_id, int
  dst_id, int local_edge_pos,
5.                    long long int global_edge_pos, int
  vector_index, DelayedWriteNEC &delayed_write)
6.     {
7.          if(_levels[dst_id] == UNVISITED_VERTEX)
8.          {
9.               _levels[dst_id] = _current_level + 1;
10.          }
11.     };
12.
13.     graph_API.advance(_graph, frontier, edge_op);
14.     auto on_next_level = [_levels, current_level] (int
  src_id)->int
15.     {
16.          if(_levels[src_id] == (current_level + 1))
17.               return NEC_IN_FRONTIER_FLAG;
18.          else
19.               return NEC_NOT_IN_FRONTIER_FLAG;
20.     };
21.     graph_API.generate_new_frontier(_graph, frontier,
  on_next_level);
22.     current_level++;
23.}
```

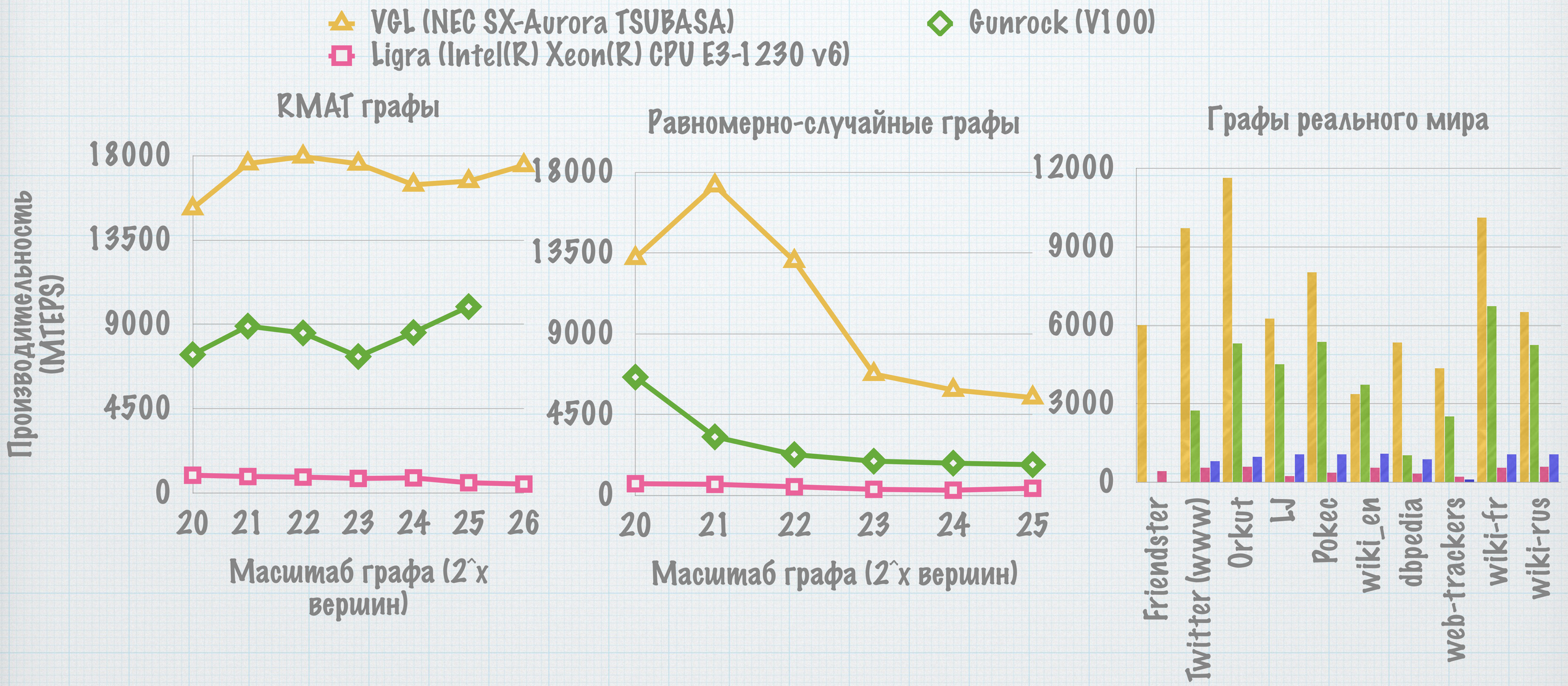
Performance Evaluation (PR)



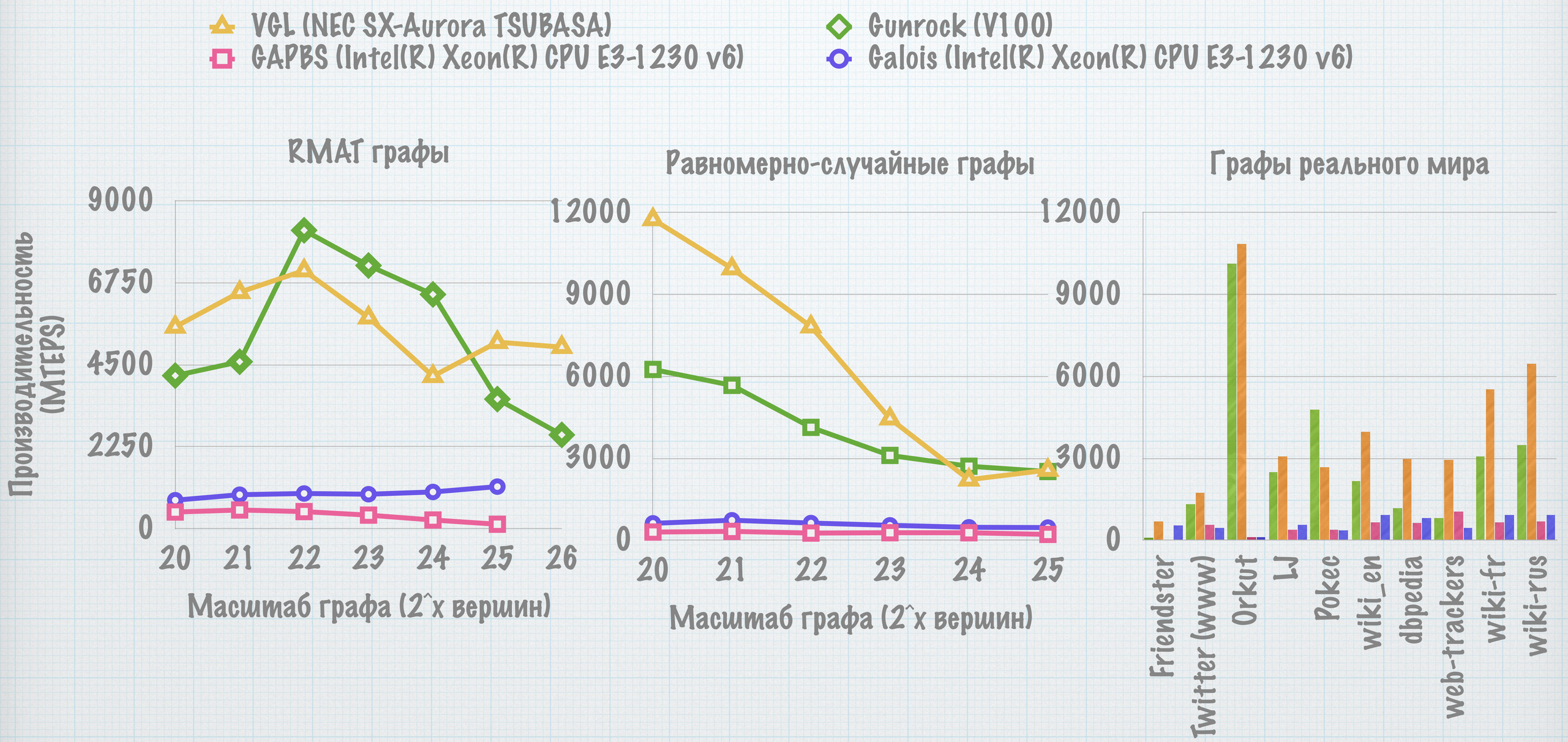
Performance Evaluation (SSSP)



Performance Evaluation (BFS)



Performance Evaluation (CC)



Conclusions and Future Work

- We implemented world-first graph processing framework (and library) for modern vector NEC SX-Aurora TSUBASA architecture
- VGL provides users a simple interface, allowing to implement many graph algorithms in under 100 lines of C++ code, which require almost no knowledge about vectorized data-processing
- Developed implementations significantly outperform existing frameworks and libraries for multicore CPUs (up to 14 times) and NVIDIA GPUs (up to 3 times)
- We plan to extend VGL framework for other vector architectures with HBM memory (for example A64FX and modern GPUs), developing world-first **architecture-independent** framework for modern vector systems