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.

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 multicore CPUs 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...
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 \([0, 1, 0, 0, 0 \ldots]\)

All-active frontier has its 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;
    }
};
```

blue vertices are «active»

operations applied sequentially

operations applied in parallel/random order
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
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 (adjacent 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!

```c
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(....);
    }
}
```

Working with sparse frontier for the last part is complicated...

```c
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(....);
    }
}
```

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
**Generate_new_frontier abstraction**

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

![Diagram of flags and vertices]

* 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

![Diagram of list of indexes]

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

regular parallelism $\Rightarrow$ 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)**
- Compute: init levels and parents
- Generate New Frontier: select graph vertices of current BFS level
- Advance: update all unvisited vertices, adjacent to the frontier

**Page Rank:**
- Compute: set ranks equal to $1/V$
- Compute: calculate degrees excluding loops
- Advance: calculate number of loops for each vertex
- Compute: calculate reversed degrees of each vertex
- Reduce: calculate input of dangling nodes
- Reduce: check convergence
- Advance: calculate new ranks for each vertex

**Connected components:**
- Compute: init components
- Advance: «hook» phase, merging different components connected with edge

**SSSP:**
- Compute: init distances
- Generate New Frontier: select vertices with recently update distances
- Advance: try to minimize distances to all adjacent vertices

**Compute**
- init levels and parents
- Generate New Frontier: select graph vertices of current BFS level
- Advance: update all unvisited vertices, adjacent to the frontier
- «hook» phase, merging different components connected with edge
- «jump» phase, collapsing «trees» into «stars»

**Reduce**
- check convergence

**Compute**
- reduce input of dangling nodes
- check convergence

**Reduce**
- check convergence

**Connected components:**
- yes
  - hook changes?
- no
  - yes
  - jump changes?
  - no
Examples of using VGL framework: top-down BFS

1. `frontier.set_all_active();`
2. `int current_level = FIRST_LEVEL_VERTEX;`
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. `};`
10. `while(frontier.size() > 0)`
11. `{`
12. `auto edge_op = [levels, _current_level](int src_id, int dst_id, int local_edge_pos, long long int global_edge_pos, int vector_index, DelayedWriteNEC &delayed_write)`
13. `{`
14. `if(_levels[dst_id] == UNVISITED_VERTEX)`
15. `{`
16. `_levels[dst_id] = _current_level + 1;`
17. `}`
18. `else`
19. `return NEC_NOT_IN_FRONTIER_FLAG;`
20. `};`
21. `graph_API.generate_new_frontier(_graph, frontier, on_first_level);`
22. `current_level++;`
Performance Evaluation (PR)

RMAT графы

Масштаб графа (2^x вершин)

Производительность (MTEPS)

0
7500
15000
22500
30000

VGL (NEC SX-Aurora TSUBASA)
NVGRAPH (V100)
GAPBS (Intel(R) Xeon(R) CPU E3-1230 v6)
Gunrock (V100)
Ligra (Intel(R) Xeon(R) CPU E3-1230 v6)

Равномерно-случайные графы

Графы реального мира

Приложение
Performance Evaluation (SSSP)

RMAT графы

Производительность (MTEPS)
0
750
1500
2250
3000
Масштаб графа (2^x вершин)
20 21 22 23 24 25 26

Равномерно-случайные графы

Производительность (MTEPS)
0
500
1000
1500
2000
Масштаб графа (2^x вершин)
20 21 22 23 24 25

Графы реального мира

Friendster
Twitter (www)
Orkut
LJ
Pokec
wiki_en
dbpedia
web-trackers
wiki-fr
wiki-rus
Performance Evaluation (BFS)

RMAT graphs

Graphs of the real world

- Friendster
- Twitter (www)
- Orkut
- LJ
- Pokec
- wiki_en
- dbpedia
- web-trackers
- wiki-fr
- wiki-rus
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