Developing Efficient Implementations of Shortest Paths and Page Rank algorithms for NEC SX-Aurora TSUBASA Architecture

Ilya Afanasyev, Vladimir Voevodin, Vadim Voevodin [2]

Hiroaki Kobayashi, Kazuhiko Komatsu [1]

[1] Tohoku University (Japan)

[2] Lomonosov Moscow State University (Russia)

RuSCD 2019

22.09.2019

Moscow, Russia

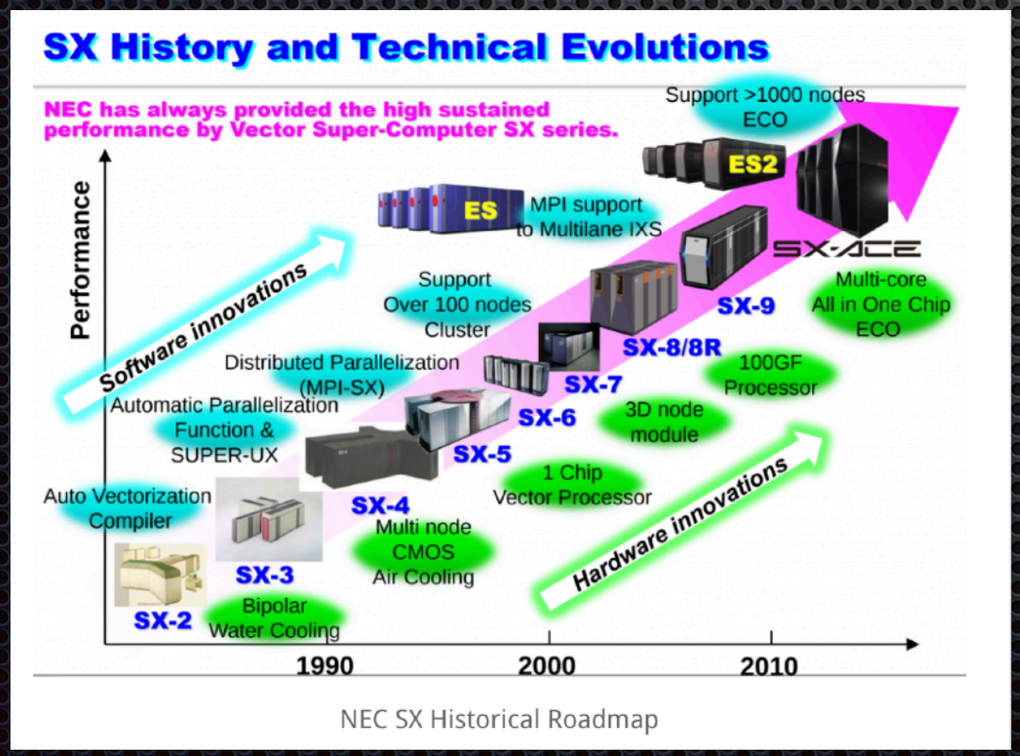
Motivation

- Large-scale graph processing problems are extremely relevant to study nowadays, since graph-analytics has numerous real-world applications
- In modern supercomputing a variety of different platforms, architectures, and configurations are used on hardware side
- It is important to study which platforms are capable of more efficient graph-processing, and which programming techniques it is necessary to use in order to maximise execution efficiency
- Current work is devoted to the investigation of vector processing possibilities for large-scale graph problems on **NEC SX-Aurora TSUBASA** architecture
- Implementing graph algorithms for vector architectures is typically challenging, because of irregularity in graph data-structure and memory access patterns
- No implementations of graph algorithms for SX-Aurora TSUBASA exist yet (!)

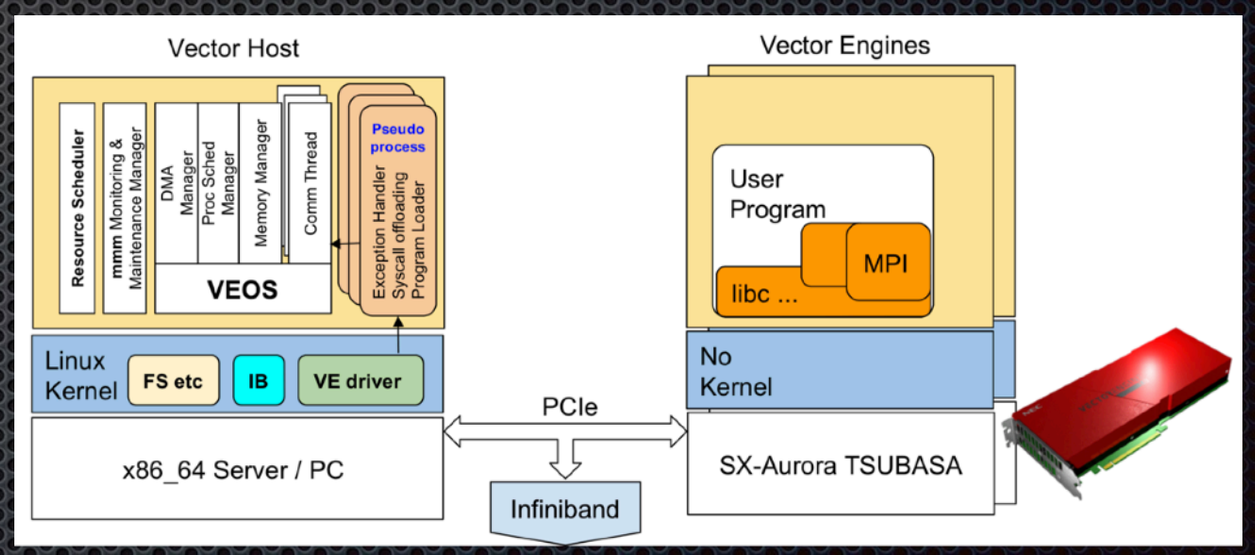
NEC SX-Aurora TSUBASA Architecture Details

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



NEC SX Historical Roadmap



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

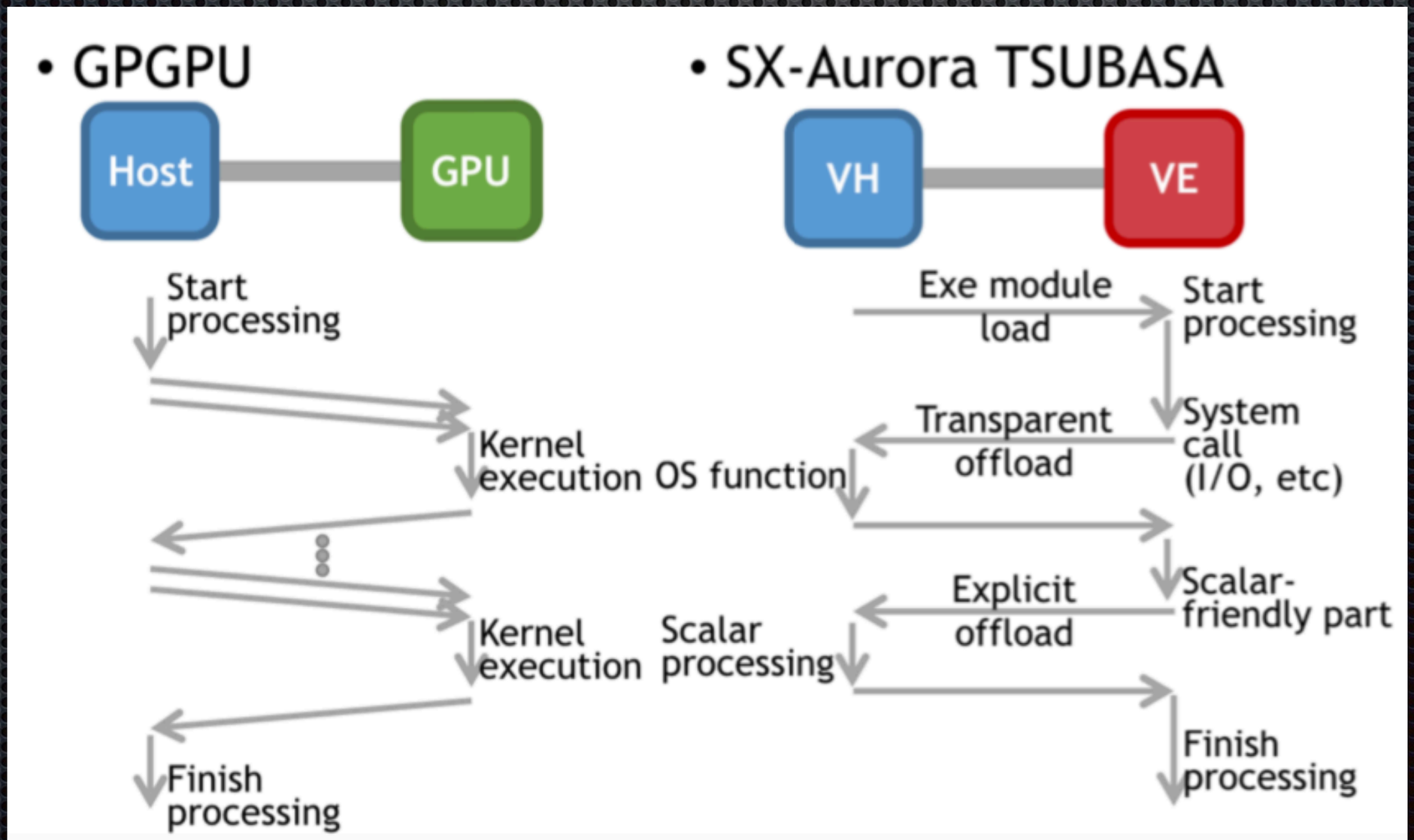
NEC SX-Aurora TSUBASA & GPUs

- NEC SX-Aurora TSUBASA vector engines have many similar properties/characteristics with modern GPUs:
  - A combination of MIMD and SIMD execution model
  - High-bandwidth memory utilisation, optimised for collective memory accesses performed by warps/vector instructions
  - Installed as co-processor (with different execution model)

- Many people are familiar with GPU programming, and many graph-processing frameworks are already implemented for GPUs (Gunrock, NVGRAPH, cuSHA, etc.)
- **Question:** Does these similarities mean that graph algorithms can be implemented on SX-Aurora TSUBASA in the same way?

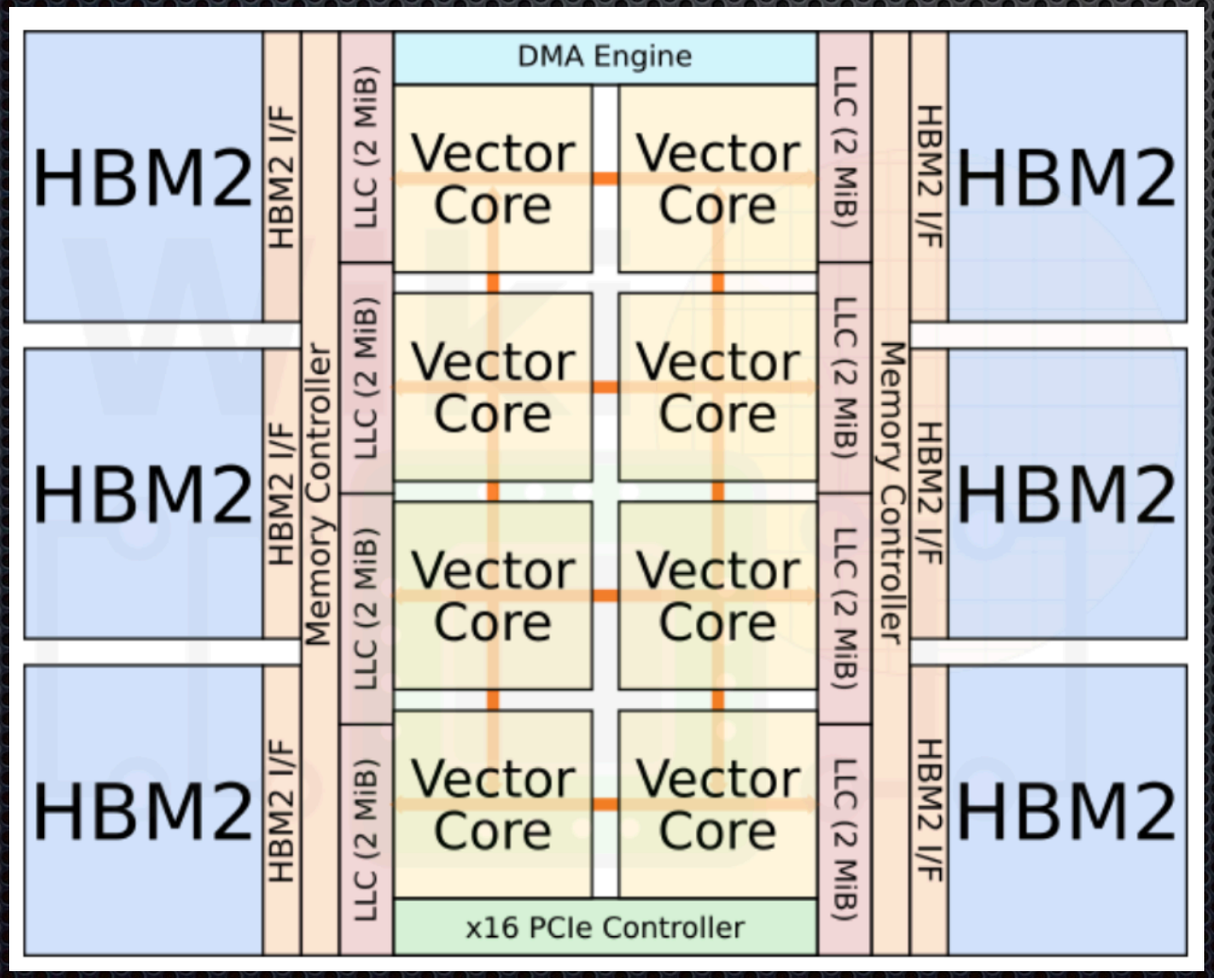
Both Systems Are Installed As Co-processors

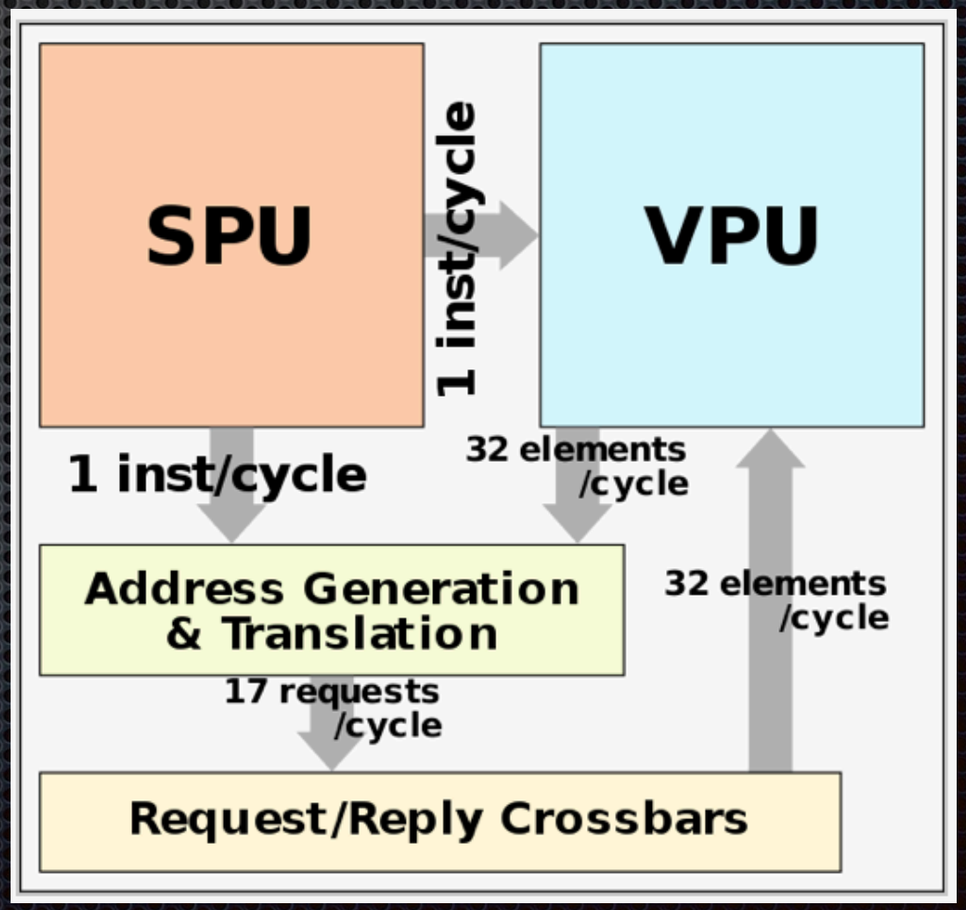
- Despite both architectures are installed as co-processors, their execution models are different:



NEC SX-Aurora TSUBASA Vector Cores

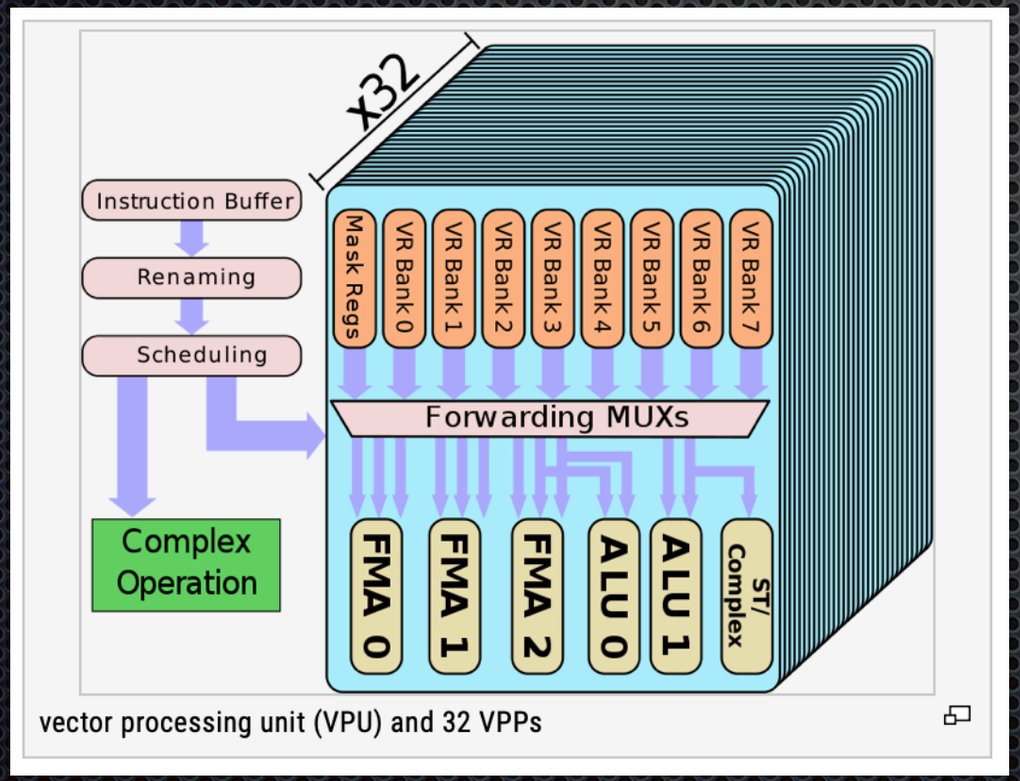
- 8 powerful vector cores
- Each core has scalar (SPU) and vector (VPU) processing units



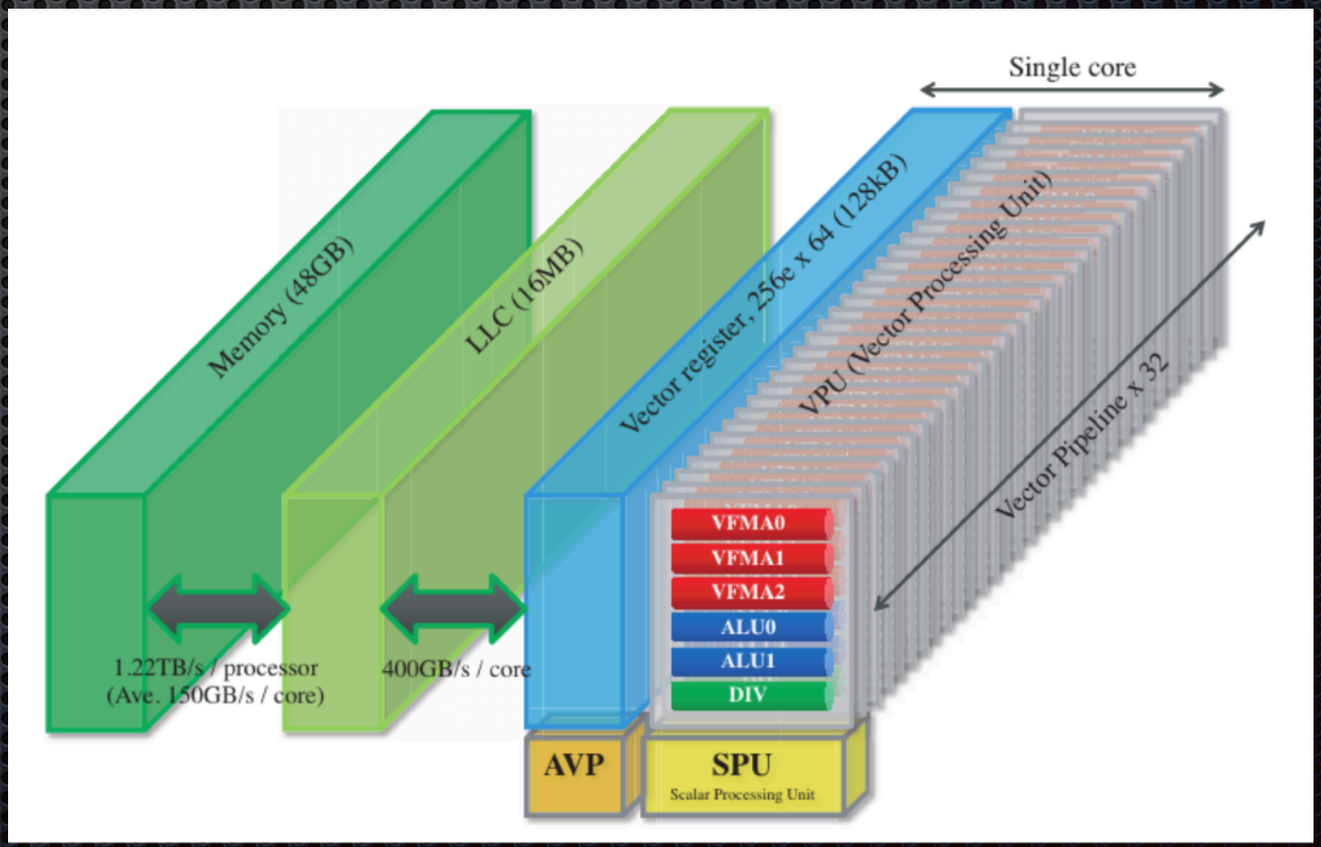


NEC SX-Aurora TSUBASA Execution Model

- Different vector cores work according to MIMD model
- Each vector core operates with vector instructions of length 1-256 (SIMD)
- Each vector command of length 256 is processed by 32 VPP in portions, pipelining is organised between different computational units (FMA, ALU,….)



vector processing unit (VPU) and 32 VPPs



Memory Bandwidth & Cache Hierarchy

- Both NEC SX-Aurora TSUBASA and modern GPU architectures utilise High Bandwidth Memory 2 (HBM2) technology, and thus have similar memory hardware characteristics
- However, SX-Aurora and GPUs have different cache-hierarchy: SX-Aurora Vector cores direct transactions through large LLC shared cache (16 MB), while CUDA-cores use relatively small L1(64 KB) and L2 (4 MB) caches

Architecture	Memory type	Memory capacity	Theoretical peak bandwidth (GB/s)	Bandwidth achieved on STREAM benchmark (GB/s)	The ratio of bandwidth achieved on STREAM to theoretical
SX-Aurora TSUBASA	HBM2	up to 48 GB	1200	995	82 %
NVIDIA Pascal P100	HBM2	up to 16 GB	732	628	85 %
NVIDIA Volta V100	HBM2	up to 32 GB	900	809	89 %

Which graph algorithms we are going to investigate?

Graph Problems & Input Data

- Currently we implemented 4 important graph-processing algorithms for different real-world problems:
  - **Single Source Shortest Paths (SSSP)** — *Navigation, Peer to Peer Networks*
  - **Page Rank (PR)** — *ranking web-pages, finding leaders in communities*
  - **Connected Components (CC)** — *finding communities in social networks*
  - **Single Source Widest Paths (SSWP)** - *optimising traffic in networks*

- We have also experimented with **various data sets** (input graphs), including:
  - **Synthetic graphs** (RMAT, Uniform-random)
  - **Road map graphs** (USA, New-York, CA, etc)
  - **Social network graphs** (Twitter, LiveJournal, Pockec, Wikipedia)
  - **Web-graphs** (various domain subgraphs)

Implementation Details

- In current work we investigate only **shared-memory architecture** implementations

- Shared-memory implementations of graph algorithms are generally **much more efficient**, since graph-processing problems tend to have an enormous amount of **inter-node communications**, what bottlenecks node performance

- Moreover, modern studies demonstrate that platforms with shared-memory architecture are currently able to process a lot of **real-world graphs** (different social networks, etc)

Implementation Details

Challenges of Implementing Graph Algorithms on NEC SX-Aurora TSUBASA Architecture

1. Optimising effective memory bandwidth during graph traversals (while loading both vertices and edges data)
    - loading information about **vertices** implies **indirect memory access** pattern
    - loading information about **edges** implies **direct memory access** pattern
    - both should be executed with high efficiency!
2. Traversing graphs using **vector instructions** of maximum (256) length
3. Efficiently balancing parallel workload between 8 **vector cores**

Challenge 1: Optimising Effective Memory Bandwidth

- Graphs algorithms are typically memory-bound
- Thus, it is important to maximise **effective memory bandwidth** during graph traversals
- In order to achieve high memory bandwidth on SX-Aurora TSUBASA:
    1. all required data should be loaded either from LLC cache or sequentially from memory
    2. vector cores should load data with a specific pattern
- Thus, a **specific graph storage format** should be used
- Ideally, this format should also be helpful with 2 other problems (utilising vector instructions and inter-core load balancing)

Graph Algorithms Classification (Active-Based)

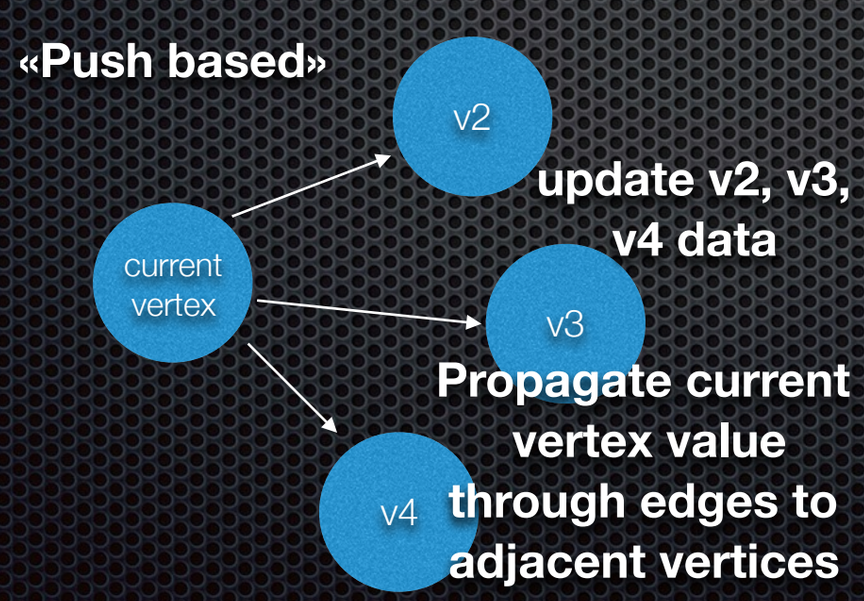
- Iterative graph algorithms can be classified into 2 groups:
  1. **All-active** (all vertices of input graphs are traversed on each iteration)
  2. **Partially-active** (only specific «active» vertices of input graph are traversed on each iteration)

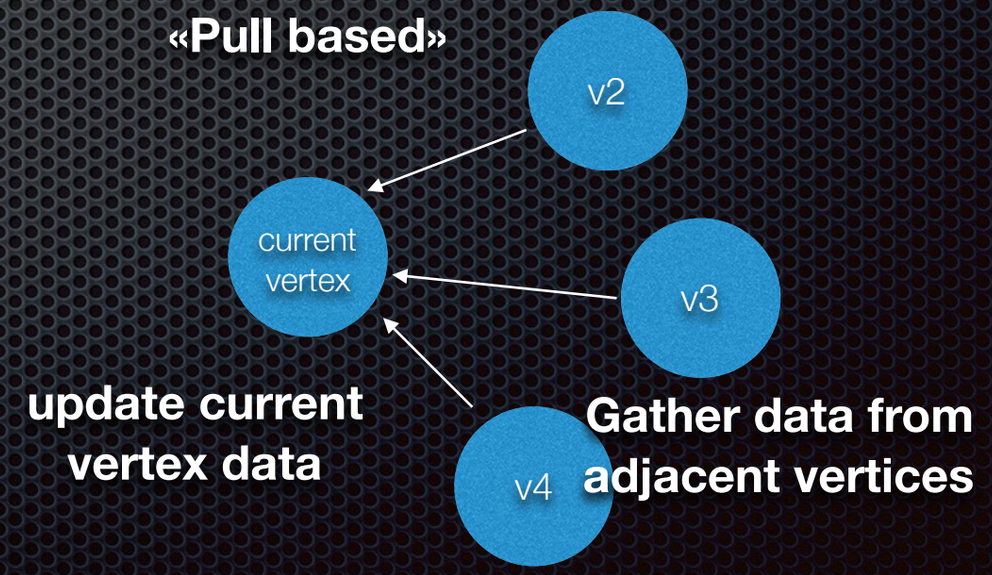
- Typically «partially-active» graph algorithms have lower computational complexity, but are significantly harder to implement (especially on vector systems)
- All listed problems (SSSP, PR, CC, SSWP) can be solved with all-active algorithms rather efficiently

Graph Algorithms Classification (Direction-Based)

- Another possible classification of graph algorithms is based on direction of traversals:
  1. **pull-based** (each vertex updates it's own value)
  2. **push-based** (each vertex updates values of it's neighbours)
- We use pull-based traversals, since they don't require atomic operations support and typically allow to obtain higher effective bandwidth

- On SX-Aurora Gather operation is faster compared to Scatter operation
- Pull-based traversals require transposed (reversed graphs)





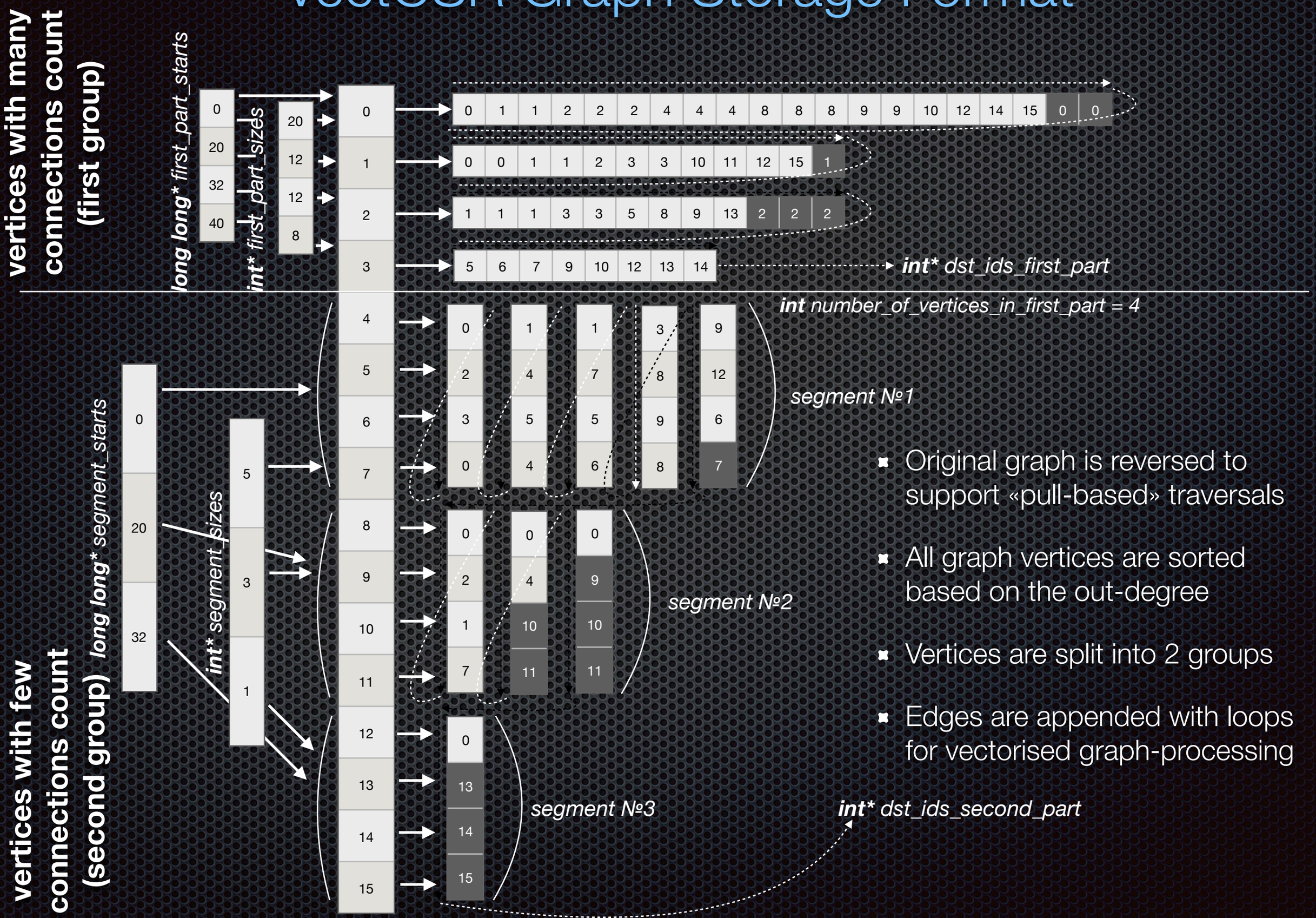
Developing Graph Storage Format...

- «All-active» graph algorithms allow to perform efficient vectorised graph traversals and support efficient memory access pattern

- We use an approach, when each vector instruction processes 256 different graph vertices in parallel

- We propose VectCSR graph storage format, which is optimised for supporting vectorised graph traversals and efficient memory accesses

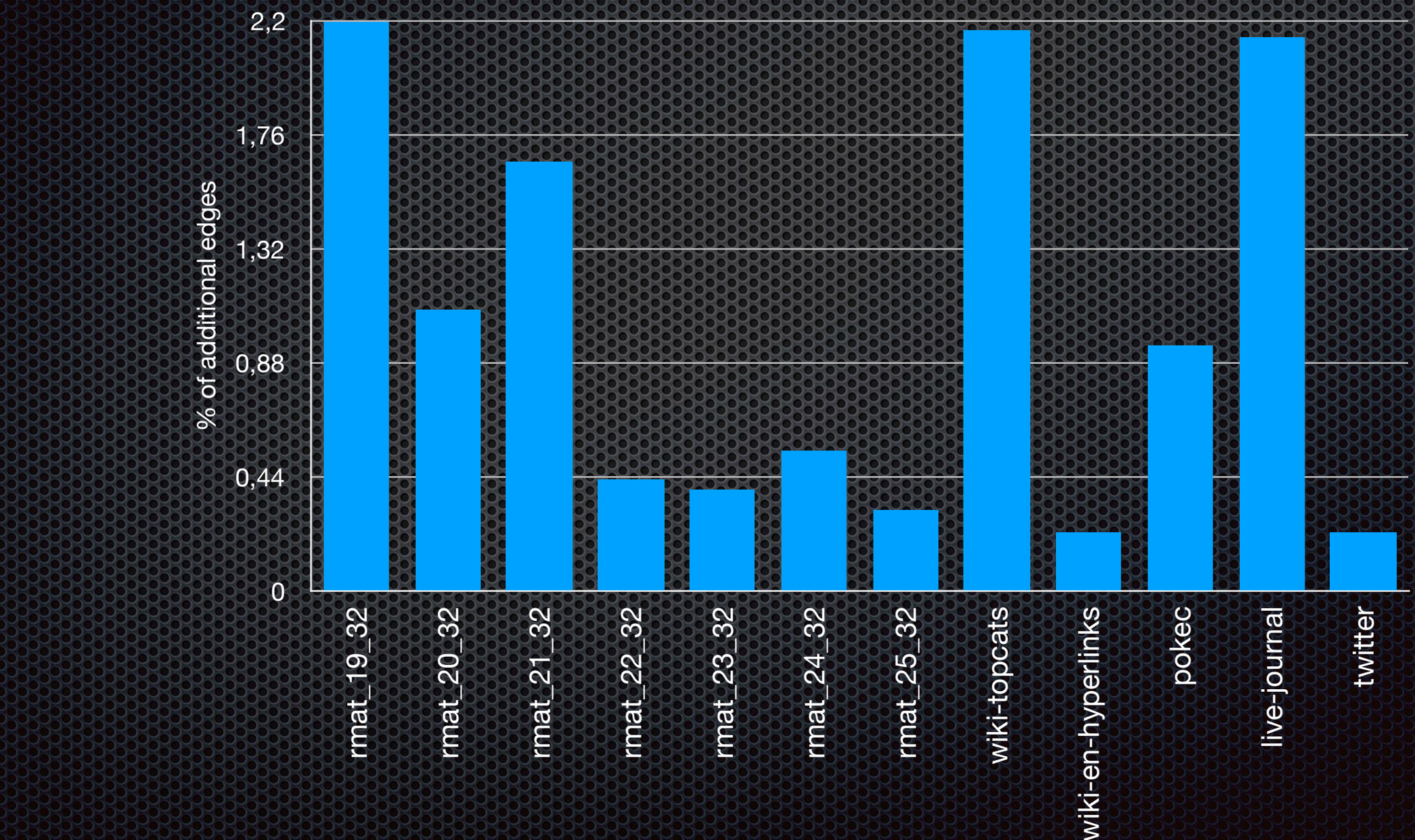
VectCSR Graph Storage Format



- Original graph is reversed to support «pull-based» traversals
- All graph vertices are sorted based on the out-degree
- Vertices are split into 2 groups
- Edges are appended with loops for vectorised graph-processing

How much extra space VectCSR format requires?

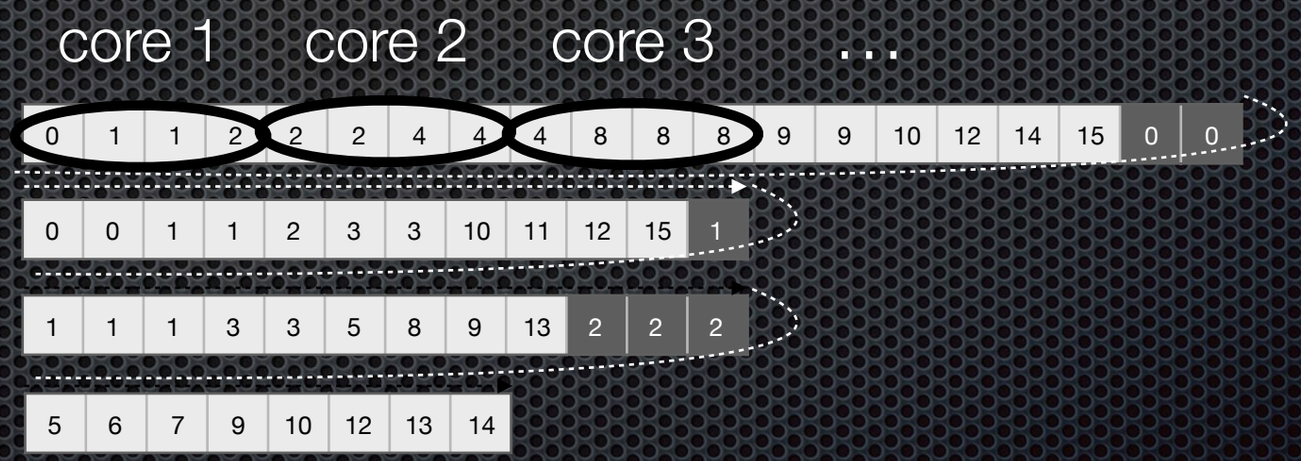
- Not many additional edges are added to the graph (due to vertex sorting):



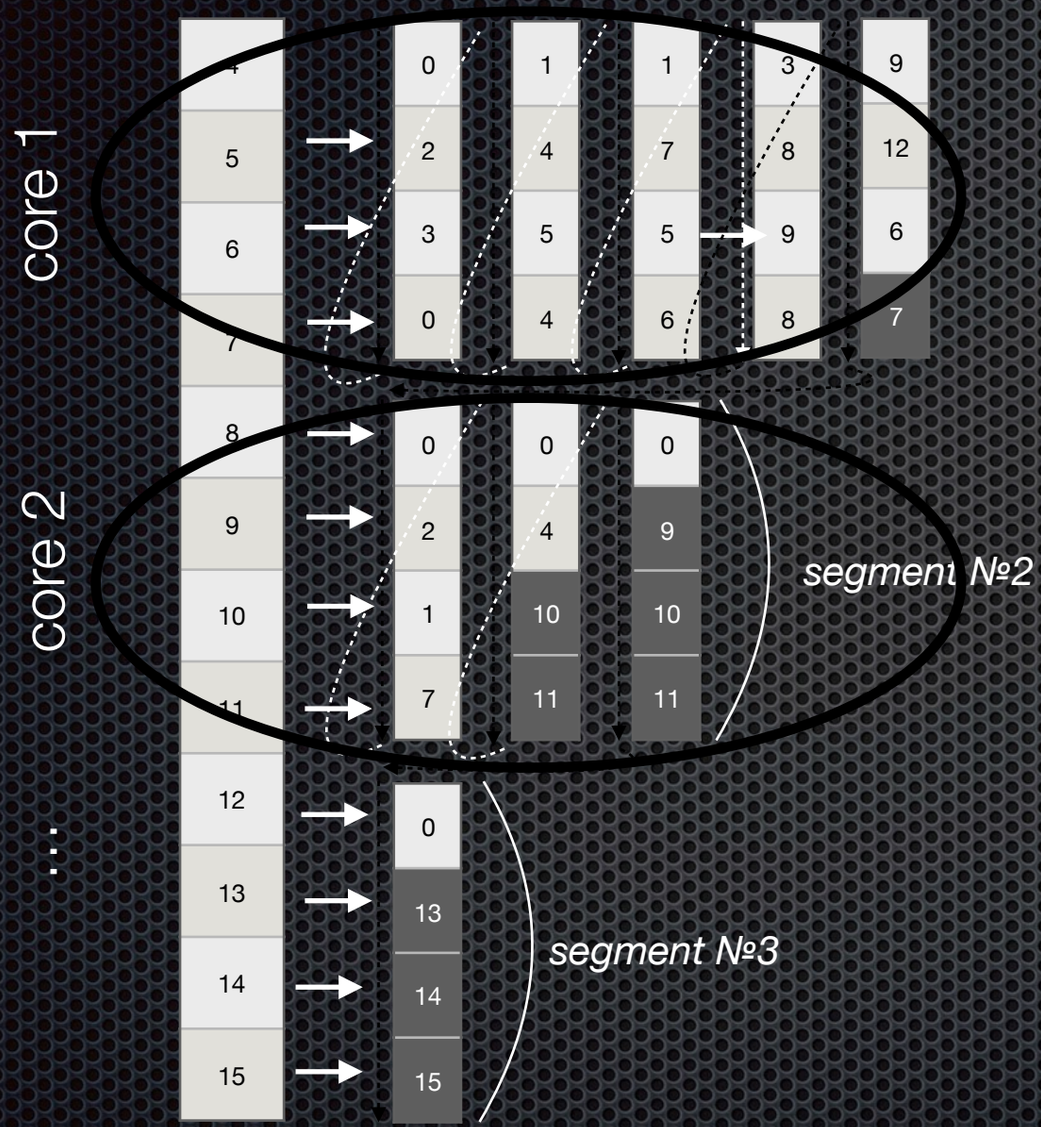
Graph Traversals in VectCSR Format (First Group)

- Each vertex of the first group is processed collectively by all 8 vector cores with vector instructions of 256 length

- #pragma omp for schedule(static, 1) is used

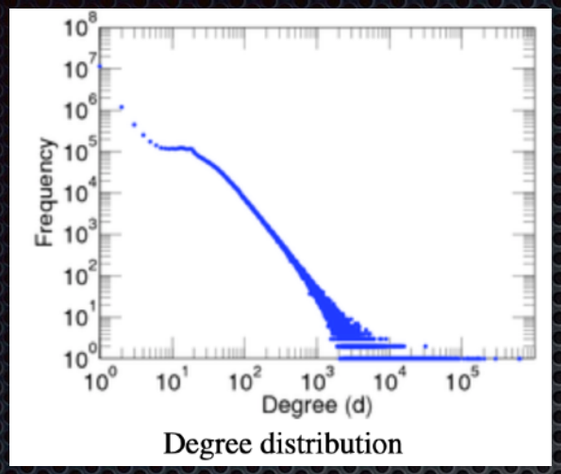


Graph Traversals in VectCSR Format (First Group)



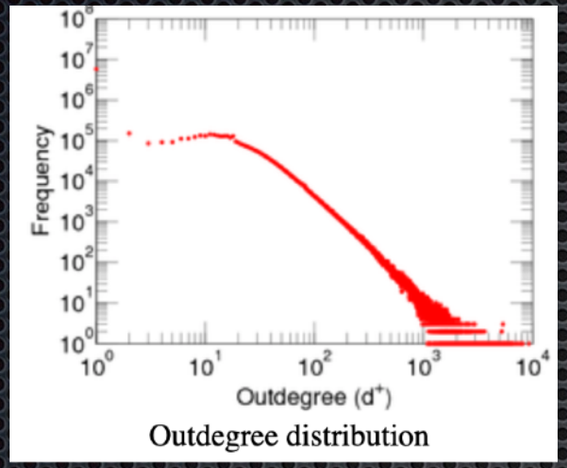
- each 256 sequential vertices from the second group are processed by a single vector core
- #pragma omp for schedule(static, 1) is used to process different groups of vertices

Memory Access Patter in Social-Network Graphs

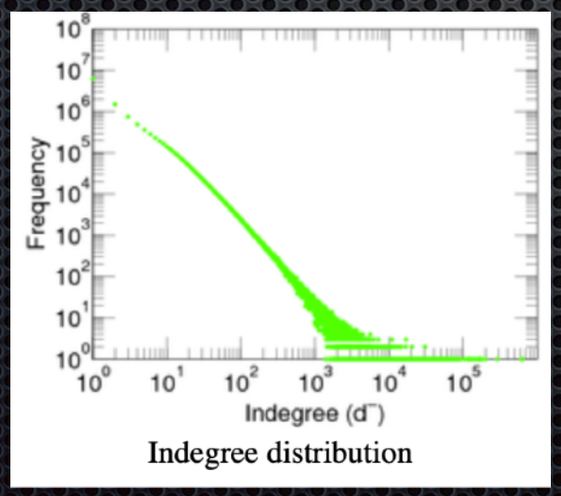


Degree distribution

- Social-network graphs have power-law properties, which:
  1. Allow to efficiently cache **most frequently accessed** graph vertices **(+)**
  2. Create a lot of cache-conflicts when accessing these vertices **(-)**



Outdegree distribution



Indegree distribution

- We implement **2 optimisations**, aimed to improve accesses to these vertices:
  1. Storing information about most frequently accessed vertices in private copies of arrays, located in different areas of LLC cache **(eliminate inter-core conflicts)**
  2. Storing information about most frequently accessed vertices with intervals of 3 elements
- These 2 optimisations allow to achieve similar performance when processing **power-law** graphs compared to **random-uniform** graphs

2 Memory Access Pattern Optimisations

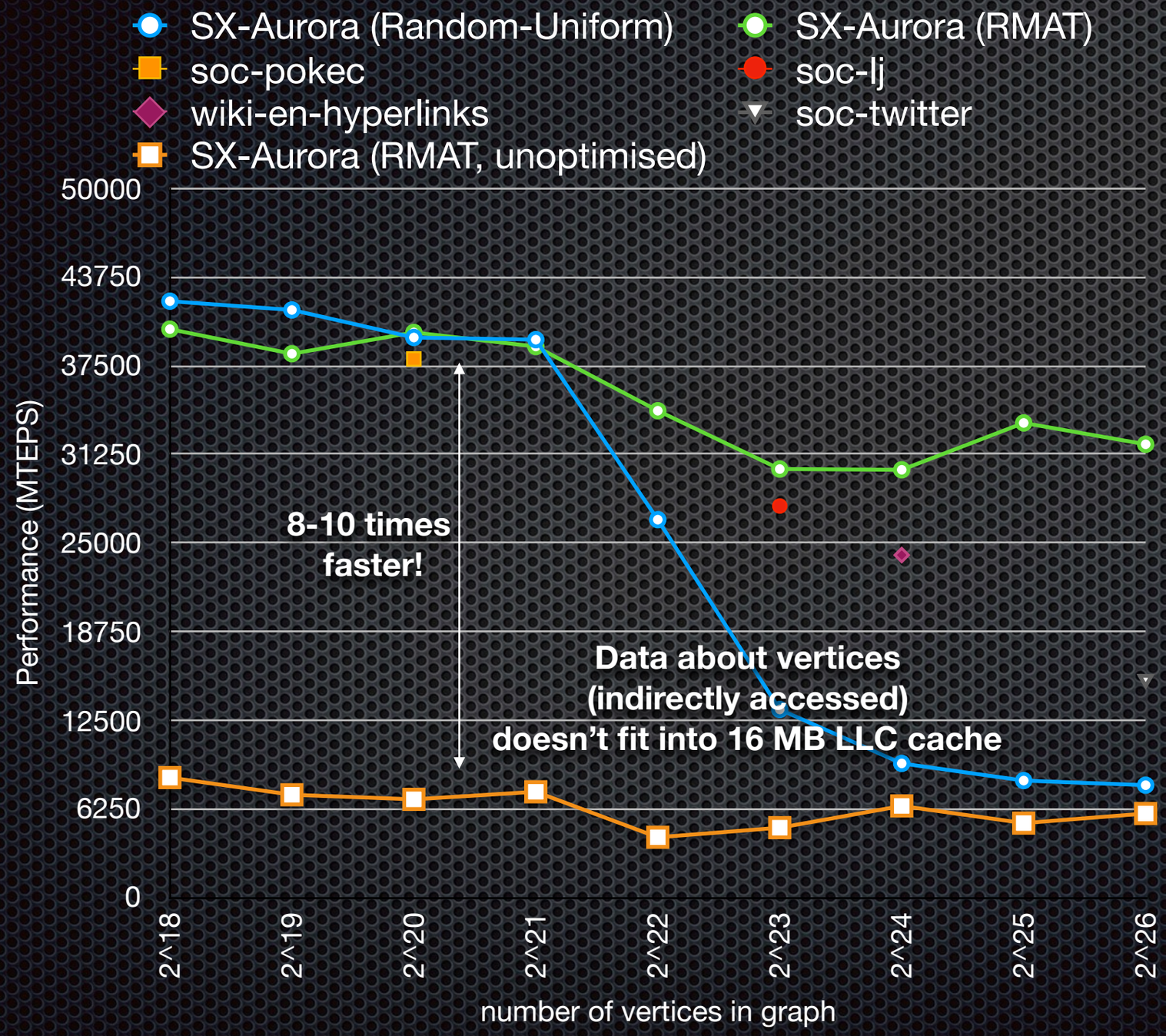
```
#ifdef __USE_NEC_SX_AURORA__
#pragma _NEC retain(private_src_array)
#endif

for(int i = 0; i < CACHED_VERTICES; i++)
    private_src_array[i * CACHE_STEP] = _src_array[i]; // copy data about most frequently accessed vertices to private
arrays

#ifdef __USE_NEC_SX_AURORA__
#pragma _NEC ivdep
#pragma _NEC vovertake
#pragma _NEC novob
#pragma _NEC vector
#endif
#pragma omp for
for(long long int i = 0; i < edges_count; i++)
{
    _T dst_value = 0;
    int dst_id = outgoing_ids[i];
    if(dst_id < CACHED_VERTICES)
    {
        dst_value = private_src_array[dst_id * CACHE_STEP]; // store data about most frequently accessed vertices with
intervals
    }
    else
    {
        dst_value = _src_array[dst_id];
    }
    _dst_array[i] = dst_value;
}
```

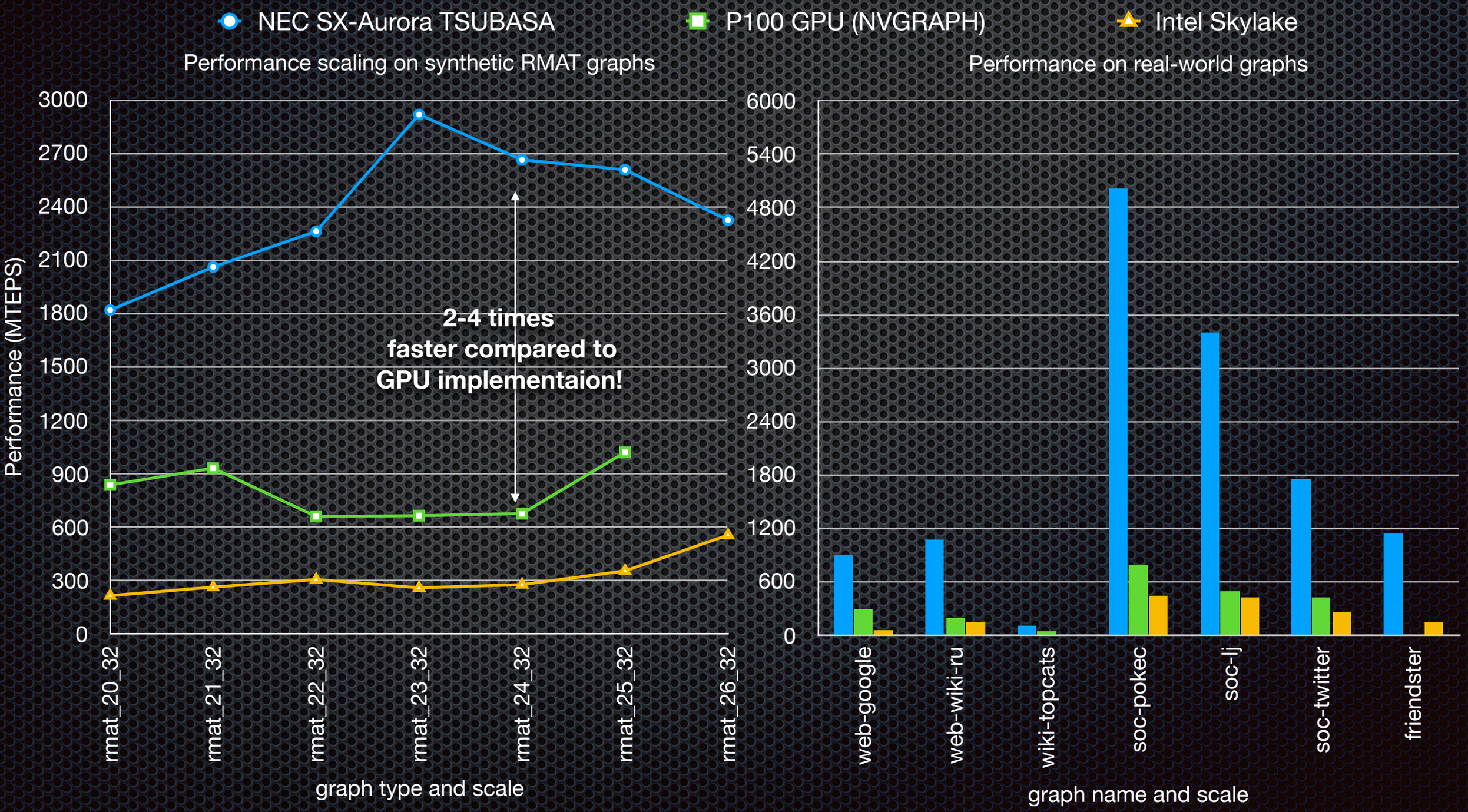
Performance Evaluation

Performance of Graph Traversal in VectCSR Format

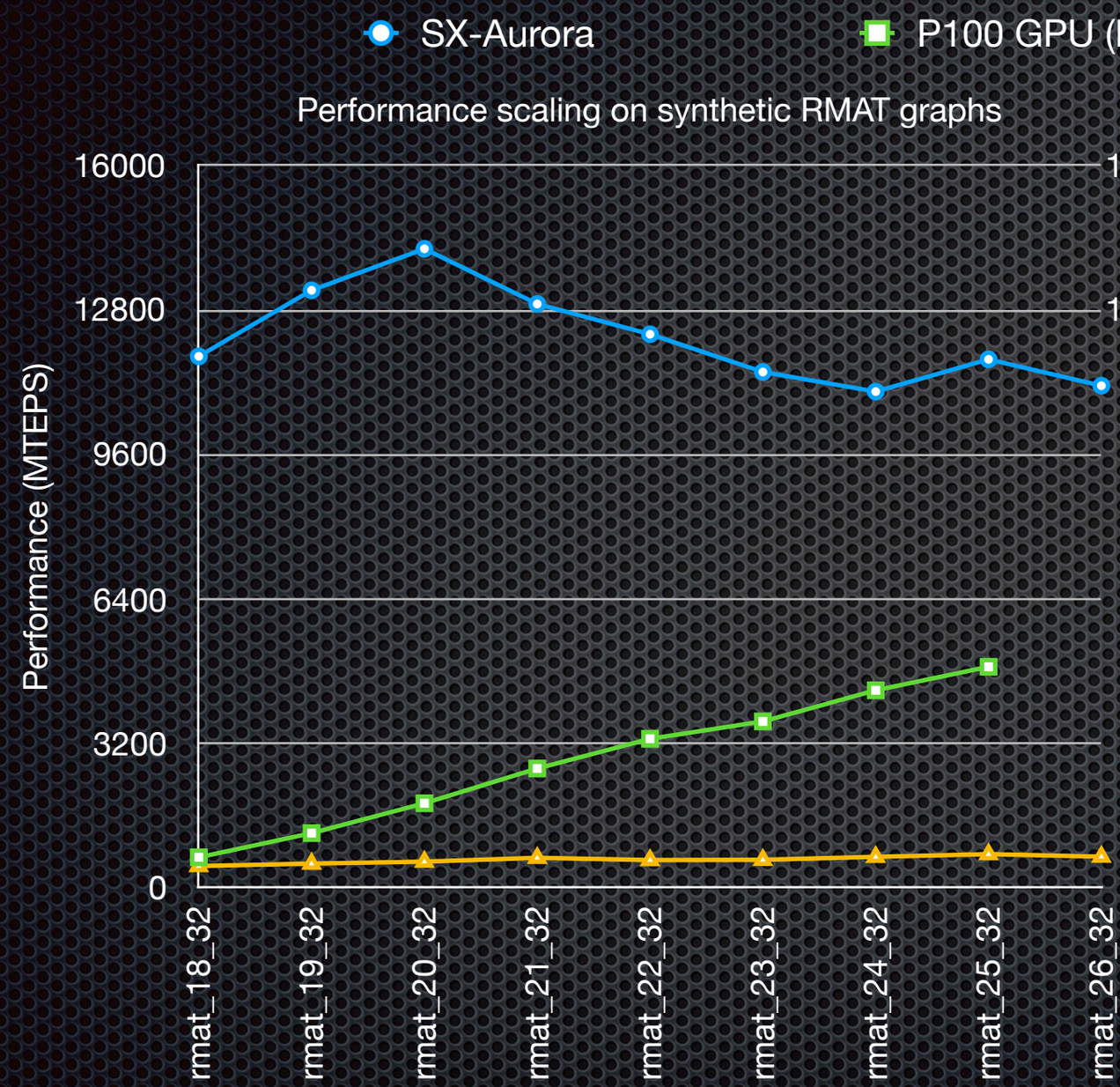


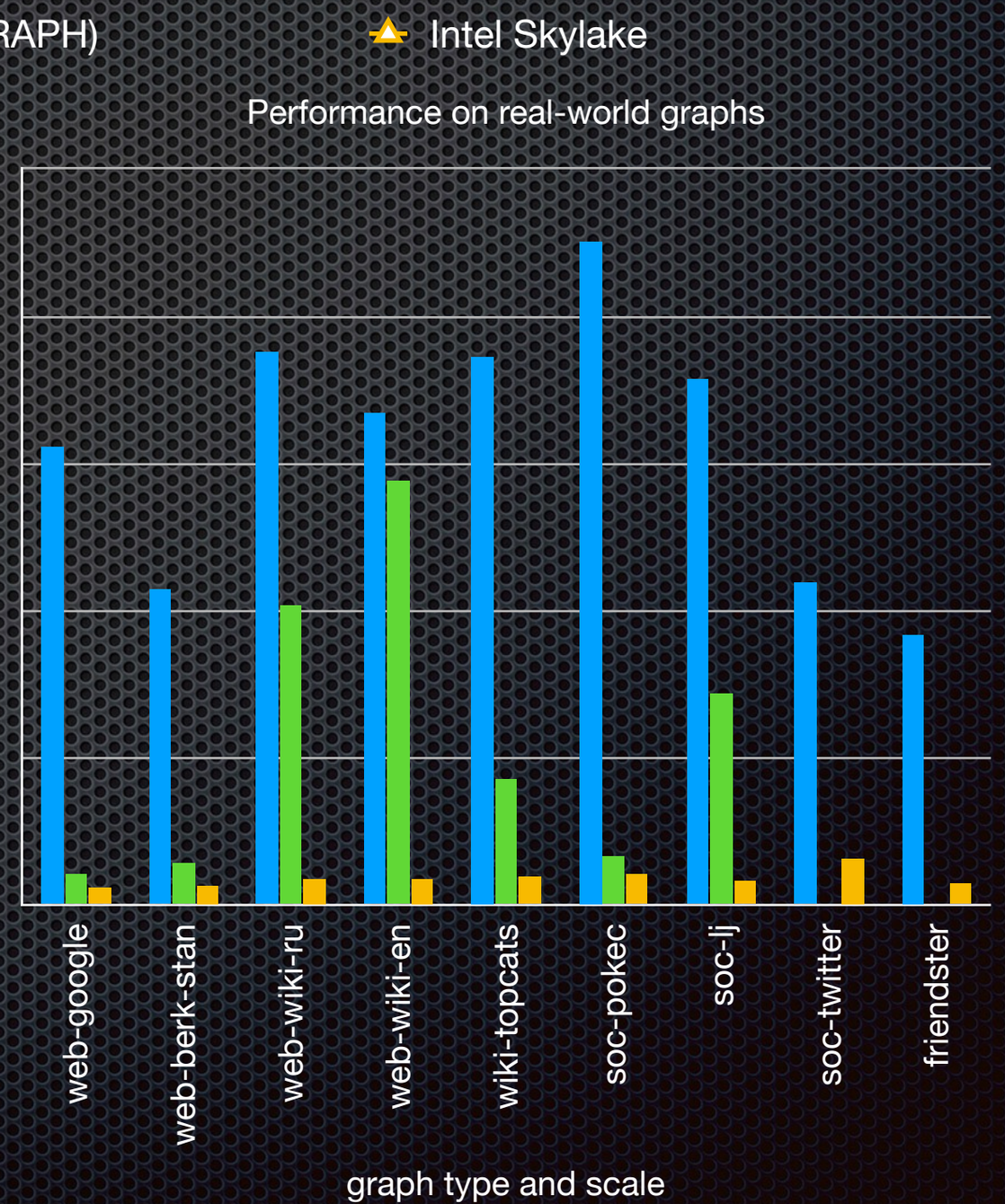
- Performance on RMAT scales well since most indirect memory accesses are directed to the first vertices with large connections count
- Performance for Uniform-Random Graphs drops significantly when vertices data can't be stored in LLC cache
- Real-World graphs have «middle» performance

Single Source Shortest Paths (SSSP) Performance



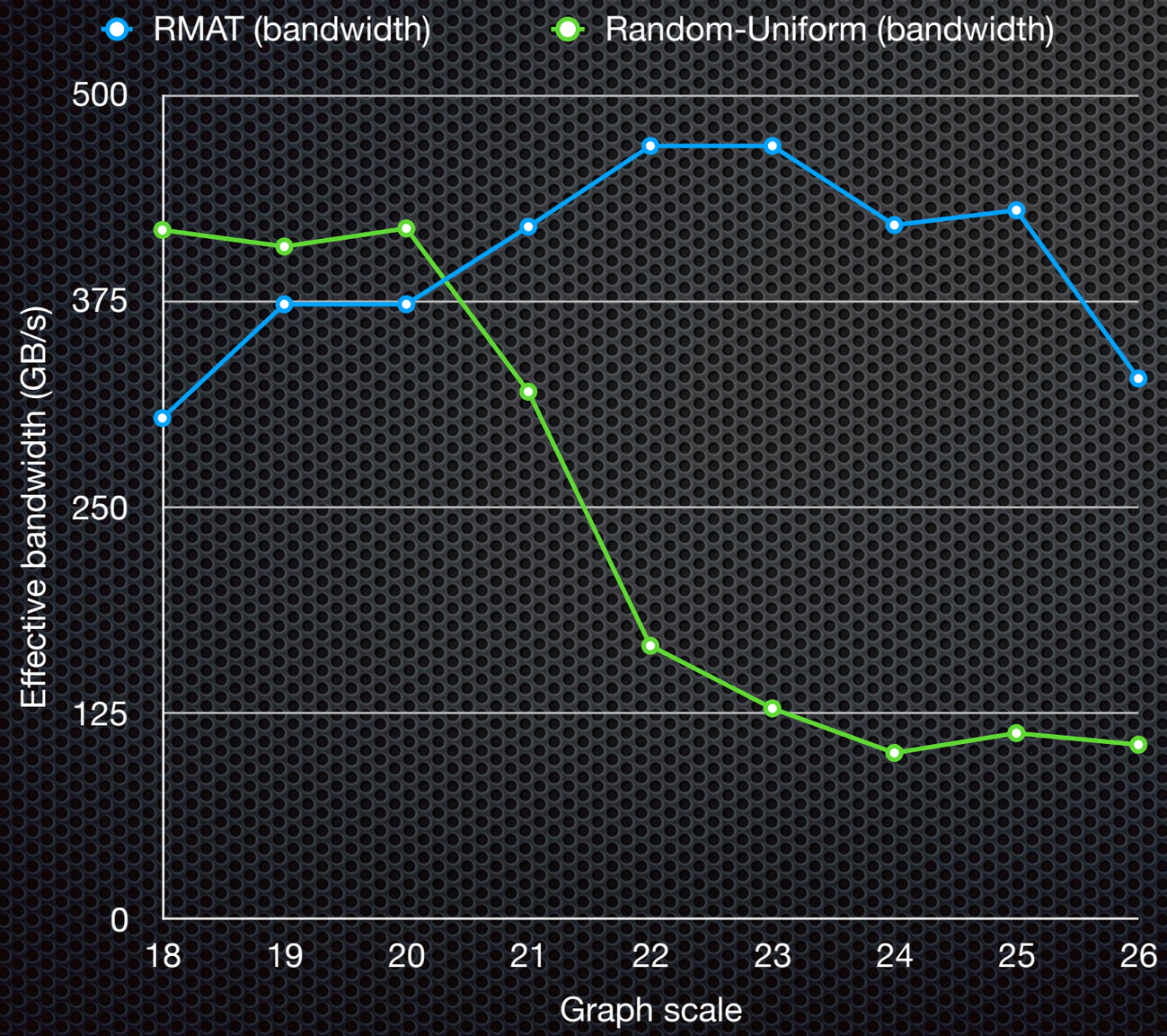
Page Rank (PR) Performance





What about bandwidth?

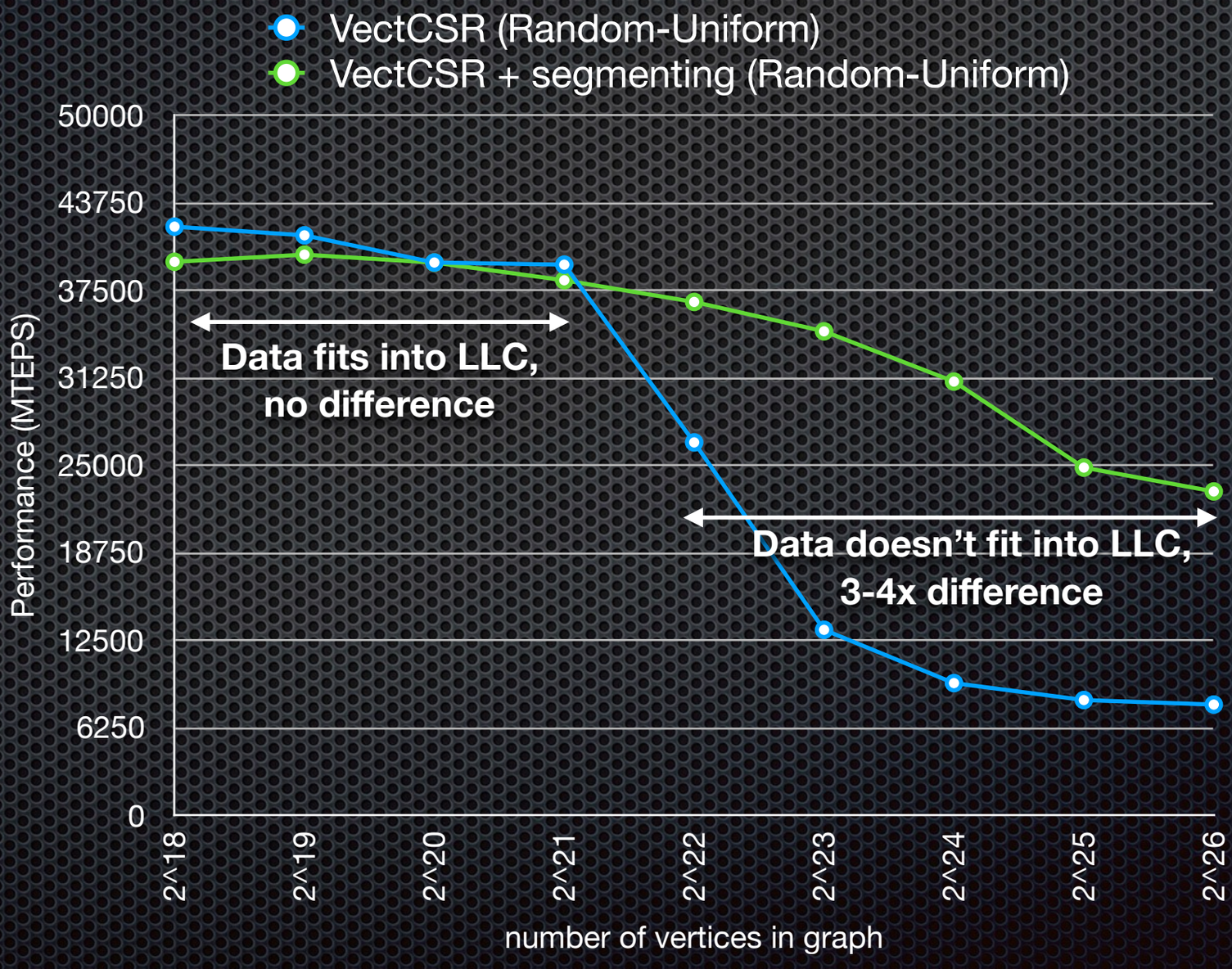
- For RMAT graphs we obtain about 40-50% of theoretical peak bandwidth
- Bandwidth for Random-Uniform graphs drops significantly when indirectly accessed vertices data can not be placed inside LLC cache



- Higher effective memory bandwidth (80-90%) can not be obtained in programs with gather/scatter instructions when working with 4-byte data, since:
  - gather/scatter instructions load 256*4=1024 bytes of data
  - load/store instructions are capable of loading 2048 bytes of data, which effectively doubles achieved bandwidth

Is it Possible to Improve Performance for Large Random-Uniform Graphs (and other real-world graphs)?

- In order to improve locality of indirect memory accesses, **cache-blocking technique** can be used (similar to CPU, unlike to GPU)



Conclusions

- We presented **world-first** attempt to implement **vectorised graph algorithms** for NEC SX-Aurora TSUBASA architecture
- We discussed techniques, which can be used to implement «all active» «pull-direction» graph algorithms, including **VectCSR** format
- 4 fundamental graph processing algorithms **(SSSP, CC, PR, SSWP)** have been implemented, significantly outperforming **similar GPU-based** implementations
- Developed implementations allow to achieve **40-50%** of t**heoretical peak memory bandwidth** on several power-law graphs
- Possibility of using **cache-blocking** technique was investigated
- Future works includes developing more algorithms for SX-Aurora architecture, and investigating approaches for developing **different groups of graph algorithms**