The comparison of large-scale graph processing algorithms implementation methods for Intel KNL and NVIDIA GPU

Ilya Afanasyev[1] and Vladimir Voevodin[1]

[1] Lomonosov Moscow State University, Moscow, Russia

Abstract. The paper describes implementation approaches to large-graph processing on two modern high-performance computational platforms: NVIDIA GPU and Intel KNL. The described approach is based on a deep a priori analysis of algorithm properties that helps to choose implementation method correctly. To demonstrate the proposed approach, shortest paths and strongly connected components computation problems have been solved for sparse graphs. The results include detailed description of the whole algorithm's development cycle: from algorithm information structure research and selection of efficient implementation methods, suitable for the particular platforms, to specific optimizations for each of the architectures. Based on the joint analysis of algorithm properties and architecture features, a performance tuning, including graph storage format optimizations, efficient usage of the memory hierarchy and vectorization is performed. The developed implementations demonstrate high performance and good scalability of the proposed solutions. In addition, a lot of attention was paid to profiling implemented algorithms with NVIDIA Visual Profiler and Intel® VTune ™ Amplifier utilities. This allows current paper to present a fair comparison, demonstrating advantages and disadvantages of each platform for large-scale graph processing.

Keywords: Graph algorithms, GPU, KNL, CUDA, vectorization, SSSP, SCC, large-scale graph processing.

1   Introduction

The interest to large-scale graph processing is growing rapidly, since graphs successfully emulate real-world objects and connections between them. In many areas, people need to identify some patterns and rules from object relationships that results into processing large amounts of data. The examples of such objects and relationships are: analysis of social, semantic and Internet networks, infrastructural problems solution (analysis of transport and energy networks), biology (analysis of the network of protein-protein interactions), health-care (epidemic spreading analysis), social-economic modelling. All those problems have one common property: their model graphs have a very large size, so a parallel approach is required to perform computations in reasonable amount of time.

2

The question, which parallel computational platforms are able to process graphs more efficiently, is still open. Graphic accelerators and coprocessors perform really well for solving traditional problems, such as linear algebra computations, image processing or solving molecular dynamics problems, since they provide high performance and energy efficient computational power together with high throughput memory. The most well-known and widely used families of coprocessors are NVidia GPU and Intel Xeon Phi. Recent important trend is that vendors try to combine central processors and coprocessor functions, which results into modern KNL Xeon PHI architecture.

2   Target architectures

2.1   NVidia GPU

Modern NVidia GPUs belong to three architectures: Kepler, Maxwell and Pascal. Currently, Kepler is the most common and widely used architecture in HPC. Tesla K40 accelerator, which has been used for all testing in current paper, has 2880 cores with clock signal rate of 745 MHz. This GPU provides peak performance up to 4.29 TFLOPs on single precision computations and 12 GB device memory with 288 GB/s bandwidth. The PCI-express 3.0 bus with 32 Gbps bandwidth is used to maintain connection between host and device. Memory hierarchy also includes L1 (64 KB), and L2 (1.5 MB) caches. Device computational model is very important: thread is a single computational unit; 32 threads are grouped into a warp, which works using SIMD approach. The warp performance is also very affected by memory access data pattern (coalesced memory access) and conditional operations presence.

During the tests the corresponding host was equipped with Intel(R) Xeon(R) CPU E5-2697 v3 @ 2.60GHz processor. For compilation NVCC v6.5.12 from CUDA Toolkit 6.5 has been used with –O3 –m64 –gencode arch=compute_35,code=sm_35 options.

2.2   Intel KNL

The newest architecture of Intel Xeon Phi coprocessors is Knights Landing (KNL). Each processor has 64-72 cores (in current paper a 68-core accelerator is used) with a clock signal rate of 1.3-1.5 GHz. Processor provides a peak performance up to 6 TFLOPs on single precision computations. It also has two memory levels: high-bandwidth MCDRAM memory with a capacity up to 16 GB and bandwidth up to 400 GB/s, and DDR4 memory with a capacity up to 384 GB and bandwidth up to 90 GB/s. Cores are grouped in Tile-s (pair of cores), each having a common 1 MB size L2 cache. Another important feature of Intel KNL is the support of vector instructions AVX-512, containing gather and scatter operations, which are necessary for graph processing. For compilation ICC 17.0.0 has been used with –O3 –m64 options.

3 State of art

Algorithms for solving shortest paths problem for CPU and GPU are described following papers: [1][2][3]This approach can be applied for KNL architecture too, which is demonstrated in the current paper. Sequential (Tarjan) algorithm for solving strongly connected components problem is presented in [4]. Another algorithm (Forward-Backward), which has a much larger parallelism potential, but also have a larger computational complexity, is presented in the papers [5] and [6]. CUDA implementation of this algorithm is also researched in papers [6] and [8].

4 Research methodology

The current paper uses the following structure to describe both graph problems. First, an accurate mathematical problem definition is formulated, to prevent any ambiguity. After that, a review of most important possible algorithms is presented together with target architecture features. Based on the results of this survey, well-suited algorithms for all architectures are selected.

After that, first implementation of all selected algorithm is developed, followed by a series of iterative optimizations and profiling. It is extremely important to analyze the final implementation perfomance, and how well the implementations use target hardware features. As a result, conclusions about advantages and disadvantages of both architectures for solving a specific graph problem can be presented.

5 Shortest paths problem

5.1 Mathematical description

A directed graph $G = (V, E)$ with vertices $V = (v_1, v_2, \ldots, v_n)$ and edges $E = (e_1, e_2, \ldots, e_m)$ is given. Each edge $e \in E$ has a weight value $w(e)$. The path is defined as edges sequence $\pi_{u,v} = (e_1, \ldots, e_k)$, beginning in vertex u and ending in vertex v, so that each edge follows another one. A path length can be defined as $w(\pi_{u,v}) = \sum_{i=1}^{k} w(e_i)$. A path $\pi_{u,v}^*$ with minimal possible length between vertices u and v is called a shortest path: $d(u, v) = w(\pi_{u,v}^*) = \min w(\pi_{u,v})$.

Depending on a vertices pair choice, between which a search is performed, the shortest paths problem can be formulated in three different ways:

— SSSP (single source shortest paths) — shortest paths from a single selected source vertex are computed.
— APSP (all pairs shortest paths) — shortest paths between all pairs of graph vertices are computed.
— SPSP (some pairs shortest paths) — shortest paths between some pre-selected pairs of vertices are computed.

4

In the current paper the SSSP problem will be researched, since it is the simplest and most basic between these problems: for example, APSP problem for large-scale graphs can be solved by repeated calls of SSSP operation for each source vertex, since traditional algorithms, such as Floyd-Warshal, can not be applied because high memory requirements.

5.2   Algorithm descriptions

SSSP problem can be solved with two traditional algorithms: Dijkstra and Bellman-Ford.

— **Dijkstra's algorithm** is designed to solve the problem in graphs with edges, having only non-negative weights. A variation of the algorithm, implemented with a Fibonacci heap has the most efficient time complexity $O(|E| + |V|\log|V|)$. The algorithm's computational core includes sequential traversals of vertices beginning from the source vertex; during each traversal algorithm while puts adjacent vertices to the stack (or heap), so they can be processed later. As a result, the global vertices traversal in the algorithm can be performed only sequentially, while local adjacent vertices traversal can be executed in parallel as described in [10], but it's usually provides not enough parallelism for significant GPU utilization.
— **Bellman-Ford algorithm** is designed to solve the problem in graphs, including those which have edges with negative weights. The computational core of the algorithm consists of a few iterations, each of which requires traverses of all graph edges. Computations continue until there are no changes in the distance array. The algorithm has a sequential complexity equal to $O(p|E|)$, where p is the maximum possible length of the shortest path from the source vertex to any other. As a consequence, the worst-case complexity is equal to $O(|V||E|)$. However, for many real-world graphs, the algorithm is terminates in a much smaller amount of steps. Moreover, the algorithm has a significant parallel potential: its parallel complexity is equal to $O\left(p\frac{|E|}{N}\right)$, where N is the number of processors being used.

5.3   Algorithm selection for target architectures

Before the implementation, one needs to select the algorithms, most suitable for all target architectures. Both KNL and NVidia GPU have a large number of cores with a relatively low clock rate. If Dijkstra's algorithm, which is strictly sequential, is used for computations, all those cores will be idle, and, in addition, it will be very difficult to handle a stack or queue complex data structures on cores with a low clock rate. At the same time, Bellman-Ford algorithm does not require a processing of complex data structures; moreover, on any iteration this algorithm performs a parallel traversal of all graph edges. It will be shown later, that those properties will compensate algorithm's greater arithmetical complexity. In addition, it is possible to develop Bellman-Ford algorithm modification, which allows to process graphs with a size larger than the amount of available memory. This property is very important advantage for architectures with a limited amount of available memory, such as GPUs.

Before implementing the chosen algorithm, it is important to determine the storage data format for input graphs. For Bellman-Ford algorithm, the most suitable format is an edges list, where each edge is stored as a triple {vertex-start, vertex-end, edge's weight}; all edges are stored in a single array in any order.

5.4   GPU implementation

5.4.1. Basic version

CUDA-kernel, implementing the basic version of Bellman-Ford algorithm for the GPU is presented in listing 1:

Listing 1: Bellman-Ford algorithm's CUDA-kernel

```
 1  register const long long idx = blockIdx.x * blockDim.x + threadIdx.x;
 2
 3  if (idx < _edges_count) // for all graph edges do
 4  {
 5      register int src_id = _src_ids[idx];
 6      register int dst_id = _dst_ids[idx];
 7      register _TEdgeWeight weight = _weights[idx];
 8
 9      register _TEdgeWeight src_distance = __ldg(&_distances_for_read[src_id]);
10      register _TEdgeWeight dst_distance = __ldg(&_distances_for_read[dst_id]);
11
12      if (dst_distance > src_distance + weight)
13      {
14          _distances_for_write[dst_id] = src_distance + weight;
15          _modif[0] = _iter + 1;
16      }
17  }
```

The presented non-optimized kernel fully corresponds to the classical version of Bellman-Ford algorithm. The kernel is executed on number of threads, equal to the graph edges count. Each thread gets its corresponding edge's data (5) (6) (7), then reads current distance data of source and destination vertices of corresponding edge (9) (10). If those values minimize current distance to the destination vertex, the array of distances (12),(14) is updated. In figure 1 the results of profiling (obtained with NVidia Visual Profiler) of the kernel are presented, clearly demonstrating kernel's disadvantages.
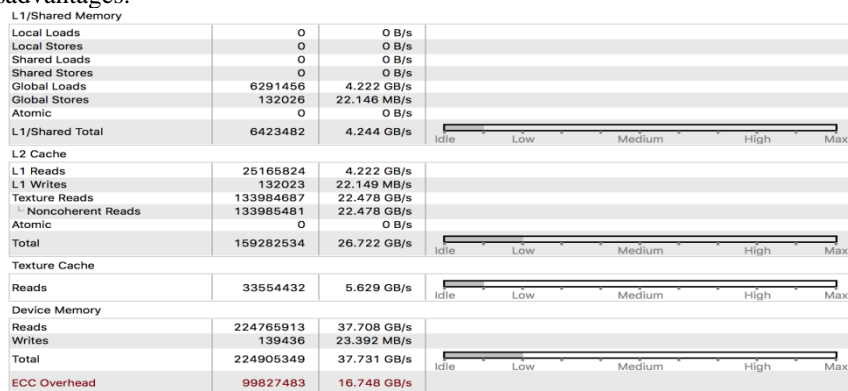
L1/Shared Memory			
Local Loads	0	0 B/s	
Local Stores	0	0 B/s	
Shared Loads	0	0 B/s	
Shared Stores	0	0 B/s	
Global Loads	6291456	4.222 GB/s	
Global Stores	132026	22.146 MB/s	
Atomic	0	0 B/s	
L1/Shared Total	6423482	4.244 GB/s	Idle Low Medium High Max
L2 Cache			
L1 Reads	25165824	4.222 GB/s	
L1 Writes	132023	22.149 MB/s	
Texture Reads	133984687	22.478 GB/s	
└ Noncoherent Reads	133985481	22.478 GB/s	
Atomic	0	0 B/s	
Total	159282534	26.722 GB/s	Idle Low Medium High Max
Texture Cache			
Reads	33554432	5.629 GB/s	Idle Low Medium High Max
Device Memory			
Reads	224765913	37.708 GB/s	
Writes	139436	23.392 MB/s	
Total	224905349	37.731 GB/s	Idle Low Medium High Max
ECC Overhead	99827483	16.748 GB/s	

Fig. 1. Analysis of memory bandwidth usage for the basic kernel implementation.

6

A first important observation is that this kernel is memory-bound, since every two arithmetic operations are followed by 6 memory access operations. Second, there is an indirect addressing during (9),(10),(12) and (14) memory accesses, where the optimal memory access pattern for the GPU is violated (coalesced memory access). In addition, values src_id and dst_id may point into completely different memory locations, what prevents efficient using of GPU caches.

The results of profiling clearly demonstrate the main reason of basic kernel's low performance — inefficient usage of GPU memory bandwidth ("device memory total" metric, due to non-coalesced memory accesses), as well as weak usage of L1 and L2 caches (due to weak locality of data accesses). These problems can be avoided with graph storage format optimization, which allows changing memory access pattern, making it more suitable for GPU architecture.

5.4.2. Graph storage format optimization

In the current section, the main optimization (graph edges reordering) is described. It allows improving memory access pattern, to achieve higher performance, since the data with indirect memory accesses will be placed more locally and stored in the caches. Modern K40 GPUs of Kepler architecture are equipped with 64 KB of L1 cache and 1.5 MB of L2 cache. The distances array has 1 MB size for a graph with 2^{18} vertices, 2 MB for a graph with 2^{19} vertices; so, even for medium-scaled graphs, the whole distance array can't be fully placed in caches.
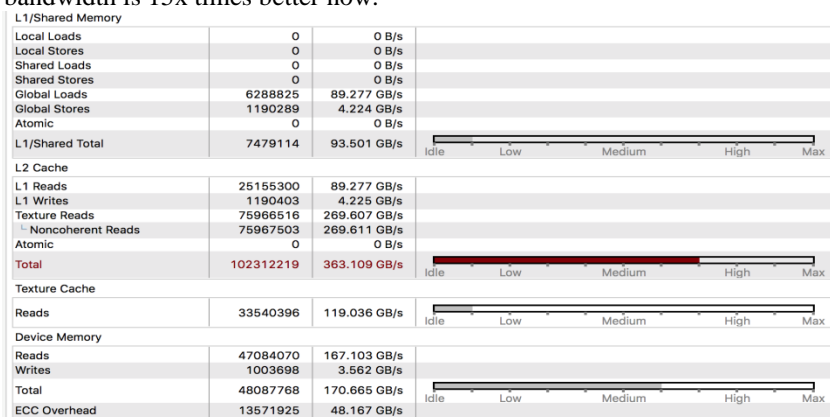
That is why edges rearrangement strategy is used to make sure that the data from distances arrays remains in cache memory as long as possible. The reordering process is illustrated in Figure 3. A similar reordering approach is described in [7].

An array of distances is divided into segments (red, green and blue colors – in Figure 2), which size is equal to the size of the lowest level cache - L2 for GPU (size 2 on figure 2). After that, the edges are placed into the array in the following way: in the beginning of the array edges are stored, which source vertices belong the first segment of the distances array, then to the second, then to the third. Edges with the same segment number are sorted with the similar strategy, applied to their destination vertices.
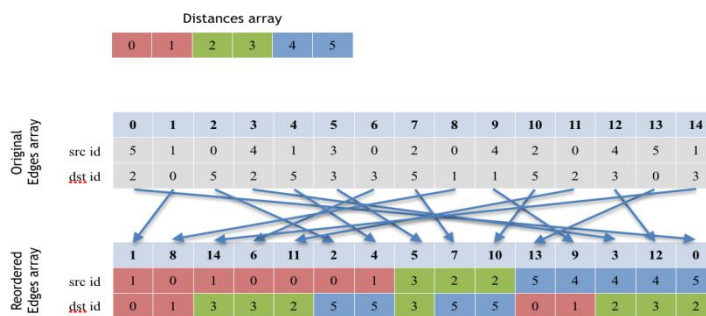
Fig. 2. Graph edges reordering example for graph with 5 vertices and 16 edges.

Due to described sorting approach, threads from the same warp will be accessing data within one or two segments; this result into a smaller number of different memory cells accessed by a single warp. Without this optimization, 32 different memory cells could be accessed, which would lead to a 32-times slowdown. The profiling report of optimized kernel is presented on figure 3. It demonstrates almost 5 times increase in used memory bandwidth (device memory reads / total). In addition, for the threads from a single block, distance array data will be stored in L2 cache for a much longer time, which can be also observed on presented profiling report: L2 cache used bandwidth is 15x times better now.

| L1/Shared Memory			
Local Loads	0	0 B/s	
Local Stores	0	0 B/s	
Shared Loads	0	0 B/s	
Shared Stores	0	0 B/s	
Global Loads	6288825	89.277 GB/s	
Global Stores	1190289	4.224 GB/s	
Atomic	0	0 B/s	
L1/Shared Total	7479114	93.501 GB/s	Idle Low Medium High Max
L2 Cache			
L1 Reads	25155300	89.277 GB/s	
L1 Writes	1190403	4.225 GB/s	
Texture Reads	75966516	269.607 GB/s	
└ Noncoherent Reads	75967503	269.611 GB/s	
Atomic	0	0 B/s	
Total	102312219	363.109 GB/s	Idle Low Medium High Max
Texture Cache			
Reads	33540396	119.036 GB/s	Idle Low Medium High Max
Device Memory			
Reads	47084070	167.103 GB/s	
Writes	1003698	3.562 GB/s	
Total	48087768	170.665 GB/s	Idle Low Medium High Max
ECC Overhead	13571925	48.167 GB/s	

Fig. 3. Analysis of memory bandwidth of basic kernel with optimized graph format storage

5.4.2. GPU implementation results

The performance comparison between basic and fully optimized kernel versions (with graph storage format optimization) is demonstrated on figure 4. Another important GPU-program characteristic is the percentage of program execution time, spent for data transfers between host and device. That is why fully optimized version is represented with two curves: with and without time spent for memory copy operation. The performance is measured with TEPs metrics — number of traversed edges per second (the edge is considered "traversed" when the data about it's source and destination vertices is requested). RMAT graphs with average connection count equal to 32 and vertices count from 2^{15} to 2^{24} are used as input data.

8
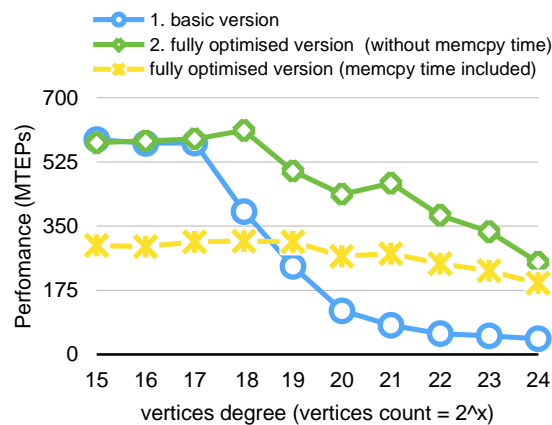
Fig. 4. Performance comparison of GPU Bellman-Ford algorithms versions, RMAT graphs with $2^{15} — 2^{24}$ vertices.

Figure 4 demonstrates two important implementation properties. First, non-optimized and optimized versions have similar behavior on graphs with size less than 2^{18}, since on smaller sizes graphs corresponding to the distances array can be fully placed into L2 cache of GPU. Second, the presented results demonstrate, that data transfers between host and device indeed require a significant amount of time. However, in other shortest paths problem variations (APSP, SPSP), data transfers will be less important, since more computations will be performed after coping data into device memory.

5.5   KNL implementation

First of all, it is important to decide which parallel technology should be used for KNL implementations [9]. The two most widely-used technologies are openMP and Intel TBB. Experimental results confirm that openMP technology is more suitable for graph algorithms implementations, since it requires fewer overheads for threads creation and synchronizations.

Simple openMP implementation is universal, since it can be compiled and executed on both classic CPUs and on Intel Xeon co-processors. However, even for the simplest version it is important to take into an account some KNL features, discussed below.

First, the threads creation must be performed only once in the beginning of the algorithm. Second, the number of synchronizations between threads should be minimized, since those synchronizations are extremely expensive with a larger number of parallel threads (60-70 for KNL). Last, it is important to select thread scheduler correctly between static, guided and dynamic thread scheduling policies. VTune Amplifier analysis on figure 5, demonstrates the crucial difference between static and guided modes.
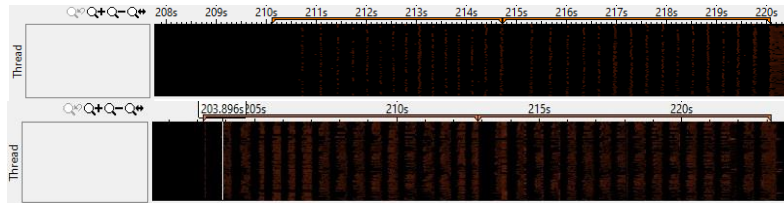
Fig. 5. Threads occupation analysis for static (top) and guided (bottom) modes, red color shows threads stall time.

In addition, Intel KNL has two types of memory: DDR4 and MCDRAM. The simplest usage of high-performance MCDRAM memory is possible with the following command: *numactl -m 1 ./program_name*, where 1 is the MCDRAM memory node number. Also, hbwmalloc library can be used to allocate MCDRAM memory region inside the program; it allows allocating only certain arrays in high-bandwidth memory, if program memory requirements are larger than MCDRAM memory size. It can be very useful for large-scale graph processing, where only the distances array can be stored in MCDRAM memory, while edges arrays can be stored in usual DDR4 memory with larger size.
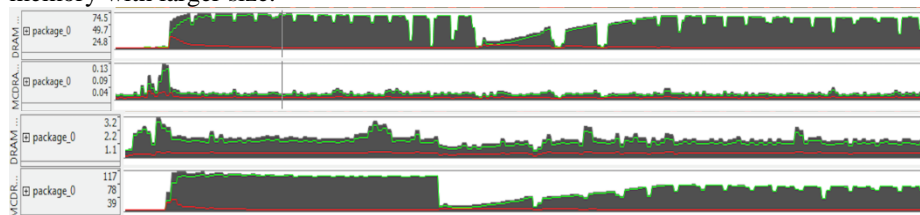
Fig. 6. Memory throughput usage analysis for different types of launches: program launched on MCDRAM node (bottom) and DDR4 node (top)

As second optimization, a similar reordering of graph edges (discussed in section 5.4.2) was performed. Segment size was chosen equal to KNL L2 cache size, devided on two (since L2 cache is shared by 2 cores in Tile).

The last important optimization was vectorization. An important feature of vectorization is the possibility to manually load distance data into the cache using *_mm512_prefetch_i32extgather_ps* instructions. As a result, vectorization allowed to achieve in average 1.5 times acceleration, which can be observed on figure 7.

10
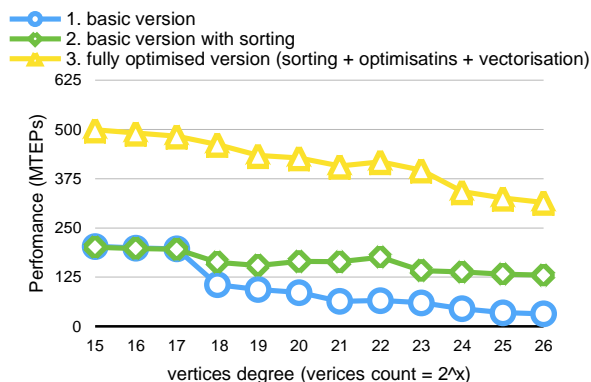
Fig. 7. Performance comparison of KNL Bellman-Ford algorithm implementations for KNL, RMAT graphs with 2^{15} — 2^{26} vertices.

5.6 GPU and KNL implementations comparison

Current section demonstrates general comparison of the two architectures in the context of solving shortest paths problem. The two GPU implementations are presented: with and without memory copies from host to device and back. For Intel KNL, the most optimized version with vectorization is presented. All graphs used for testing have RMAT and SSCA-2 structure and average connections count equal to 32.

Results from figure 8 demonstrate, that, first, KNL is able to process graphs with larger size. GPU is limited with 12 GB device memory, which can only contain graphs with 2^{24} vertices and 2^{29} edges. KNL processors can be equipped with up to 384 GB memory, which is able to contain graphs with up to 2^{29} vertices and 2^{34} edges.
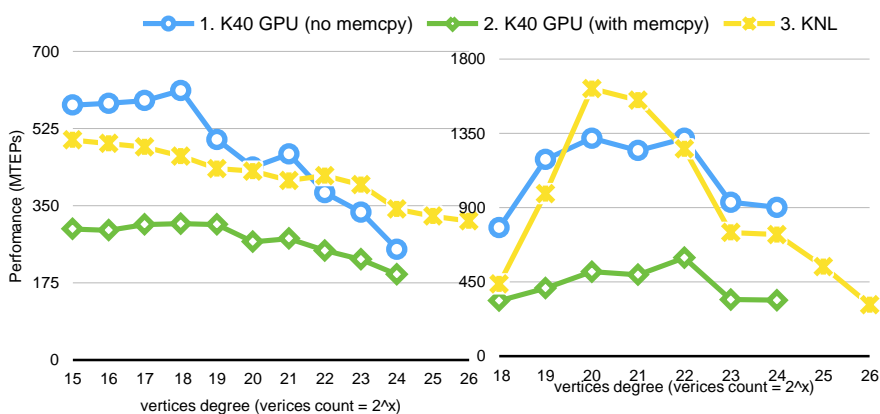
Fig. 8. Performance of Bellman-Ford algorithm implementations for different architectures. RMAT graphs with 2^{15} — 2^{26} vertices (left), SCCA-2 graphs with 2^{18} — 2^{26} vertices (right).

Second, GPU has better performance on small-scale RMAT graphs, since it requires less preprocessing before starting computations (no reallocation of aligned arrays and faster threads creation), but on large-scale RMAT graphs KNL show higher performance. For SSCA-2 graphs performance behavior is different, because of irregular size of those graphs cliques. As a result, the following conclusion can be made: KNL has better performance for large-scale graphs of both types, and is also capable of processing significantly larger graphs.

6   Strongly connected components

6.1   Mathematical description

A directed graph $G = (V, E)$ with vertices $V = (v_1, v_2, ..., v_n)$ and edges $E = (e_1, e_2, ..., e_m)$ is given. Edges may not have any data assigned (so graphs without edges weights are discussed in the current section). A strongly connected component (SCC) of a directed graph G is a strongly connected subgraph, which is maximal within the following property: no additional vertices from G can be included in the subgraph without breaking its property of being strongly connected.

6.2   Algorithm descriptions

Strongly connected components can be found with one of the following algorithms.

— **Tarjan's algorithm** is based on a single depth first search (DFS) and uses $O(|E|)$ operations. Due to the fact that the algorithm is based on the DFS, only a sequential implementation is possible.
— **The DCSC algorithm** (Divide and Conquer Strong Components), or FB (Forward-backward) is based on BFS and requires $O(|V| * \log(|V|))$ operations. This algorithm is initially designed for parallel implementations: at each step it finds a single strongly connected component and allocates up to three subgraph, each of which may contain other strongly connected components, and, as a result, can be processed in parallel.
— **Variations of the DCSC algorithm**, such as Coloring and FB with step-trim. These modified versions of the DCSC algorithm are described in detail in papers [5] and [6].

6.3   Algorithm selection for target architectures

For obvious reasons, Tarjan's algorithm is not suitable for solving SCC problem on parallel architectures, since it is based on a depth first search, as well as complex data structures (stack and queue) processing, which can not be implemented efficiently on GPUs.

A large number of papers, such as [6], have already investigated different variation of DCSC algorithms, which can be more or less effective for different types of

12

graphs; paper [6] concludes that the forward-backward-trim algorithm is the most efficient way to process RMAT graphs; the same was also proved during the current research.

The Forward-Backward-Trim algorithm is designed in the following way: in the first step, the removal of the strongly connected components of size 1 is performed. After that, on each step the algorithm finds one nontrivial strongly connected component and allocates up to three subgraphs, each of which contains other components, and, more important, can be processed in parallel. This step heavily relies on breadth-first search to find all vertices, which can be reached from the selected pivot, and all vertices, from which the current pivot vertex can be reached. Thus, this algorithm has two levels of parallelism: "BFS level" and "parallel subgraphs handling level", which appears to be a big advantage for target parallel architectures.

6.4   GPU implementation

A forward-Backward-Trim algorithm is based on three important stages — a trim step, a pivot selection and BFS in selected subgraphs. At the trim step the number of edges, adjacent to each vertex (equal to number of incoming and out-coming edges), is calculated, with a removal of vertices, which incoming or outgoing degrees are equal to zero. Since the graph is stored in edges list format, these values can be computed using new atomic operations, added in Kepler architecture. Random pivot selection can be implemented with a simple kernel, based on random nature of thread execution. Breadth-first search can be performed by the algorithm, similar to Bellman-Ford shortest paths computations. The downward is that it has a higher computational complexity, compared to the traditional BFS algorithm (using queues), but for RMAT graphs the efficiency of proposed approach has already been demonstrated.

Since all steps can be implemented for a graph, which is stored in edges list format, this format is selected again for graph storage. Since FB algorithm will be performing BFS and trim steps both in original and transposed graphs, it is even more important to sort graph edges using approach, already discussed in 5.4.2 section. Without edges reordering, sub step performance (such as BFS) in transposed graph will be much lower, compared to the performance in original graph.

There is another way how this problem can be avoided — with a pre-processing transpose of the input graph before SCC operation (as proposed in [8]), but edges reordering is proved to be much more efficient for two reasons. First, edges reordering can be performed much faster on parallel architectures (such as KNL), since it can be based on parallel sorting algorithms, and does not require complex data structures (like maps or dictionaries) to be supported. Second, proposed reordering is universal for many different operations. For example, this reordering can be used to improve performance for shortest paths, breadth-first search, bridges and transitive closure computational problems. As a result, input graphs can be optimized right after generation, and stored in reordered intermediate representation to allow more efficient graph processing in the future. Figure 11 in next section demonstrates computational time difference between two approaches: when input graph is optimized and not.

It is important to notice the percentage of time, required for trim and BFS steps. Later it will be shown, that these values greatly differ for both architectures. For GPU architecture and RMAT graphs this ratio is approximately equal to 6:10; the trim step requires slightly less time, since atomic operations implementation in Kepler architecture is very effective.
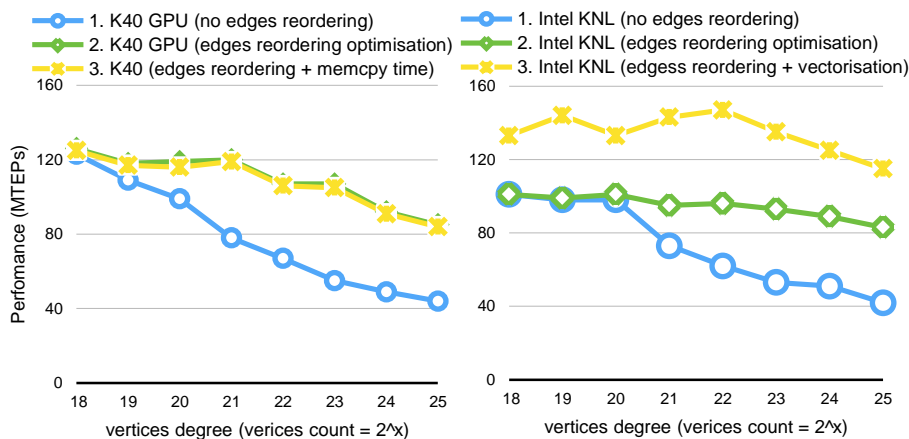
6.5   KNL implementation

First of all, it is important to study algorithms, implementing sub steps performance separately for all steps: trim, BFS and pivot selection. Trim step on KNL is executed in average 1.1-1.2x times slower, compared to GPU, since the openMP atomic operations appear to be less efficient compared to GPU ones. The breadth-first search, in contrary, can be implemented much more efficiently on KNL, using vectorization and similar to Bellman-Ford approach. Figure 9 demonstrate BFS-only step performance for single graph traversal; for RMAT graphs trim/BFS ratio is almost 1:1 on KNL.
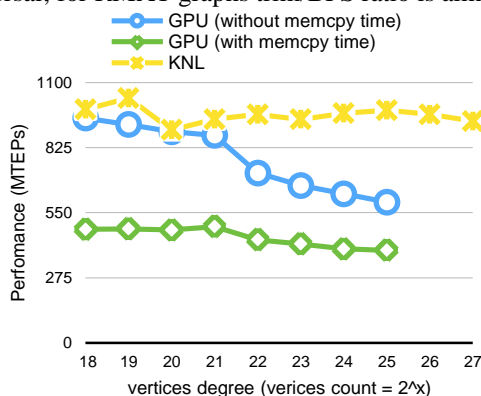
Fig. 9. Breadth-first search performance comparison for different architectures. RMAT graphs with 2^{18} — 2^{25} vertices.

As shown in Figure 9, the Intel KNL BFS implementation has significantly better performance on large-scale graphs, compared to GPU implementation
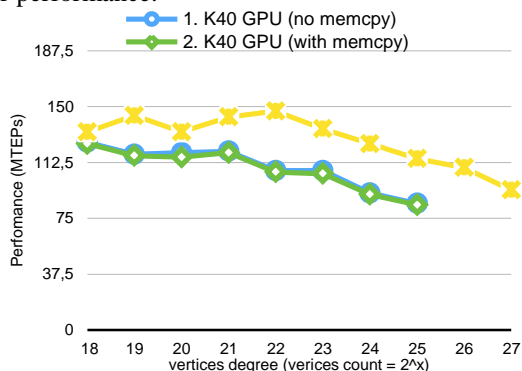
14

Fig. 10. Forward-Backward-Trim algorithm implementations performance for different architectures: NVidia GPU (left) and Intel KNL (right). RMAT graphs with 2^{18} — 2^{25} vertices.

6.6 GPU and KNL implementations comparison

Similar to previously discussed shortest paths problem, SCC algorithm implementation for Intel KNL is also capable of processing larger graphs (up to 2^{27}) vertices. Since strongly connected components problem doesn't require edges weights stored, this value is bigger, compared to shortest paths one (2^{26}). Inetl KNL also demonstrates slightly better execution time, since its BFS implementation for RMAT graphs demonstrates better performance.

Fig. 11. Forward-Backward-Trim algorithm implementations performance for different architectures. RMAT graphs with 2^{18} — 2^{27} vertices.

7   Conclusion

In the current paper, an implementation comparison of two important graph-processing problems on modern high-performance architectures (NVidia GPU and Intel KNL) has been discussed in details. Algorithms have been selected for both architectures, based on algorithm properties and target architecture features. As a result of many optimizations, high-performance and scalable parallel implementations have been created; moreover, the implementations have been examined in details using profiling utilities and theoretical research, which granted the ability to find potential bottlenecks and significantly improve final performance.

The best performance was achieved by Intel KNL processor for both investigated problems. Moreover, it was shown that Intel KNL is capable of processing much larger graphs with up to 134 million vertices and 42 billion edges. On K40 GPU, the maximum processed graph consisted from 33 million vertices and 10 billion edges. It is important, that Kepler architecture accelerators are currently outdated, while new GPUs from Pascal generation can achieve higher performance.

References

1. Pawan Harish and P. J. Narayanan. «Accelerating large graph algorithms on the GPU using CUDA». Center for Visual Information Technology, International Institute of Information Technology Hyderabad, INDIA.
2. Gary J Katz1, «All-Pairs-Shortest-Paths for Large Graphs on the GPU». Joe Kider. University of Pennsylvania.
3. Hector Ortega-Arranz, Yuri Torres, Diego R. Llanos, and Arturo Gonzalez-Escribano. «A New GPU-based Approach to the Shortest Path Problem». Dept. Informática, Universidad de Valladolid, Spain.
4. Tarjan, Robert Endre, and Uzi Vishkin. «An Efficient Parallel Biconnectivity Algorithm» SIAM Journal on Computing 14, no. 4 (1985): 862–74.
5. Fleischer, Lisa K, Bruce Hendrickson, and Ali Pınar. «On Identifying Strongly Connected Components in Parallel». In Lecture Notes in Computer Science, Volume 1800, Springer, 2000, pp. 505–11.
6. Jiri Barnat, Petr Bauch, Lubos Brim, Milan Ceska, « Computing Strongly Connected Components in Parallel on CUDA», 2011, IPDPS '11 Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium.
7. A. Kolganov., Evaluating GPU performance on data-intense problems (translated from Russian), Available online at http://agora.guru.ru/abrau2014/pdf/079.pdf
8. J. Barnat, P. Bauch. «Computing Strongly Connected Components in Parallel on CUDA». Faculty of Informatics, Masaryk University, Botanicka ́ 68a, 60200 Brno, Czech Republic.
9. R. Florian, Choosing the right threading framework, 2013, Available online at: https://software.intel.com/en-us/articles/choosing-the-right-threading-framework
10. Pore, A., «Parallel implementation of Dijkstra's algorithm using MPI library on a cluster», Available online at: http://www.cse.buffalo.edu/faculty/miller/Courses/CSE633/Pore-Spring-2014-CSE633.pdf