An Efficient Implementation of the Transitive Closure Problem on Intel KNL Architecture

Ilya Afanasyev

Lomonosov Moscow State University, Moscow, Russia
afanasiev_ilya@icloud.com

Abstract. An important trend in modern supercomputing is a frequent usage of co-processors, such as GPUs and Intel Xeon PHIs. The recent generation of Intel Knights Landing processors provide high performance computational power with a large amount of high-bandwidth memory, what makes them a perfect platform for graph-processing. The presented study describes implementation approaches to large-scale graph processing on Intel KNL processors; as a sample problem, the transitive closure computation is discussed. Based on the joint analysis of algorithm properties and architecture features, the performance tuning has been performed, including graph storage format optimizations, efficient usage of memory hierarchy and vectorization. As a result, an optimized algorithm implementation for the transitive closure problem solution has been developed. The proposed implementation has been studied using different approaches, aimed at demonstrating advantages and disadvantages of Intel KNL architecture in solving graph-processing problems.

Keywords: transitive closure · Intel Knights Landing · graph processing

1 Introduction

An interest to large-scale graph processing is recently growing rapidly, since graph data structure perfectly emulate real-world objects and connections between them. Some examples of such objects are social and infrastructural (energy and transport) networks, micro-biology, databases and social models, etc. The mentioned problems have an important common property – the corresponding graphs have very large size (up to millions vertices and billions edges); as a result, a parallel approach is required to process those data-structures in a reasonable amount of time. Transitive closure computation is one of the fundamental graph-processing problems, which can be used to evaluate vertex connectivity. The transitive closure computation problem can be applied in several already mentioned application fields, for example, in database systems modeling [1], [2].

2 The Mathematical Description of the Problem

A directed graph $G = (V, E)$ with vertices V and edges E is given. The path $P(u, v)$ between vertices u and v is defined as a sequence of edges $e_1 = (u, w_1), e_2 =$

$(w_1, w_2), \ldots, e_k = (w_{k-1}, w_k)$, starting in vertex u and ending in vertex v, where edges are following each other. Vertex v is reachable from vertex u, if at least a single path $P(u, v)$ between vertices u and v exists (every vertex is considered reachable from itself).

Computing the transitive closure of graph G means obtaining graph $G^+ = (V, E^+)$, where an edge (v, w) from G belongs to E^+ if and only if vertex w is reachable form vertex v in graph G. As a result, the transitive closure problem solution requires $|V|^2$ storage space, so it can't be calculated using modern computers computational node's memory even for medium-sized graphs (starting with around 2^{20} vertices). For this reason the generalization of the transitive closure problem is used in current paper: only the specified pairs of vertices $(u_1, v_1), (u_2, v_2), \ldots, (u_n, v_n)$ are checked to belong to the transitive closure. The amount of the pairs required to check n becomes an additional flexible algorithm parameter: varying it may greatly affect the overall algorithm performance.

3 State of the Art: Algorithms and Implementations

Current section uses the following notations to evaluate the complexity of reviewed algorithms: $|V|$ corresponds to the count of vertices in input graph G, $|E|$ - to the count of edges. The transitive closure computation problem in directed graph G can be solved using three different traditional approaches, described below.

1. The transitive closure computation can be reduced to the shortest paths computation in a corresponding graph with identical weights. Consequently, it can be solved with Floyd-Warshall algorithm, introduced in [3] and [4]. This algorithm has $\mathcal{O}(|V|^3)$ computational complexity, and historically is the the first developed algorithm for the transitive closure problem solution [5]. An important property of Floyd-Warshall algorithm is $\mathcal{O}(|V|^2)$ memory requirement for computations, what immediately reduces its applicability only to small-scale graphs.
2. The transitive closure can be obtained using several multiple breadth-first searches (BFS), executed from each vertex of the graph. BFS algorithm has been first described in [6]; the proposed algorithm allows checking reachability between the selected source vertex and other vertices of the graph with only $\mathcal{O}(|E| + |V|)$ operations required (using a queue-based implementation approach). As a result, the full transitive closure computation requires $\mathcal{O}(|V| * (|V| + |E|))$ operations.
3. Among the reviewed approaches, the most optimal computational complexity has Purdom's algorithm, introduced in [7]. Purdom's algorithm is based on the following idea: the transitive closure computation for graph G can be reduced to the transitive closure computation for graph G^-, obtained from graph G by collapsing G's strongly connected components into G^- vertices. The described approach provides $\mathcal{O}(|E| + u|V|)$ computational complexity, where u is the count of edges in graph G^-. The provided estimate is based on

the assumption, that asymptotically optimal Tartan's algorithm ($\mathcal{O}(|E|)$) is used for strongly connected components computation; if another algorithm, such as DCSC[1] is used, computational complexity can be different.

There are already many efficient implementations of mentioned algorithms for different parallel architectures, including multicore central processors, co-processors and graphic accelerators (GPUs). The brief review of these implementations is following.

Breadth-first searche can be considered as one of the most well-studied graph-processing algorithms. Not only BFS algorithm allows solving the transitive closure problem with repeated calls from each vertex of the input graph, but it is also a very important building-block of Purdom's algorithm. The first approach to parallel BFS implementation on co-processors (GPU) was discussed in [8], where quadratic complicity parallelization strategy (in the worst case) was used. The proposed method was completely revised in [9], where it was shown to be not very efficient for non-RMAT graphs; to solve this problem, this paper proposes $\mathcal{O}(|V| + |E|)$ complexity parallel GPU-algorithm, which achieves much better performance for various types of graphs. Parallel BFS implementations on multi-core central processes are usually based on the parallel queues approach, and can be found, for example, in [17]. Moreover, for Intel Xeon PHI processors a few implementation attempts were made, such as [19] or [20].

Some efficient approaches to parallel implementation of Floyd-Warshall algorithm for GPUs are discussed in [15], [16]. These papers also highlight the main GPU flaw for this algorithm - the lack of device memory, required to store all necessary computational data.

Purdom's algorithm implementations on parallel architectures are less well-studied; an approach to implementation for central processors is described in [18]. Also, Purdom's algorithm is based on strongly connected components search operation, which can be solved by several parallel algorithms, which are currently well-investigated both for CPUs and GPUs in [10], [11].

4 Purdom's Algorithm Parallel Properties Research

Based on computational complexity estimates, the most suitable approach to solve the transitive closure problem is Purdom's algorithm. However, during the selection of the most suitable algorithm for particular parallel architecture, it parallel algorithm's properties have to be studied. For this purpose, information graphs, introduced in [14], can be used.

Figure 1 (left) demonstrates information graph of Purdom's algorithm. The presented graph is rather complicated and therefore includes two subgraphs, each one corresponding to an important algorithm building-block: breadth-first

[1] Divide and Conquer Strong Components (DCSC, also known as Forward-Bakward or Forward-Backward-Trim algorithms), a family of algorithms based on recursive partitioning of graph into disjoint sets, which on the lowest level of the recursion contain a single strongly connected component in the each set.

search (BFS) and strongly connected components (SCC) computation. Informational graph of parallel queue-based breadth-first search is demonstrated on figure 1 (right), while figure 2 showes information graph of Forward-Backward-Trim algorithm, used as the main algorithm for parallel SCC computation in current paper.

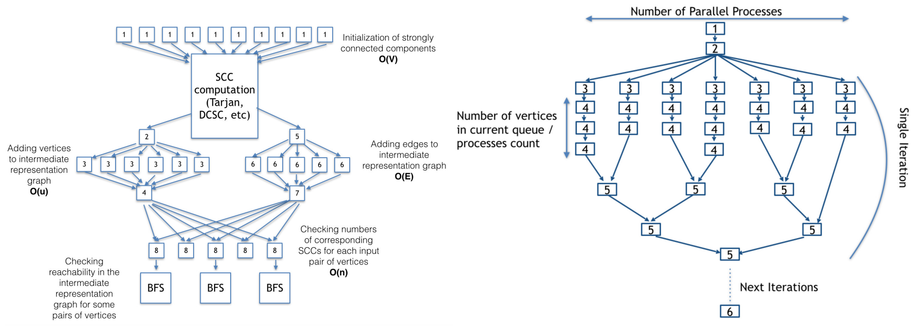
Initialization of strongly connected components
O(V)
SCC computation (Tarjan, DCSC, etc)
Adding vertices to intermediate representation graph
O(u)
Adding edges to intermediate representation graph
O(E)
Checking numbers of corresponding SCCs for each input pair of vertices
O(n)
Checking reachability in the intermediate representation graph for some pairs of vertices
BFS
Number of Parallel Processes
Number of vertices in current queue / processes count
Single Iteration
Next Iterations

Fig. 1. Purdom's algorithm information graph (left), Breadth-first search informational graph (right)

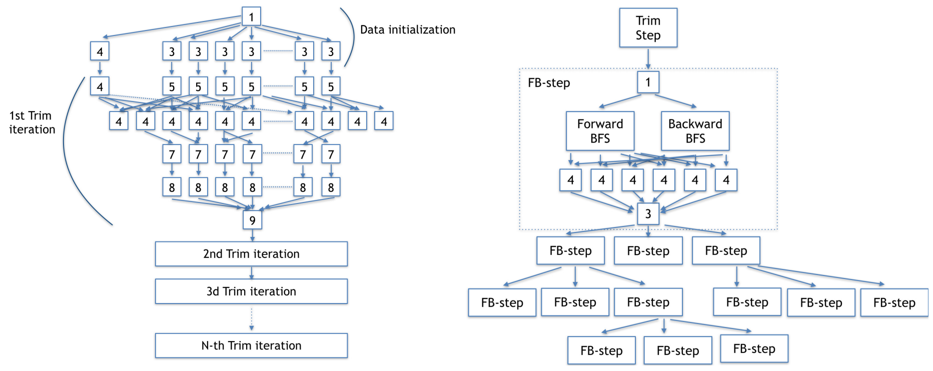

Fig. 2. Forward-Backward-Trim information graph: top level (left), bottom-level (right)

The reviewed information graphs vividly demonstrate that Purdom's algorithm has a significant parallelism potential ($\mathcal{O}(|E|)$ or $\mathcal{O}(|V|)$ operations) on each level. Since $\mathcal{O}(|E|)$ and $\mathcal{O}(|V|)$ values are extremely huge for large-scale graphs, this algorithm can be reliably chosen for Intel KNL architecture.

5 The Main Features of KNL Architecture

Intel Knights Landing (KNL) is one of the newest architectures of Intel Xeon Phi coprocessors. These processors are equipped with up to 72 cores, each one with a relatively low clock signal rate of 1.3-1.5 GHs and an ability to efficiently execute up to 4 threads (hyperthreading technology). As a result, Intel KNL processor is able to achieve up to 6 TFLOPs performance on single precision computations and 2.6 TFLOPs on double precision. Target processor's memory architecture is even more important than peak performance for graph-processing, since graph problems are usually memory-bound. Intel KNL has two memory levels: high-bandwidth MCDRAM memory with 16 GB capacity and 400 GB/s bandwidth, and DDR4 memory with 384 GB capacity (in the best available configuration) and 90 GB/s bandwidth. Processor's cores are grouped by pairs into tiles; each tile shares common 1 MB L2 cache, while each core has its own 64 KB L1 cache.

AVX-512 vector instructions support is an another important new feature of Intel KNL generation processors. These instructions allow simultaneous processing of 16 single precision variables, and, what is more important, introduce gather and scatter operations support, which is crucial during indirect memory accesses vectorization.

6 Implementation Approach

Current section describes Purdom's algorithm implementation approach for Intel KNL architecture. The developed algorithm can be divided into 4 separate stages (steps):

1. strongly connected detection in input graph G,
2. creation of intermediate representation graph G^-,
3. computing an answer for all vertex pairs, related to the same strongly connected component,
4. computing an answer for the rest pairs, using parallel BFS in the intermediate representation graph G^-.

If required, intermediate representation graph can be saved to hard drive after stage 1, so later the transitive closure can be found more efficiently (no repeated SCC computation). In current section, some implementation approaches for each algorithm stage will be described together with optimizations for each step.

Parallel Forward-backward-Trim algorithm is implemented according to the approach described in [10], [11] with a few differences. First, in order to avoid building reverse (transposed) graph, what is necessary for efficient backward search implementation, graph is converted to edges list storage format. To further increase the performance, the input edges list can be pre-sorted: the basic idea of this sort is reordering graph edges the way, that edges stored in adjacent memory cells start pointing to adjacent cells in reachability array. The described approach allows to greatly improve data locality and L2 cache usage. Moreover, this optimization also significantly improves trim step efficiency, since it has similar memory access pattern. The described sorting is not required for intermediate representation graph G^{-}, since usually these graphs have significantly smaller count of vertices compared to original input graphs. As a result, corresponding reachability arrays of intermediate representation graphs usually fit into L2 cache. Table 1 demonstrates comparison between sizes of original input graphs and graphs of corresponding intermediate representations.

Table 1. The comparison of sizes between original input graphs and corresponding intermediate representation graphs

Count of vertices in the original graph	Distances array size in the original graph	Count of vertices in the intermediate representation graph	Distances array size in the intermediate representation graph	Graph type
1m	1 MB	14k	13 KB	RMAT[12]
8m	4 MB	659k	0.6 MB	RMAT
33m	35 MB	3.4m	3.2MB	RMAT
1m	1 MB	46k	44KB	SSCA2[13]
8m	4 MB	366k	357KB	SSCA2
33m	35 MB	1.45m	1.3MB	SSCA2

The second stage includes the intermediate representation graph generation. This stage has the following structure: first, the count of strongly connected components is calculated; after that, non-dublicate edges (connecting different SCCs) are added to the intermediate representation graph, which has the number of vertices equal to SCC count. In the simplest case, both these operations are executed sequentially, since they require processing map-like data structures. But it is also possible to perform these operations in parallel, providing a separate data structure to each process, followed by a sequential merge of these data structures on the root processor (the latter approach is used in current paper). Another alternative is tree-like structure of inter-processes merges, but our tests demonstrated worse performance of this approach compared to sequential merges. It is also important to select an optimal graph storage format for intermediate representation graph. This format is not necessary has to be the same as the input graph format, since SCC and BFS operations require different supported operations during the computation process. The edges list format allows only the quadratic BFS parallelization (since it doesn't support traversal of adjacent vertices from the selected one), while adjacency list format allows queue parallelization approach. These two approaches may result in the very different performance values of BFS for intermediate representation graphs, what will be demonstrated in the next section.

The third stage contains verification of input pairs of vertices for the property, if these vertices belong to the same strongly connected component. This stage has a relatively small computational complexity $\mathcal{O}(n)$ and, moreover, can be efficiently parallelised (verification for each pair is independent). The last step includes obtaining an answer for the rest pairs of vertices, which appear to belong to different components. This step is based on BFS searches, performed from each source-vertex in intermediate representation graph. It is important to notice, that edges list format is also used for intermediate representation graph storage, since those graphs usually have low diameter compared to original input graphs.

To investigate the implementation bottlenecks, it is necessary to measure execution times for each stage, while comparing the values between each other. For different graph types and different numbers of input pairs, these values may differ a lot; in the next table, RMAT, SSCA-2 and random uniform graphs with 2^{23} vertices and average vertex-degree 32 are used, among with $10k$ pairs of vertices to be checked. Table 2 demonstrates the percentage of time, spent on each execution stage for the most optimized algorithm version.

Table 2. Percentage of time, required for each algorithm step

Graph type	SCC computation (step 1)	Creating Intermediate Representation (step 2)	BFS and checks (step 3,4)
RMAT	$41,2\%$	$8,9\%$	$49,5\%$
SSCA-2	$98,6\%$	$0,497\%$	$0,857\%$
Random uniform	88.1%	10.2%	1.7%

For the whole algorithm it is very important to use high-performance MCDRAM memory, what can be achieved by two different approaches: for small and medium scaled graphs, the program can be executed only in MCDRAM memory space. For the graphs which size exceeds 16 GB, only the important arrays (like distances arrays on stage 1 or intermediate representation graph data) can be stored in MCDRAM memory. In the next section a performance comparison for MCDRAM and usual DDR4 memory modes is demonstrated.

7 Performance Analysis

Current section includes the performance analysis for Purdom's algorithm. The first important efficiency metric is the algorithm's performance dependance from the size of the input graph G. The performance is defined with TEPS (traversed edges per seconds) metric, which demonstrates the graph-processing efficiency with the increase of graph size. Usually, the performance decreases because of the less and less efficient usage of the cache hierarchy with the growing graph size. An edge is called "traversed" during algorithm execution when it's corresponding data is loaded from memory. The described dependency is demonstrated on figure 3 for several versions of the developed algorithm: the basic non-optimized version(where all graphs are stored in edges list format), version with the intermediate representation graph optimization (when it is stored in the adjacency list format), version with input graph edges reordering, and version with MCDRAM memory and vectorization usage.

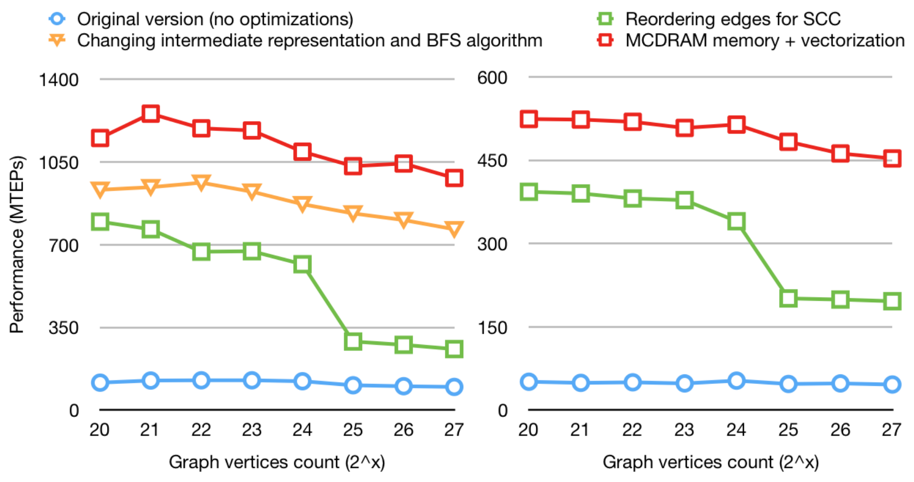

Fig. 3. Purdom's algorithm implementations performance for different optimizations included. RMAT graphs with 2^{18} - 2^{27} vertices (left), SSCA2 graphs(right)

Since Intel KNL provides hyperthreading technology support, it is necessary to check on practice if the usage of $4x$ threads provides a significant performance increase. In current paper, launching 272 threads on 68 cores provides the best achieved performance. An another important metric is an acceleration of the most optimized developed parallel algorithm, compared to the sequential versions (for KNL and usual multi-core CPUs). On this figure multi-core Intel(R) Xeon(R) CPU E5-2697 v3 CPUs have been used for testing. The comparison demonstrated on figure 4 allows evaluating how efficiently the parallel algorithm utilizes parallel resources of the target architecture.

In the conclusion, it is important to research the sources of possible bottlenecks in the developed algorithm, since it allows to prove that the algorithm is implemented efficiently, and, moreover, to highlight Intel KNL architecture advantages and disadvantages for graph-processing. Since transitive closure computation belongs to memory-bound problem category, it is necessary to study the achieved memory throughput during the program execution. During SCC and BFS stages, the memory throughput achieved is approximately 200 GB/s, which is half of the maximum MCDRAM bandwidth. Generating the intermediate representation step has much lower memory throughput used, since on this step the program operates with complex map and vector data structures scattered in the memory, and, as a result, has poor data locality.

8 Conclusion

In current paper, a parallel implementation of the transitive closure computation problem for Intel KNL processors has been proposed and discussed in details. To

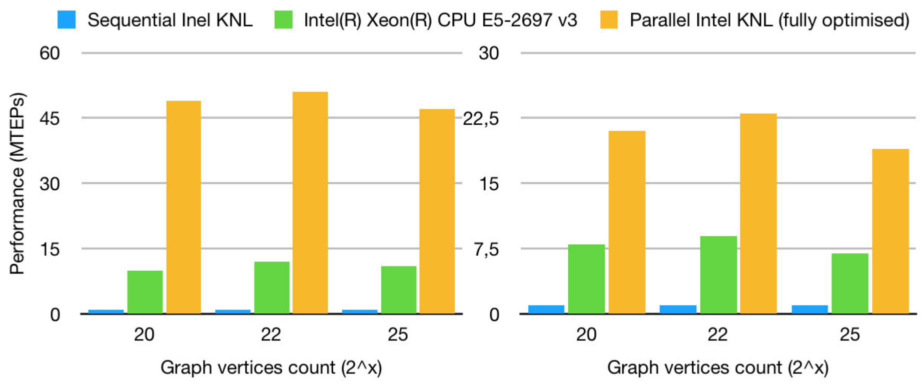

Fig. 4. An acceleration of the optimized parallel Purdom's algorithm compared to sequential version. RMAT graphs (left), SSCA2 graphs (right)

solve the problem, Purdom's algorithm has been selected, since it has an optimal computational complexity and non-demanding memory requirements among the reviewed algorithms. Moreover, presented information graphs demonstrate that the selected algorithm have significant parallelism resources, which can be efficiently utilised on a highly-parallel architecture, such Intel KNL.

Based on the algorithm selection, its basic version for Intel KNL architecture has been implemented. After that, a lot of optimizations, including graph edges sorting, reducing the amount of data loaded from processor memory, usage of MCDRAM memory and vectorization has been applied to tune the performance of the proposed implementation. As a result, a high-performance and scalable parallel implementation of Purdom's algorithm has been developed. This implementation demonstrates almost a $50x$ acceleration compared to sequential implementation on Intel KNL, and a $5x$ acceleration compared to sequential implementation on Intel(R) Xeon(R) CPU E5-2697 v3.

Moreover, it was shown that Intel KNL is capable of processing very large graphs with a size up to 134 million vertices and 42 billion edges, what significantly exceeds other coprocessors results.

Acknowledgments. The results were obtained in the Lomonosov Moscow State University with the financial support of the Russian Science Foundation (agreement № 17-71-20114).

References

[1] S. Dar. Augmenting databases with generalized transitive closure. PhD thesis, Department of Computer Science, University of Wisconsin, Madison, 1994.

[2] M. Yannakakis. Graph-theoretic methods in database theory. In Proc. of the 9th ACM SIGACT-SIGMOND-SIGART Symposium on Principles of Database Systems, pages 230-242. ACM, 1990.

[3] Floyd, Robert W. Algorithm 97: Shortest Path. Communications of the ACM 5, no. 6 (June 1, 1962): 345. doi:10.1145/367766.368168.

[4] Warshall, Stephen. A Theorem on Boolean Matrices. Journal of the ACM 9, no. 1 (January 1, 1962): 11-12. doi:10.1145/321105.321107.

[5] Roy, Bernard. Transitivit Et Connexit. Comptes Rendus De l'Acadmie Des Sciences 249 (1959): 216-218p.

[6] Lee, C Y. An Algorithm for Path Connections and Its Applications. IEEE Transactions on Electronic Computers 10, no. 3 (September 1961): 346-65. doi:10.1109/TEC.1961.5219222.

[7] Purdom, Paul, Jr. A Transitive Closure Algorithm. Bit 10, no. 1 (March 1970): 76-94. doi:10.1007/BF01940892.

[8] 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.

[9] Hector Ortega-Arranz, Yuri Torres, Diego R. Llanos, and Arturo Gonzalez-Escribano. A New GPU-based Approach to the Shortest Path Problem. Dept. Informatica, Universidad de Valladolid, Spain.

[10] Fleischer, Lisa K, Bruce Hendrickson, and Ali Pinar. On Identifying Strongly Connected Components in Parallel. In Lecture Notes in Computer Science, Volume 1800, Springer, 2000, pp. 505-511.

[11] J. Barnat, P. Bauch. Computing Strongly Connected Components in Parallel on CUDA. Faculty of Informatics, Masaryk University, Botanicka 68a, 60200 Brno, Czech Republic.

[12] Chakrabarti, Deepayan, Yiping Zhan, and Christos Faloutsos. R-MAT: A recursive model for graph mining. Proceedings of the 2004 SIAM International Conference on Data Mining. Society for Industrial and Applied Mathematics, 2004.?

[13] Bader, David A., et al. Hpcs scalable synthetic compact applications 2 graph analysis. SSCA 2 (2006): v2.

[14] Voevodin, V.V. Parallel Computing. 608p. BHV, St. Petersburg (2002). (in Russian)

[15] Katz, G. J., Kider Jr, J. T. (2008, June). All-pairs shortest-paths for large graphs on the GPU. In Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware (pp. 47-55). Eurographics Association.

[16] Buluc, Aydin, John R. Gilbert, and Ceren Budak. Solving path problems on the GPU. Parallel Computing 36.5 (2010): 241-253.

[17] Hong, S., Oguntebi, T., Olukotun, K. Efficient parallel graph exploration on multi-core CPU and GPU. In Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on (pp. 78-88). IEEE.

[18] Bloemen, Vincent. On-The-Fly parallel decomposition of strongly connected components. MS thesis. University of Twente, 2015.

[19] A.Frolov, E. Golovina, A. Semenov. Performance Evaluation of Breadth-First Search on Intel Xeon Phi. OAO “NICEVT”, 2016.

[20] M. Paredes, G. Riley, M. Lujan. Breadth First Search Vectorization on the Intel Xeon Phi. 2016.