Performance Evaluations of Graph Database using CUDA and OpenMP-Compatible Libraries

Shin Morishima[1] and Hiroki Matsutani[1,2,3]

[1]Keio University, 3-14-1 Hiyoshi, Kohoku-ku, Yokohama, Japan
[2]National Institute of Informatics, [3]Japan Science and Technology Agency PRESTO
{morisima,matutani}@arc.ics.keio.ac.jp

ABSTRACT

Graph databases use graph structures to store data sets as nodes, edges, and properties. They are used to store and search the relationships between a large number of nodes, such as social networking services and recommendation engines that use customer social graphs. Since computation cost for graph search queries increases as the graph becomes large, in this paper we accelerate the graph search functions (Dijkstra and A* algorithms) of a graph database Neo4j using two ways: multi-threaded library and CUDA library for graphics processing units (GPUs). We use 100,000-node graphs generated based on a degree distribution of Facebook social graph for evaluations. Our multi-threaded and GPU-based implementations require an auxiliary adjacency matrix for a target graph. The results show that, when we do not take into account additional overhead to generate the auxiliary adjacency matrix, multi-threaded version improves the Dijkstra and A* search performance by 16.2x and 13.8x compared to the original implementation. The GPU-based implementation improves the Dijkstra and A* search performance by 26.2x and 32.8x. When we take into account the overhead, although the speed-ups by our implementations are reduced, by reusing the auxiliary adjacency matrix for multiple graph search queries we can significantly improve the graph search performance.

1. INTRODUCTION

Recent advances on the Internet, network services, mobile devices, and information sensing devices produce a vast amount of data which are difficult to be processed by conventional database management systems or data processing applications. A visible example of big data sources is a social networking service which is now being a big infrastructure to build social relationships among people from all over the world. From an economical point of view, these services can be used for recommendation systems based on users' trust, in addition to making connections between users. In these services, a vast amount of users' profile and their relationships are stored in databases and various queries that update, scan, and search the social graphs are performed. As a target graph is large, the graph search performance becomes an important issue.

A graph database is a database system specialized for graph structures to store data sets as nodes, edges, and properties. They are used to store and search the relationships between a large number of nodes, such as social networking services and recommendation engines that use customer social graphs. Since graph search queries typically require high computation power, in this paper, we accelerate the graph search functions (Dijkstra and A* algorithms) of a graph database Neo4j by utilizing multi-core processors and graphics processing units (GPUs). More specially, we design graph search functions by using OpenMP compatible multi-threaded library and CUDA library for GPUs, and we integrate them into a practical graph database system. We use 100,000-node graphs generated based on a degree distribution of Facebook social graph for evaluations.

The rest of this paper is organized as follows. Section 2 introduces graph search algorithms used in graph databases and surveys prior works that accelerate graph algorithms using GPUs. Section 3 explains our multi-threaded and GPU-based graph search functions integrated for a graph database. Section 4 evaluates them using random and synthetic large graphs based on degree distribution of a social networking service. Section 5 concludes this paper.

This work was presented in part at the international symposium on Highly-Efficient Accelerators and Reconfigurable Technologies (HEART2014), Sendai, Japan, June 9-11, 2014.

2. BACKGROUND AND RELATED WORK

2.1 Graph Database and Algorithms

In this paper, as a target graph database, we use Neo4j which is one of the famous open-source graph databases implemented in Java language [5]. The following algorithms are used for graph search.

Dijkstra Algorithm. Dijkstra algorithm is a graph search algorithm that solves the single-source shortest path problem for a weighted graph. The way of solving the problem is searching the node in ascending order of weight from source node and extending the area where shortest path is found. Although typical Dijkstra algorithm searches the shortest paths from single source to every other nodes, in Neo4j the algorithm finds the shortest path between given source and destination pair. In this case, we can terminate the algorithm immediately when the shortest path between given source-destination pair is found, instead of examining all the destination nodes.

A Algorithm.* A* algorithm can be said as a variant of Dijkstra algorithm. An estimated cost to the destination is assigned to every node, and it preferentially explores the node with the minimum estimated cost.

Shortest Path. Shortest Path algorithm finds all the shortest paths between given source and destination pair. It explores paths between the source and destination pair in the ascending order of the hop count and terminates the algorithm when it finds a path whose hop count is longer than the minimum.

All Path. All Path algorithm finds all the paths between given source and destination pair whose hop count value is less than or equal to a specified hop count value. It is a variant of Shortest Path algorithm, in which we can specify the threshold hop count value to explore.

All Simple Path. All Simple Path algorithm is similar to All Path algorithm but it excludes paths that visit the same node more than once.

Although the termination conditions of Shortest Path, All Path, and All Simple Path algorithms are different, they can

be said as variants of Dijkstra algorithm that uses the constant cost for all the edges. In this paper, therefore, we focus on Dijkstra and A* algorithms in order to improve the graph search performance on Neo4j.

2.2 GPU-based Graph Processing

Several works show that graph processing algorithms can be accelerated with GPUs. Ortega-Arranz et.al. reported that Dijkstra algorithm can accelerated by using a GPU and achieved up to 220x speed-up with respect to the CPU [7]. Breadth-first search (BFS) is another important graph search algorithm. Merrill et.al. reported that BFS can be accelerated by using a GPU and achieved up to 29x speed-up with respect to the CPU [4]. Prim algorithm is used to build a minimum spanning tree that covers a given graph. Nobart et.al. reported that Prim algorithm can be accelerated by using a GPU and achieved up to 14x speed-up with respect to the CPU [6]. In this way, much works that accelerate such graph processing algorithms by using GPUs have been done so far.

There is a prior work that accelerates a simple key-value store database using a GPU, although it is not a graph database. Memcached is a kind of in-memory key-value stores widely used as a cache layer of various network services. Hertherington et.al. reported that the key-value store database can be accelerated by using a GPU [2]. By introducing zero-copy data transfer between GPU and host CPU, their GPU-based implementation achieves up to 33x speed-up with respect to the original implementation. However, if such zero-copy data transfer could not be used, the speed-up benefit would be reduced. We believe that computation cost of such key-value stores is relatively small and their performance is limited by I/O bandwidth, such as memory and network I/Os; thus key-value store applications are classified as I/O intensive. On the other hand, graph databases discussed in this paper perform graph search queries for large graphs, which is computation intensive; thus there is enough room to accelerate the graph search queries by GPUs.

However, to the best of our knowledge, there is no reports that practically apply such GPU-based graph processing to widely-used graph databases.

3. DESIGN AND IMPLEMENTATION

In this section, we show our multi-threaded and GPU-based graph search functions integrated for Neo4j.

3.1 Generation of Adjacency Matrix

Neo4j is implemented with Java language, in which nodes and edges are defined as Java classes to encapsulate these data and group the methods to access them. Although such implementation is highly modular and extendable for practical systems, it is not friendly to parallelize such data structures with parallel libraries, such as OpenMP and CUDA. Thus, in this work, we employ an auxiliary adjacency matrix that represents weight of every edge between two vertices (nodes) for parallel executions of graph search algorithms. We modified Neo4j so that the adjacency matrix is internally generated from corresponding Java classes if necessary and used for graph search using multicores and GPUs.

To reduce memory usage of the adjacency matrix and the number of accesses to the matrix, we implemented it as three linear arrays instead of a conventional two-dimensional adjacency matrix. Figure 1 illustrates the adjacency matrix structure implemented with three linear arrays extracted from a graph. The left side of Figure 1 shows an example of a graph, in which numbers in nodes and those near edges represent node-IDs and edge weights, respectively. The right side of Figure 1 shows the three linear arrays: Destination, Weight, and Pointer arrays. The lengths of Destination and Weight arrays are equivalent to the number of uni-directional links

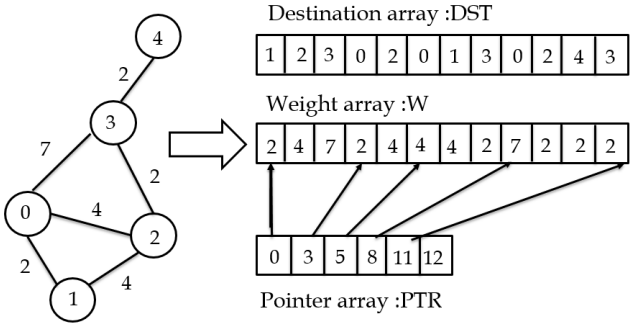


Figure 1: Adjacency matrix is implemented with three arrays

and that of Pointer array is equivalent to the number of nodes + 1.

- Each element in Pointer array $PTR[n]$ represents accumulated degree of Node-n, where $PTR[n] = \Sigma_{i=0}^{n} D_i$ and D_i represents degree of Node-i.
- Each element in Destination array $DST[p]$ shows a connected node from Node-n, where $PTR[n] \leq p < PTR[n+1]$.
- Each element in Weight array $W[p]$ shows the corresponding edge weight from Node-n to Node-$PTR[n]$.

Assume, for example, we want to know the edge weight from Node-3 to its second destination (Node-2). We can see that $PTR[3]$ is five from Pointer array. Thus, the edge weight from Node-3 to its second destination (Node-2) $W[PTR[3]+1]$ is four from Weight array.

In this work, the original graph search functions of Neo4j are hooked by our parallelized version that uses this data structure. The parallel graph search results are fed back to the original graph search functions and processed by Neo4j as well as normal execution. We have confirmed that our parallel graph search results are identical to those of the original Neo4j.

3.2 Parallelization with JOMP

One of the most common approach to parallelize the graph processing is to exploit multi-threaded execution, since most microprocessors recently equip more than four cores that share the same memory space on a single die or package. Since Neo4j is implemented with Java language, we employed JOMP [1] which is an OpenMP-like shared-memory multi-threaded library for Java platform. We parallelized Dijkstra's algorithm by using JOMP so that data access conflicts between multiple threads can be avoided.

In the case of A* algorithm, data access conflicts arise at list structures that maintain the computation target nodes and those that have already completed the computation. Although these two lists can be implemented with linked-lists to reduce memory usage, data access conflicts may occur when multiple threads add or remove nodes to/from the lists. To avoid such conflicts, we employed simple fixed-length arrays whose array length is equivalent to the number of nodes, in which value '1' means that the node exists in the list and '0' does not. Using these fixed-length structures, we parallelized A* algorithm so as not to add or remove the same node from multiple threads simultaneously. Although such a bitmap-like implementation consumes more memory compared to the linked-list implementation, it can omit the list traversal processing and shorten the data access time. Notice that their memory footprint is quite smaller than that of the adjacency matrix; thus the overhead is slight.

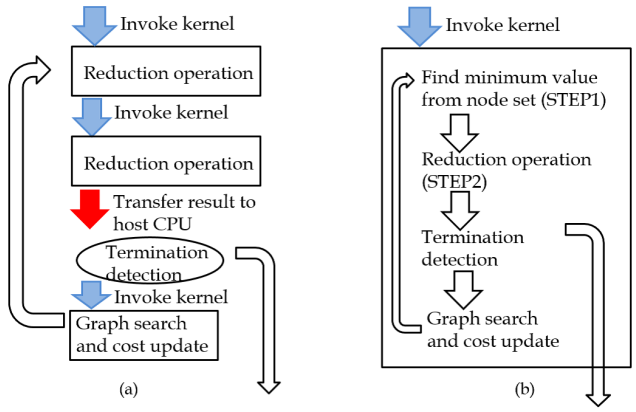


Figure 2: Parallelization with CUDA: (a) The number of threads is set to the number of nodes , (b) The number of threads is set to the maximum value of the thread block. (the number of nodes > the maximum value of the thread block.)

3.3 Parallelization with CUDA

To further boost the graph search processing of Neo4j we parallelized the graph search algorithms for CUDA-enabled GPUs. Since Neo4j is implemented with Java language, we employed jcuda in order to call CUDA from Java [3].

To fully exploit thousands of CUDA cores in a single GPU, we can typically generate much more threads than the number of actual cores. In this work, on the other hand, we employ a modest number of threads, i.e., 1,024 threads, which is the maximum number of threads in a single CUDA thread block, in order to avoid global synchronizations between thread blocks. This is because a global synchronization that introduces stop and restart of a kernel is costly when we use CUDA from Java. Thus we use a modest number of threads that fit into a single thread block.

As target graph search algorithms, we implemented Dijkstra and A* algorithms by using jcuda library for CUDA-enabled GPUs. They consist of the following three steps: 1) finding the minimum value among all the nodes to find a node explored in the next step, 2) termination detection of the algorithm, and 3) graph search and cost update of nodes. These algorithms repeat the three steps until they are terminated by the second step. Among these steps, the first step consumes the largest execution time.

Reduction operation is used for finding the minimum value among all the nodes. Typically, we can generate as many threads as the number of graph nodes, and reduction operations are performed recursively so that the minimum value can be found.

Figure 2(a) illustrates a flowchart for finding the minimum value when the number of threads is set to the number of graph nodes, while Figure 2(b) shows the case where the number of threads is limited to the number of a thread block. If the number of a target graph nodes is less than the number of a thread block, 2(a) and 2(b) is same behavior. In this paper, we don't treat these small graph.

Note that Figure 2(a) shows a simple case in which the result is obtained by only two reduction operations. In this figure, operations represented in boxes are performed by the GPU kernel, while those represented in circles are performed by the host CPU. In Figure 2(a), first, two reduction operations are performed by invoking the kernel to find the minimum value. Then the result is transferred to the host CPU in order to examine the loop termination, because this loop is executed and controlled by the host CPU. Until the loop is terminated, the same procedure (i.e., searching the graph, updating the cost, and finding the minimum value) is repeated.

In Figure 2(b), on the other hand, because a single reduction operation that processes more than 1,024 graph nodes cannot be implemented in single thread block, finding the minimum-valued node is divided into the following two steps.

- STEP1 in Figure 2(b): Each thread calculates the minimum-valued node from a node set that consists of node IDs of (n + multiples of 1024), where n is the thread ID.
- STEP2 in Figure 2(b): A reduction operation is performed to find the minimum value from the results collected by all the threads in the first step.

Figure 2(a) invokes three kernel calls and one data transfer to the host CPU for each loop iteration. Figure 2(b), on the other hand, calls the kernel only once at the beginning of whole the steps. Because of our Java-based implementation, the overheads of Figure 2(a)'s flow (i.e., calling the kernel and transferring data to the host CPU) are large; thus, Figure 2(b)'s flow that limits the number of threads to 1,024 but reduces the kernel invocation overheads is faster than Figure 2(a).

Although the problem is that it cannot utilize all the CUDA cores when the target GPU has more CUDA cores, such unused cores can be fully utilized by executing multiple kernels as different streams in Kepler architecture GPUs. Especially for graph database applications, we can see that multiple graph search queries will be performed for the same graph data. Thus, although a single kernel execution may not utilize all the CUDA cores, by executing multiple kernels in parallel, we can utilize all the cores and improve the throughput without degrading the latency.

4. EXPERIMENTAL RESULTS

4.1 Target Graphs

The modified Neo4j that utilizes multi-threaded library or GPU is evaluated with two types of graphs: 1) random graphs in which edges are generated between two randomly-selected nodes, and 2) a synthetic graph generated based on the degree distribution of a social networking service. They were generated as follows.

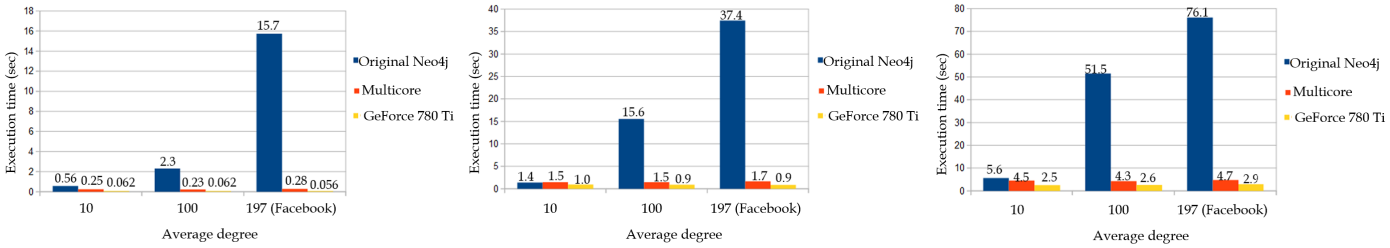
Random Graph. Given an average degree, a procedure that randomly selects a node pair and connects them via a bi-directional link is repeated until the given average degree is satisfied. Edge weight are also randomly generated. To analyze the relationship of these graph parameters and graph search performance, various-sized random graphs whose average degree is 10 and 100 are generated for the experiments.

SNS-Like Synthetic Graph. Reference [8] reports degree distribution of the Facebook social graph. We generate various-sized synthetic graphs so that their degree distribution becomes similar to that of the Facebook. The median and average of the degree were set to 99 and 197, respectively.

4.2 Experimental Environment

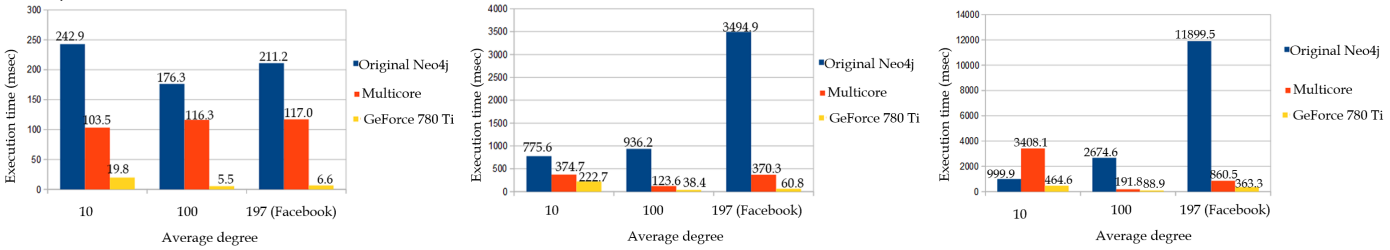
The original and our multi-threaded graph search of Neo4j are performed on AMD Opteron 4238 whose core frequency is 3.3GHz and the number of CPU cores is eight. As for the GPU-based graph search, we use NVIDIA Quadro K600 as a low-end GPU and NVIDIA GeForce GTX 780 Ti as a high-end GPU. Their parameters are listed in Table 1.

Graph databases can be used for social networking services. As practical graphs for these purposes, we generated the above-mentioned random and the synthetic Facebook graphs of 10,000, 50,000 and 100,000-node sizes. Note that our multi-threaded and GPU implementations of graph search algorithms require an auxiliary adjacency matrix extracted from the Neo4j original graph data structure and generating it introduces a computation overhead. In the following, we will show the ideal



(a)10,000 nodes (b)50,000 nodes (c)100,000 nodes

Figure 3: Execution time of Dijkstra algorithm with the original Neo4j, multi-threaded, and GPU (GeForce 780 Ti) implementations



(a)10,000 nodes (b)50,000 nodes (c)100,000 nodes

Figure 4: Execution time of A* algorithm with the original Neo4j, multi-threaded, and GPU (GeForce 780 Ti) implementations

Table 1: GPUs used in the experiments

	Quadro K600	GeForce GTX 780 Ti
Number of core	192	2,880
Core clock	875MHz	875MHz
Memory clock	900MHz	1,750MHz
Memory datapath width	128bit	384bit
Memory bandwidth	29GB/s	336GB/s

performance gains that do not take into account the overheads of the auxiliary matrix. Then we will analyze the performance gain with the overheads in Section 4.5.

4.3 A Single Execution

4.3.1 *Dijkstra Algorithm*

In general, Dijkstra algorithm finds the shortest paths from a single source node to all the destination nodes. However, since the graph search functions for graph databases are typically used to find the shortest path between a given pair of source and destination nodes, our multi-threaded and GPU implementations also find the shortest path between specified source and destination nodes. In the experiments, thus, each graph search query specifies the source and destination pairs selected randomly.

Figure 3 shows average execution time to perform a graph search of Dijkstra algorithm with the original Neo4j, our multi-threaded version, and GPU-based version. NVIDIA GeForce 780 Ti is used as a GPU. We can see that the execution time of the original Neo4j increases as the average degree increases. For example, the execution time of a graph search for the Facebook degree distribution graph is up to 28x larger than that for a random graph whose average degree is 10. On the other hand, the execution times of the multi-threaded and GPU versions are almost constant even when the average degree increases. This difference comes from the difference of the data structure of each implementation. The results show that our multi-threaded and GPU-based implementations are advantageous compared to the original one when the average node degree is quite large, such as Facebook social graph whose average degree is 197.

The multi-threaded version is faster than the original Neo4j implementation except for the the 50,000-node sparse graph whose degree is 10. Especially in the 100,000-node synthetic graph based on the Facebook degree distribution, the multi-threaded version is up to 16.2x faster than the original implementation. The GPU version that uses GeForce 780 Ti is always faster than the original implementation. In the 100,000-node synthetic graph based on the Facebook degree distribution, it is up to 26.2x faster than the original implementation.

4.3.2 *A* Algorithm*

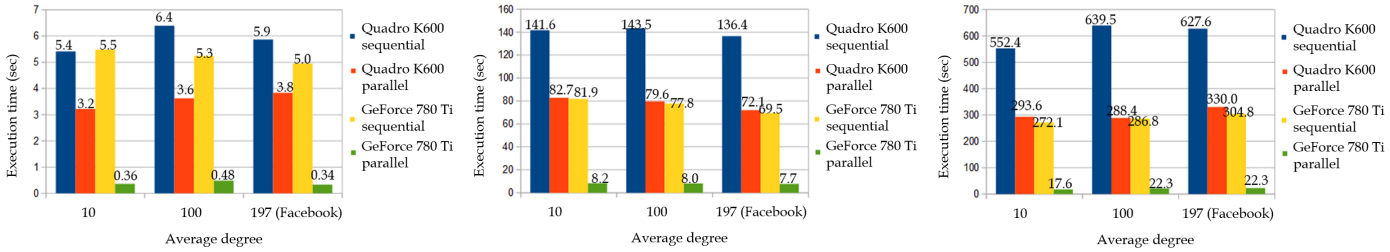
A* algorithm uses an estimated cost value from source to destination for each graph search in order to further improve the performance. Thus, such an estimated cost value is required for each graph search beforehand.

A simple but commonly-used way to estimate the cost is first assigning coordinates to every node and then calculating the distance between the source and destination coordinates. We use this simple but practical method in this experiment. Since the best way to find the estimated cost value is highly dependent on the target graph or target application, the cost estimation is performed by a single CPU (not using multi-threads nor GPUs) as well as the original implementation in this experiment.

As well as Dijkstra algorithm, here we measured the execution time to find the shortest path between a specified source and destination nodes pair using A* algorithm.

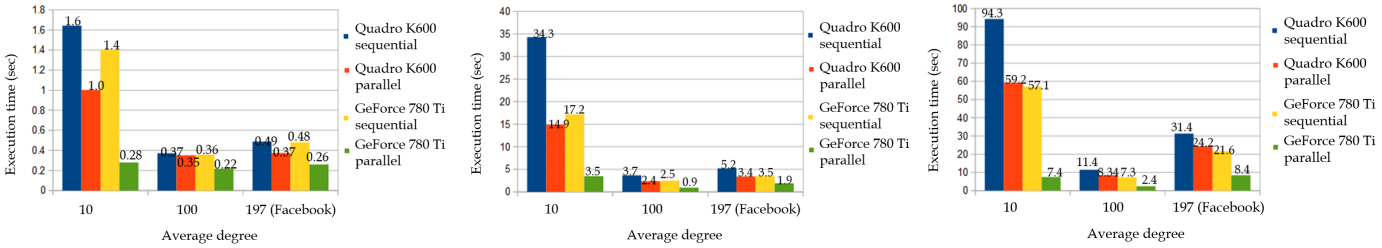
Figure 4 shows average execution time to perform a graph search of A* algorithm with the original Neo4j, our multi-threaded version, and GPU-based version. The results show a different tendency compared to those of Dijkstra algorithm. More specifically, the multi-threaded and GPU versions in Figures 4 (b) and (c) show the minimum execution time when the node degree is 100 compared to node degrees of 10 and 197 (i.e., Facebook graph).

Because A* algorithm preferentially searches the node that shows the smallest estimated cost to the destination, the possibility to find a closer node to the destination increases as the node degree increases. In the case of the synthetic graph based on the Facebook degree distribution, although its average degree is 197, the median of degree is 99 and 10% of the nodes have a quite small degree (i.e., ≤ 10). That is, the Facebook graph has more such small-degree nodes compared to the random graph whose average degree is 100. This is the reason why the multi-threaded and GPU versions sometimes show the minimum execution time when the node degree is



(a)10,000 nodes (b)50,000 nodes (c)100,000 nodes

Figure 5: 100 sequential and parallel execution times of Dijkstra algorithm with two GPUs



(a)10,000 nodes (b)50,000 nodes (c)100,000 nodes

Figure 6: 100 sequential and parallel execution times of A* algorithm with two GPUs

100 compared to node degrees of 10 and 197.

On the other hand, the execution time of the original Neo4j implementation increases as the node degree increases as shown in Figures 4 (b) and (c), except for the small 10,000-node graph in Figure 4 (a) due to the above-mentioned reason.

The multi-threaded version is faster than the original Neo4j implementation except for the the 100,000-node sparse graph whose degree is 10. Especially in the 100,000-node synthetic graph based on the Facebook degree distribution, the multi-threaded version is up to 13.8x faster than the original implementation. The GPU version that uses GeForce 780 Ti is always faster than the original implementation. In the 100,000-node synthetic graph based on the Facebook degree distribution, it is up to 32.8x faster than the original implementation.

4.4 Parallel Executions

In this section, we evaluated the executions time for performing 100 graph searches in parallel using our GPU-based implementation.

4.4.1 Dijkstra Algorithm

Here we compare the results on the low-end and high-end GPUs and analyze the parallel execution of the GPU kernel (100 graph searches using Dijkstra algorithm in parallel). Figure 5 shows the results of 100 sequential executions with Quadro K600 (low-end GPU), parallel executions with Quadro K600, 100 sequential executions with GeForce 780 Ti (high-end GPU), and parallel executions with GeForce 780 Ti.

In Kepler architecture, 192 cores are used as a single thread block, which means that whole cores in Quadro K600 are used for a single thread block. Nevertheless, the throughput can be improved with parallel execution of the kernel, because the kernel is executed in parallel when the number of used cores is less than 192 during reduction operation and memory access. In fact, in the case of Quadro K600, the parallel execution is 1.5x to 2.2x faster than the sequential executions.

As shown in the Table 1, Quadro K600 and GeForce 780 Ti are different in terms of the number of CUDA cores and memory clock frequency. Their CUDA cores are operated at the same clock frequency. Because the number of cores used for a single thread block is 192, the difference of their sequential execution times comes from the difference of their memory access speed. In the case of the 10,000-node random graph, the difference of their sequential execution times is small because the memory access speed is not dominant in terms of the execution time. In the cases of the 50,000 and 100,000-node graphs, the sequential execution time with GeForce 780 Ti is 1.7x to 2.2x faster than that of Quadro K600.

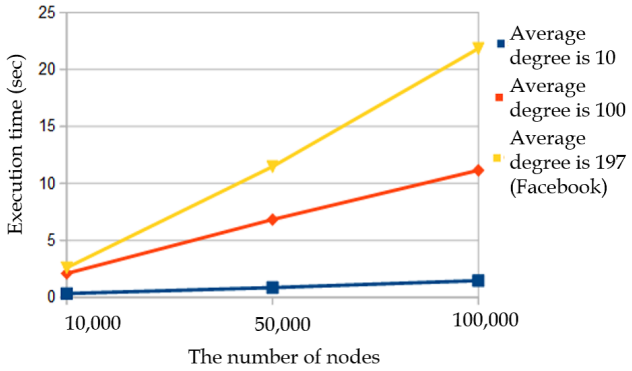


Figure 7: Overhead to generate auxiliary adjacency matrix

In the parallel executions with GeForce 780 Ti, because the number of cores used is larger than 192, the throughput is also increased significantly. As a result, the parallel execution performance is up to 15.4x faster than the sequential execution.

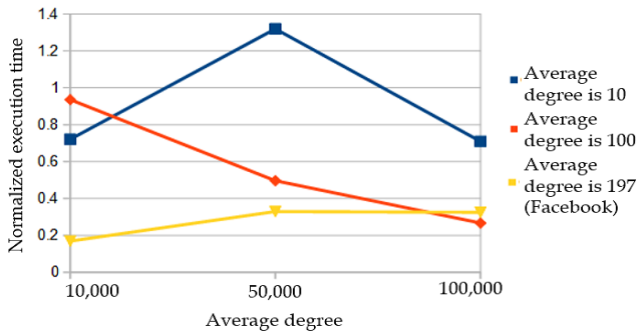
4.4.2 A Algorithm*

Here we measured execution times to perform A* algorithm 100 times sequentially or in parallel. Figure 6 shows the sequential execution time with Quadro K600 (low-end GPU), parallel execution with Quadro K600, sequential execution with GeForce 780 Ti (high-end GPU), and parallel execution with GeForce 780 Ti.

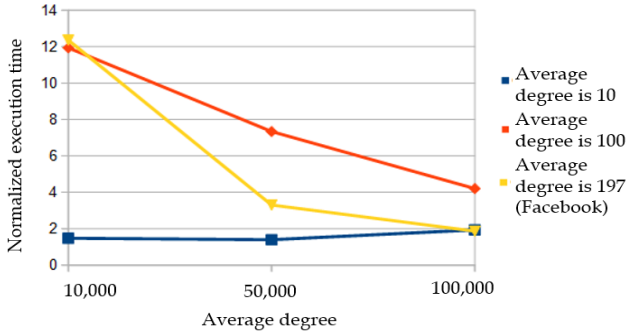
We can see that in all the cases, the execution time is large when average degree is 10 and it decreases when the average degree is 100, as well as the results in Figure 4. When we focus on the relative differences on the four cases, their execution times show the similar tendency as Dijkstra algorithm (Figure 5). Although the tendency is similar to that of Dijkstra algorithm, the speed-up for A* algorithm with GeForce 780 Ti is smaller than that for Dijkstra algorithm because of the overhead to calculate estimated cost value with host CPU.

4.5 Overhead

Although our multi-threaded and GPU-based implementation require the same auxiliary adjacency matrix extracted



(a)Dijkstra algorithm



(b)A* algorithm

Figure 8: Relative execution time that includes the auxiliary adjacency matrix generation and a single graph search on GeForce 780 Ti (the execution time of the original Neo4j implementation is normalized to 1.0)

from the Neo4j original graph data structure, so far we did not take into account the overhead to generate it. The overhead depends on the number of nodes and node degree. Assuming that we perform multiple graph search queries on a target graph, the auxiliary adjacency matrix is generated only at the first query and the same adjacency matrix is used for successive queries. Thus, our implementations are suitable for applications that perform a certain amount of queries for the same graph structure. Conversely, in the cases of applications that perform graph searches on "the latest" target graph whose structure changes very frequently, the adjacency matrix generation overhead may cancel the benefit of the multi-threaded and GPU executions.

Figure 7 shows overhead to generate the adjacency matrix for each graph. We can see that the overhead proportionally increases as the number of nodes and the node degree.

Figure 8 shows the relative execution time that includes the auxiliary adjacency matrix generation and a single graph search on GeForce 780 Ti. The execution time of the original Neo4j implementation is normalized to 1.0. Thus, when the relative execution time is greater than 1.0, a single graph search query using GPU cannot improve the performance compared to the original Neo4j implementation.

In the case of Dijkstra algorithm (Figure 8(a)), our GPU implementation is faster than the original implementation except for the 50,000-node random graph whose average degree is 10, even when the auxiliary adjacency matrix must be regenerated for each query. When multiple queries are performed for the same graph structure, our GPU-based implementation becomes more advantageous.

In the case of A* algorithm (Figure 8(b)), because its execution time is shorter than that of Dijkstra algorithm, the impact of the overhead is bigger. As a result, our GPU-based implementation is slower than the original Neo4j implementation. In other words, our GPU-based implementation can improve the performance when multiple search queries can be performed for the same auxiliary adjacency matrix.

Note that the overhead discussed in this section is the calculation time to newly generate the auxiliary adjacency matrix. On the other hand, in typical use cases of graph databases, graph update queries change only a part of the whole graph (e.g., some specific nodes, edges, and properties) and do not change the whole graph structure drastically. In this case, we can modify only a part of the auxiliary adjacency matrix instead of newly regenerating the matrix. Since the same auxiliary adjacency matrix can be used for a long period, we can drastically reduce the overhead of generating the adjacency matrix. Consequently, we believe that the speed-up obtained by our implementations is significant, as mentioned in Sections 4.3 and 4.4.

5. SUMMARY

In this paper, we designed and implemented graph search functions (Dijkstra and A* algorithms) with a multi-threaded library for multi-cores and CUDA for GPUs. Although graph processings using GPUs have been well researched so far, in this work we integrated our multi-threaded and GPU-based graph search functions into a practical graph database. Our multi-threaded and GPU-based graph functions require an auxiliary adjacency matrix for a target graph. We evaluated them in term of graph database search performance on up to 100,000-node graphs generated based on a constant node degree and a degree distribution of Facebook social graph.

The results show that, when we do not take into account additional overhead to generate the auxiliary adjacency matrix, multi-threaded version improves a single search query that uses Dijkstra and A* algorithms by 16.2x and 13.8x compared to the original implementation. The GPU-based implementation improves a single search query by 26.2x and 32.8x, and the performance gain further increases as the number of search queries performed increases. When we take into account the overhead, although the speed-ups by our implementations are reduced, by reusing the auxiliary adjacency matrix for multiple graph search queries we can significantly improve the graph search performance.

6. REFERENCES

[1] J. M. Bull and M. E. Kambites. JOMP - An OpenMP-like Interface for Java. In *Proc. of International Conference on Java Grande*, pages 44–53, June 2000.

[2] T. H. Hetherington, T. G. Rogers, L. Hsu, M. O'Connor, and T. M. Aamodt. Characterizing and Evaluating a Key-value Store Application on Heterogegenenous CPU-GPU Systems. In *Proc. of the International Symposium on Performance Analysis of System and Software*, pages 88–98, April 2012.

[3] jcuda.org. `http://www.jcuda.org`.

[4] D. Merill, M. Garland, and A. Grimshaw. Scalable GPU Graph Traversal. In *Proc. of International Symposium on Principles and Practice of Parallel Programming*, pages 117–128, August 2012.

[5] Neo4j.org. `http://www.neo4j.org`.

[6] S. Nobari, T.-T. Cao, S. Bressan, and P. Karras. Scalable Parallel Minimum Spanning Forest Computation. In *Proc. of International Symposium on Principles and Practice of Parallel Programming*, pages 205–214, August 2012.

[7] H. Ortega-Arranz, Y. Torres, D. R. Llanos, and A. Gonzalez-Escribano. A New GPU-based Approach to the Shortest Path Problem. In *Proc. of International Conference on High Performance Computing and Simulation*, pages 505–511, July 2013.

[8] J. Ugander, B. Karrer, L. BackStrom, and C. Marlow. The Anatomy of the Facebook Social Graph. In *Arxiv preprint arXiv:1111.4503*, November 2011.