Distributed In-GPU Data Cache for Document-Oriented Data Store via PCIe over 10Gbit Ethernet

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

[1] Keio University, 3-14-1 Hiyoshi, Kohoku-ku, Yokohama, Japan 223-8522
{morisima,matutani}@arc.ics.keio.ac.jp
[2] National Institute of Informatics
[3] Japan Science and Technology Agency PRESTO

Abstract. As one of NOSQL data stores, a document-oriented data store manages data as documents in a scheme-less manner. Various string match queries, such as a perfect match, begins-with (prefix) match, partial match, and regular expression based match, are performed for the documents. To accelerate such string match queries, we propose DistGPU Cache (Distributed In-GPU Data Cache), in which data store server and GPU devices are connected via a PCI-Express (PCIe) over 10Gbit Ethernet (10GbE), so that GPU devices that store and search documents can be added and removed dynamically. We also propose a partitioning method that distributes ranges of cached documents to GPU devices based on a hash function. The distributed cache over GPU devices can be dynamically divided and merged when the GPU devices are added and removed, respectively. We evaluate the proposed DistGPU Cache in terms of regular expression match query throughput with up to three NVIDIA GeForce GTX 980 devices connected to a host via PCIe over 10GbE. We demonstrate that the communication overhead of remote GPU devices is small and can be compensated by a great flexibility to add more GPU devices via a network. We also show that DistGPU Cache with the remote GPU devices significantly outperforms the original data store.

1 Introduction

Recent advances on Social Networking Services, Internet of Things technologies, mobile devices, and sensing technologies are continuously generating large data sets, and a simple, scalable, and high-throughput data store is a key component for managing such big data. Structured storage or NOSQL is an attractive option for storing large data sets in addition to traditional RDBMS. NOSQL data stores typically employ a simple data structure optimized for high horizontal scalability on some selected application domains via sharding and replication over number of machines. Due to these features, NOSQL data stores are increasing their presence in Web applications and cloud-based systems that store and manage big data.

Document-oriented data store [1][2] is one of major classes of NOSQL. In a document-oriented data store, data are typically stored as documents in a JSON-like binary format without any predefined data structure or schema; thus it is referred to as a scheme-less data store. In the document-oriented data store, a search query retrieves documents whose values are matched to a search condition given by the search query. Especially, string search queries, such as perfect, begins-with (prefix), and regular expression based string match queries, are used in document-oriented data stores. Since the computation cost for string search increases as the number of

documents increases, it becomes quite high when dealing with large data sets. To mitigate the computation cost increase, database indexes are typically employed in document-oriented data stores [7], in order to reduce the cost from O(n) to O(log$n$), where n is the number of documents. Database indexing is a powerful technique especially for perfect and prefix string match queries. However, it cannot be directly applied to some important string search queries, such as partial and regular expression based string match queries, although database indexes for regular expression based search have been studied. A motivation of this paper is to accelerate search queries of the document-oriented data store without relying on database indexes.

To accelerate all the string search queries including the regular expression based search in document-oriented data stores, an array-based cache suitable for GPU processing of string search queries was proposed in [5]. The cache is extracted from the original document-oriented data store. When the document-oriented data store receives a string search query, the query is performed by a GPU device using the cache and the search result is returned to the original document-oriented data store. However, a serious problem of the cache is that, since it targets only a single GPU device, it is inefficient for data larger than a device memory capacity of a single GPU device. Thus, a horizontal scalability to add more GPU devices and increase the device memory capacity is required to manage a larger data set. In addition, since the number of documents in the data store increases and decreases dynamically, a flexibility to add and remove GPU devices dynamically is required.

To address these requirements, in this paper we propose DistGPU Cache (Distributed In-GPU Data Cache), in which a data store server (i.e., host) and GPU devices are connected via a PCI-Express (PCIe) over 10Gbit Ethernet (10GbE) technology [9], so that GPU devices that store and search data can be added and removed dynamically. We also propose a documents partitioning method that distributes ranges of cached data to GPU devices based on a hash function. DistGPU Caches can be dynamically divided and merged when the GPU devices are added and removed, respectively. An inherent concern on DistGPU Cache may be communication latency between the host and remote GPU devices connected via 10GbE. However, since in our proposal, DistGPU Caches reside in GPUs' device memory, they are not transferred to the GPU devices for every search query. Thus, communication overhead can be mitigated even with remote GPU devices. The contributions of this paper are summarized as follows.

- We propose DistGPU Cache that distributes database cache over GPU devices connected via a PCIe over 10GbE technology.
- We propose a documents partitioning method so that GPU devices that form DistGPU Cache can be added and removed dynamically.
- We evaluate document-oriented data store performance when varying the number of remote GPU devices.

The rest of paper is organized follows. Section 2 surveys related work. Section 3 proposes DistGPU Cache and the cache partitioning method for multiple remote GPU devices. Section 4 evaluates DistGPU Cache and compares it with that with local GPU devices and the original document-oriented data store using database indexes. Section 5 concludes this paper.

2 Background and Related Work

2.1 GPU-Based Regular Expression Matching

There are two approaches to implement regular expression matching: NFA (Nondeterministic Finite Automaton) based approach and DFA (Deterministic Finite

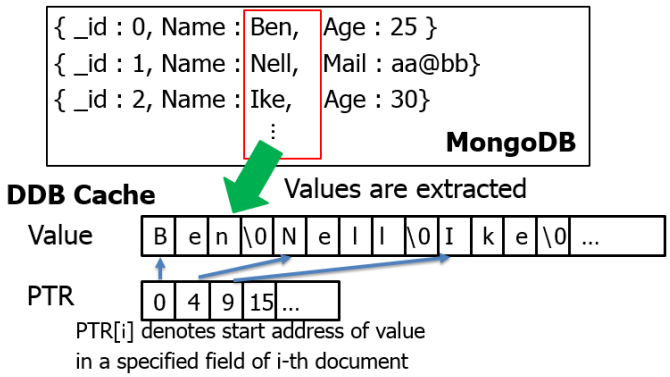

Fig. 1. DDB Cache creation.

Automaton) based approach. To accelerate the regular expression matching, GPU devices are used for DFA-based approach in [10] and NFA-based approach in [11]. Both the approaches have pros and cons. NFA-based approach is advantageous in terms of memory efficiency, while DFA-based approach is faster than NFA-based one. For a small rule-set regular expression matching, a DFA-based approach can be accelerated by using tokens of words in [6]. In addition, a text matching based on KMP algorithm is studied for database applications in [8]. In this paper, we implemented a DFA-based regular expression matching for GPU devices based on the design of [10].

2.2 GPU-Based Document-Oriented Data Store

To accelerate search queries of document-oriented data store, DDB Cache suitable for GPU processing was proposed in [5]. A similar idea was applied for a graph database in [4]. DDB Cache is an array-based data cache extracted from the document-oriented data store, and GPU-based string processing (e.g., regular expression based string matching) is performed for DDB Cached. Figure 1 shows a creation of DDB Cache from document-oriented data store. The upper half illustrates a simplified data structure of the document-oriented data store that includes multiple documents. The lower half illustrates its DDB Cache structure. DDB Cache is created for each field of documents. The same field name and value type are used for every document. Thus, only values are extracted from documents for each field and cached in a one-dimensional array structure as DDB Cache. Since the length of each value (e.g., string data) differs, an auxiliary array (PTR) is additionally used in DDB Cache to point the start address of field value of a specified document.

In DDB Cache, value and auxiliary arrays are created for each field. Although Figure 1 illustrates DDB Cache of only a single field, we can add DDB Cache for the other fields. When a search query is performed, one or more pairs of value and auxiliary arrays, which are corresponding to the field(s) specified in the query, are transferred to GPU device memory and then a string search is performed by the GPU device. Although a regular expression based string match query was accelerated by a single GPU device with DDB Cache in [5], a horizontal scalability to add or remove GPU devices dynamically was not considered though the horizontal scalability is one of the most important properties of NOSQL.

To address the horizontal scalability issue, in this paper we propose DistGPU Cache, in which a host and GPU devices are connected via a PCIe over 10GbE. We employ NEC ExpEther [9] as a PCIe over 10GbE technology. Using ExpEther,

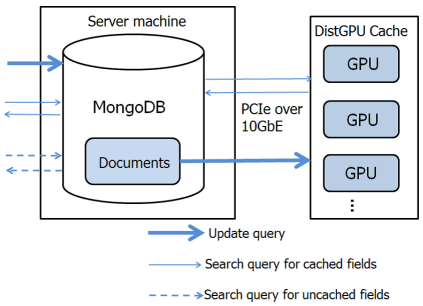

Fig. 2. Overview of DistGPU Cache.

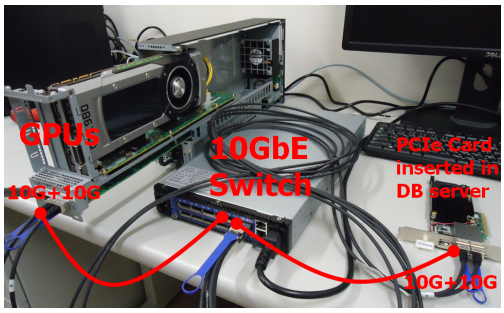

Fig. 3. Photo of remote GPU devices connected via 10GbE.

PCIe packets for hardware resources (e.g., GPU devices) are transported in 10GbE by encapsulating the packets into an Ethernet frame. Please note that there are software services based on client-server model that provides GPU computation to clients [3], while we employ a PCIe over 10GbE technology for connecting many GPU devices directly. In our case, pooled GPU devices can be connected to the data store server machine via 10GbE when necessary. Thus, our proposed DistGPU Cache is well suited to recent trends on rack-scale architecture and software-defined infrastructure.

3 DistGPU Cache and Its Partition Method

3.1 System Overview

DistGPU Cache is a distributed database cache stored in many remote GPU devices. DistGPU Cache consists of certain-sized buckets, each of which is processed by a GPU device. The detail about the buckets is described in the following subsections.

Figure 2 shows an overview of the proposed system. It consists of two components: 1) document-oriented data store and 2) DistGPU Cache distributed over remote GPU devices accessed via 10GbE. We use MongoDB [2] as a document-oriented data store in this paper and value fields of documents are cached in remote GPU devices as DistGPU Cache. Figure 3 shows remote GPU devices connected via a 10GbE switch for DistGPU Cache. Remote GPU device is connected to PCIe card via two 10GbE cables (i.e., 20Gbps) and the PCIe card is mounted in the host machine where MongoDB and DistGPU Cache are working.

The following steps are performed for each query in the proposed system.

- For UPDATE query, new data are written to the original document-oriented data store. Cached data in GPU device memory (i.e., DistGPU Cache) are updated if necessary.
- For SEARCH query, if the target fields have been cached in DistGPU Cache, the query is transferred to a corresponding GPU device to perform the document search. The search result is returned to the client via the document-oriented store.
- For SEARCH query, if the target fields have not been cached, the query is performed by the document-oriented store and the result is returned to the client.

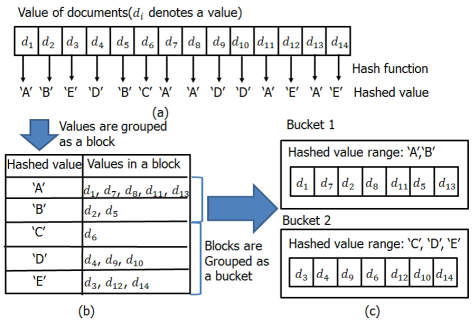

Fig. 4. Relationship between blocks and buckets using the hash function.

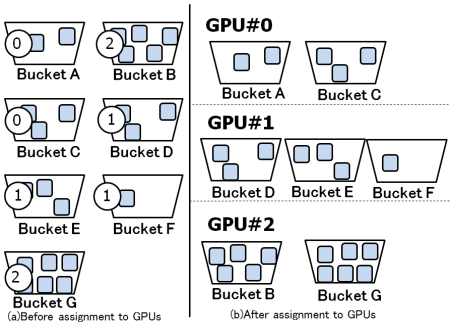

Fig. 5. Assignment of buckets to GPU devices.

3.2 Partitioning of Documents Values with Hash Function

Since DistGPU Cache is built by extracting values of a specific field of the documents, values in the DistGPU Cache are independent of each other. Thus, the set of values in DistGPU Cache can be partitioned and stored into GPUs in response to the number of GPU devices and their device memory capacity, in order to perform a search query in parallel. For example, assuming two GPU devices, the first half of the documents is stored in a GPU and the latter half is stored in another GPU. However, such a simple document partitioning is inefficient, e.g., write operations are concentrated on a single partition that contains the latest documents.

In this paper, we propose an efficient partitioning method that distributes document values to multiple GPU devices by using a hash function. More specifically, by the hash function, document values are distributed into small blocks and they are distributed to GPU devices evenly so as to equalize their workload and reduce the reconstruction overhead.

Using the proposed partitioning method, we can utilize the hash value as an index to narrow down a search space and reduce the computation cost.

Typically, a collision resistance is required for hash functions. On the other hand, we introduce a coarse-grain hash function that generates the same hashed value for a range of consecutive values. Here we define "block" as a group of values with the same hashed value. All the values in the same block are stored into the same GPU device for search. Thus we can use such a hashed value instead of a database index for search in order to narrow down the search range.

However, a block is not suitable to be used as a bucket directly, because the number of values in each block (partitioned by the coarse-grain hash function) may differ and the number of values stored in each bucket should be balanced in order to distribute the workload of each bucket and thus improve the performance. Instead of a single block, multiple blocks (with different sizes) are grouped as a "bucket" so that sizes of buckets should be balanced. We also define "hashed value range" as a set of hashed values of blocks grouped in the same bucket. One or more buckets are assigned to a GPU device (the assignment is discussed in Section 3.4). This approach tolerates collisions of hashed values. It also tolerates unbalanced sizes of blocks (and thus non-uniform distribution of hashed values). Thus, a simple hash function with a low computation overhead can be used. For example, in our implementation, the first n characters of value strings are used as hashed values. By varying n, the sizes of blocks can be controlled.

Figure 4 shows relationship between blocks and buckets using the hash function. We assume that values d_1 to d_{14} are hashed and then hashed values 'A' to 'E'

are generated for simplicity. Figure 4(a) shows the values in documents and their corresponding hashed values. As shown, multiple values that have the same hashed value are grouped as a block. Figure 4(b) shows the hashed values and their corresponding blocks. Since sizes of blocks differ, these blocks are packed into buckets so that the number of values in each bucket should be balanced, as shown in Figure 4(c). In this example, blocks that have hashed values A or B are grouped as bucket 1 and those have hashed values C, D, or E are grouped as bucket 2. The sizes of buckets 1 and 2 are balanced. Blocks in a bucket are independent with each other (i.e., blocks with different hashed values coexist in a bucket). When we add a new value to DistGPU Cache, a hashed value of the new value is computed and then the new value is stored into a bucket that covers this hashed value. Although the sizes of buckets are currently balanced in Figure 4, the number of values in each bucket will change dynamically due to write queries and thus their sizes will be unbalanced as time goes on. To handle such dynamic growth of buckets, we need to update the hashed value range of each bucket dynamically. In our design, the maximum number of values (or the maximum total sizes of values) in each bucket is predefined and if the number of values in a bucket exceeds the maximum number, the bucket is divided into two buckets.

Algorithm 1 shows a pseudo code of the proposed bucket partitioning method, assuming bucket A is divided into buckets A and B. If a bucket covers only a single hashed value, the bucket cannot be partitioned (Line 6-8). In this case, a finer hash function should be used instead. For example, when the hash function uses the first n characters as a hashed value, we can increase n. The hash function should be selected so that the number of values in each bucket does not exceed the maximum number. In Line 9-11, a half of hashed value range of bucket A is moved to bucket B. In Line 12-13, the number of values in each bucket is recomputed based on the new hashed value range. Then values in bucket A are moved to bucket B based on the new hashed value range.

Building a new DistGPU Cache or reconstructing an existing DistGPU Cache is equivalent to newly-adding whole documents to an empty bucket. In other words, first, H_A is set to all the hashed values and x_A is set to 0, then Algorithm 1 is repeated until buckets are partitioned so that their number of values does not exceed the maximum value.

Algorithm 1 Bucket partitioning

1: $A \leftarrow$ Original bucket to be partitioned
2: $B \leftarrow$ New bucket to be diverged
3: $H_A, H_B \leftarrow$ Hashed value ranges for A and B
4: $h_A \leftarrow$ Actual hashed values included in A ($h_A \in H_A$)
5: $x_A, x_B \leftarrow$ Numbers of hashed values in A and B
6: **if** $x_A = 1$ **then**
7: Terminate //Bucket A cannot be partitioned any more
8: **end if**
9: **for** $i = 1$ to $\lfloor x_A/2 \rfloor$ **do**
10: Move largest hashed value in h_A to H_B and delete it from H_A
11: **end for**
12: $x_A \leftarrow \lceil x_A/2 \rceil$ //Number of hashed values of H_A after partitioning
13: $x_B \leftarrow \lfloor x_A/2 \rfloor$ //Number of hashed values of H_B after partitioning
14: Values are moved from A to B based on new H_A

3.3 Toward Schema-Less Data Structure

In a document-oriented data store, each document may have different fields. For example, a document has fields A and B, while another document may have only field C. DistGPU Cache is required to support such a scheme-less data structure.

In MongoDB, all the documents must have _id field as a primary key. DistGPU Cache of _id field is used as a primary key to refer to those of the other fields. To do this, DistGPU Cache of _id field needs two additional data for each field: 1) bucket ID and 2) address inside the bucket where the field value is stored. Thus, DistGPU Cache of _id field has $2 \times N$ additional arrays, where N is the number of fields, to record the bucket ID and address inside the bucket where a corresponding field value is stored.

DistGPU Cache of the other fields has an additional array, in order to record the bucket ID and address in the bucket where a corresponding _id is stored. In other words, these additional arrays record the relationship between _id field and the other fields in the same document. When a field value in a document is accessed, another field value of the same document can be accessed by using these additional arrays. Please note that DistGPU Cache may not cache all the fields used in the documents. When a field value not cached in DistGPU Cache is accessed, MongoDB is invoked by specifying _id in order to retrieve all the field values.

3.4 Assignment of Buckets to GPU Devices

To store buckets in GPU device memory as DistGPU Cache, we need to take into account which buckets are stored to which GPU devices. In our design, each bucket has a random integer number which is less than the number of GPU devices (e.g., each bucket has 0, 1, or 2 as a random integer number when the number of GPU devices is three). The buckets are assigned to GPU devices based on their random integer numbers. If the number of buckets is huge, such a random assignment of buckets to GPU devices can balance the workload of GPU devices. When a new bucket is added, assignments of the other buckets to GPU devices are not changed and only the new bucket is newly assigned to a GPU device; thus the overhead to add new buckets is low.

Figure 5 shows an assignment of buckets to GPU devices. Seven buckets (Figure 5(a)) are assigned to three GPU devices, as shown in Figure 5(b). Once a bucket is assigned to GPU device, it resides in the same GPU device until DistGPU Cache is reconstructed.

In DistGPU Cache, the number of GPU devices changes dynamically and the bucket assignment also takes into account the number of GPU devices available. When a new GPU device is added, b/G buckets in existing GPU devices are randomly selected and moved to the new GPU device, where b is the number of total buckets and G is the number of total GPU devices. When an existing GPU device is removed, a new random integer number (not the current number) is generated for each bucket in the GPU device. Then buckets in the GPU device to be removed are moved to the other GPU devices based on their random integer number. When a GPU device is added or removed, a range of random integer numbers for new buckets is updated.

3.5 GPU Processing for DistGPU Cache

We use NVIDIA GPU devices and CUDA (Compute Unified Device Architecture) C development environment to implement GPU kernels.

Since in DistGPU Cache, documents are grouped as buckets and stored into GPU devices, document search is performed in bucket basis. In our design, values

that generate the same hashed value are grouped as a block in a bucket; thus some search queries may scan only a limited bucket or GPU device memory. For example, a perfect or prefix search query for string values scans only a bucket or GPU device memory. On the other hand, regular expression search without any prefix cannot limit the search space and thus scans all the DistGPU Cache.

We implemented a DFA-based regular expression search kernel similar to [10] using CUDA. When a search space is limited to a single block, the CUDA kernel is executed only once. Otherwise, the CUDA kernel is executed for each bucket in the search space. In this case, since buckets are independent with each other, the CUDA kernels for different buckets are executed in asynchronous manner. We can thus hide the CPU-GPU data transfer overhead since the data transfer to/from GPU devices and computation in GPU devices can be overlapped.

4 Evaluations

4.1 Evaluation Environment

MongoDB and our DistGPU Cache are operated in the same machine. CPU of the host machine is Intel Xeon E5-2637v3 running at 3.5GHz and memory capacity is 128GB. Up to three NVIDIA GeForce GTX980 GPUs, each of which has a 4GB device memory, are used for DistGPU Cache. We use MongoDB version 2.6.6 and CUDA version 6.0. For DistGPU Cache, the experiment system shown in Figure 3 except that remote GPU devices are directly connected to the host without L2 switch for simplicity. For comparison, we evaluate the performance when the GPU devices are directly attached to the host machine via PCIe Gen3 x16.

4.2 Performance with Different Bucket Sizes

Here we evaluate the performance of DistGPU Cache when the bucket size varies. We measured the throughputs of a perfect string match query that scans only a single bucket and a regular expression based string match query that scans all the buckets in DistGPU Cache. In addition, we measured the query execution time when the number of GPU devices varies in order to evaluate the dynamic join/leave of GPU devices vs. the bucket sizes. The number of documents in our experiments is ten million.

For the perfect string match query, ten million documents each of which has two fields, _id field and a randomly-generated 8-character string field, are generated and the perfect string match query is performed for the string field. The regular expression based string match query is also performed for the above-mentioned ten million documents.

Figure 6 shows the perfect string match query performance of DistGPU Cache when the number of GPU devices is varied from one to three and the maximum bucket size is varied from 1×2^{10} to 512×2^{10}. The throughput is represented as rps (request per second). Since a perfect string match query scans only a single bucket, the size of search space is proportional to the bucket size. Please note that when the bucket size is smaller than a certain threshold, the GPU parallelism cannot be fully utilized and thus the throughput becomes constant. As shown, when the maximum bucket size is larger than 128×2^{10} or 256×2^{10}, the throughput decreases.

Figure 7 shows the regular expression based string match query performance of DistGPU Cache. Since a regular expression based string match query scans all the buckets, the search space is constant regardless of bucket size. When the bucket size is small, since more CUDA kernels for smaller buckets are executed, the number of CUDA kernel invocations and the data transfer overhead between host and GPU

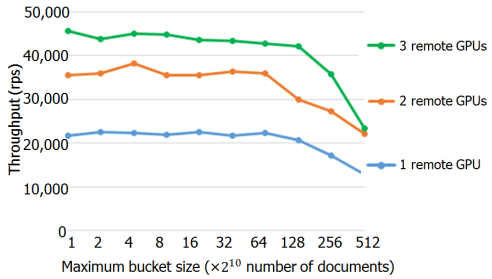

Fig. 6. Perfect string match query performance vs. bucket sizes.

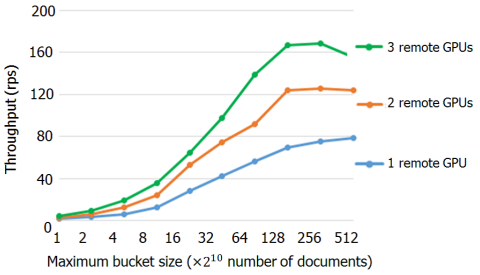

Fig. 7. Regular expression based string match query performance vs. bucket sizes.

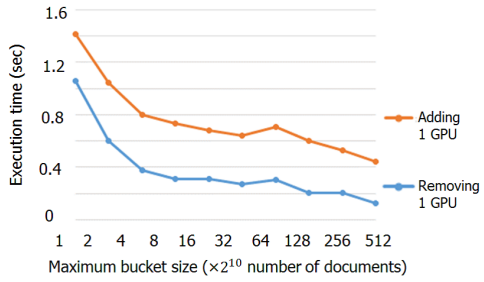

Fig. 8. Execution time when GPU devices are added or removed dynamically.

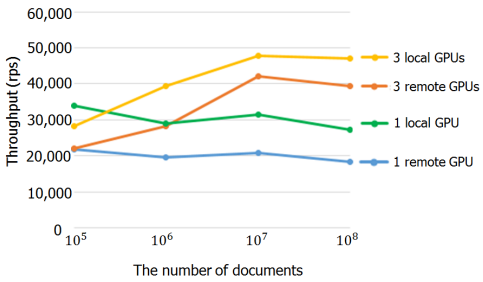

Fig. 9. Perfect string match performance when GPU devices are local and remote.

devices are increased, resulting in a lower throughput. As shown, the throughput is increased until the maximum bucket size is enlarged to 128×2^{10}. The throughput is significantly decreased when the maximum bucket size is 512×2^{10} especially when the number of GPU devices is three. This is because, as the maximum bucket size is enlarged, the number of buckets is decreased, the workload cannot be divided into the three GPU devices evenly.

Figure 8 shows the execution times when the GPU devices are dynamically added or removed. The number of GPU devices are increased from one to two and decreased from two to one. In both the cases, as the maximum bucket size is enlarged, the execution time is decreased. This is because, as the bucket size is enlarged, the number of buckets is decreased and the number of memory allocations and data transfer between host and GPU devices are decreased. When we compare both the cases (i.e., adding and removing GPU), the execution time of the latter case is shorter than that of the first case. This is because, when the GPU device is added, a device memory is allocated in the new GPU device and then a part of existing data are transferred to the new GPU device.

In summary, the performance is not degraded in both the perfect and regular expression based string match queries when the maximum bucket size is 128×2^{10}. Since this bucket size is proper in this evaluation environment, we use this parameter in the following experiments.

4.3 Performance with Local and Remote GPUs

In DistGPU Cache, we assume that GPU devices are connected to the host machine via 10GbE (in our design, two STP+ cables are used for each GPU device, resulting in 20Gbps). Of course it is possible to directly mount the GPU devices to the host machine via PCIe Gen3 x16, but the number of such local GPU devices mounted

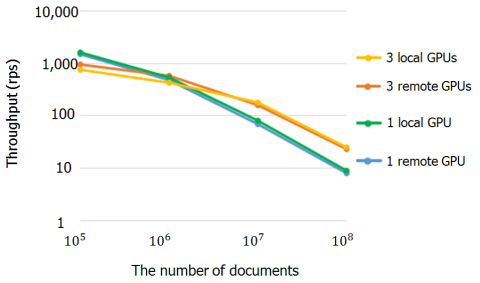

Fig. 10. Regular expression based string match performance when GPU devices are local and remote.

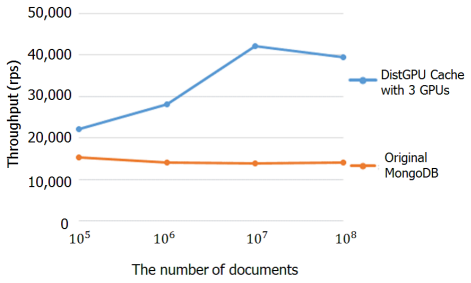

Fig. 11. Perfect string match performance of DistGPU Cache and original MongoDB.

will be limited by the motherboard or chassis. Here we measured the performance when the GPU devices are directly attached to the host machine via PCIe Gen3 x16, in order to show the performance overhead due to the "remote" GPU devices.

The perfect string match and regular expression based string match queries are performed for local and remote GPU devices. Although these queries and documents are the same as those in Section 4.2, the number of documents are varied from one hundred thousand to one hundred million.

Figure 9 shows the perfect string match query throughputs for local and remote GPU devices when the number of GPU devices is one and three. When the number of documents is quite small, the number of buckets is also small and buckets cannot be distributed to GPU devices evenly; thus the throughput of 3GPU case is decreased when the number of documents is small. The throughput of the local GPU case is always better than that of remote GPU case. However, in the remote GPU case, the throughput increases in a higher rate compared to the local GPU case. Actually, when the number of documents is one hundred million, the throughput improvement from 1GPU to 3GPU is 2.14x for the remote GPU case, while it is only 1.73x for the local GPU case. The local GPU performance is better than the remote GPU case by 1.20x when the number of documents is one hundred million and the number of GPU devices is three; thus performance degradation of remote GPU case is not significant by taking into account the scalability benefits.

Figure 10 shows the regular expression based string match query throughputs for local and remote GPU devices. In the graph, the throughput (Y-axis) is represented as a logarithmic scale. As the number of documents is increased, the computation cost is proportionally increased and thus the throughput is degraded. However, the performance degradation is relatively slow, since the CUDA kernels are executed in parallel. The local GPU performance is better than the remote GPU case by only 1.08x when the number of documents is one hundred million and the number of GPU devices is three; thus the performance degradation of the remote GPU case is quite small.

Regarding the latency, the execution times to deal with the perfect matching query are 0.30 msec and 0.22 msec for local and remote GPU cases respectively, when the numbers of documents and GPU devices are one hundred million and three respectively. Their latencies are almost constant regardless of the number of GPU devices because we can narrow down the search space only to a single bucket stored in a single GPU device. Those of the regular expression matching query are 44.0 msec and 40.9 msec for local and remote GPU cases respectively, and the latencies are decreased as the number of GPU devices increases.

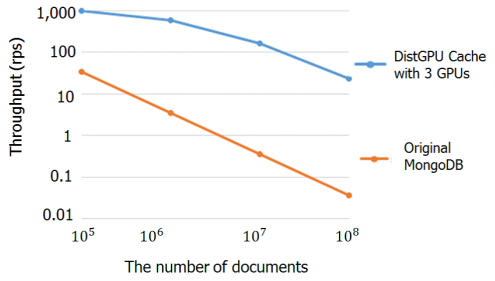

Fig. 12. Regular expression based string match performance of DistGPU Cache and original MongoDB.

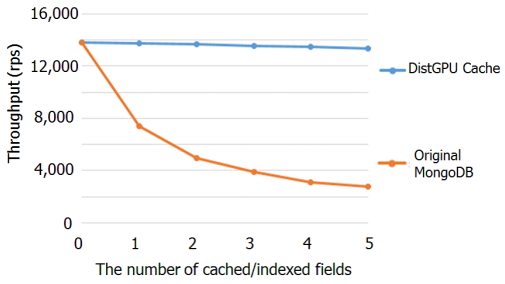

Fig. 13. Write query performance of DistGPU Cache and original MongoDB.

4.4 Performance Comparison with MongoDB

Here we compare the proposed DistGPU Cache using three remote GPU devices with the original MongoDB in terms of throughput using the same queries and documents as in Section 4.3. For MongoDB, B+tree index is used to improve the search performance of the perfect string match query, while any index is not used for the regular expression based search query since a simple indexing cannot be used for the regular expression based search query. In addition to the search query performance, the write throughput is measured in both the cases: DistGPU Cache and MongoDB with indexes. MongoDB is operated on a memory file system (i.e., tmpfs) for fair comparisons.

Figure 11 shows the perfect string match query throughputs of DistGPU Cache and the original MongoDB. Comparison between DistGPU Cache and MongoDB shows that the DistGPU Cache outperforms MongoDB even if the number of documents is small. When the number of documents is one hundred million, DistGPU Cache outperforms MongoDB by 2.79x.

Figure 12 shows the regular expression based string match query throughputs of DistGPU Cache and the original MongoDB. In the case of DistGPU Cache, the throughput degradation is suppressed especially when the number of documents is large. As a result, when the number of documents is one hundred million, DistGPU Cache outperforms the original MongoDB by 640.8x.

Regarding the latency, the execution times to deal with the perfect matching query are 0.30 msec and 0.071 msec for the DistGPU Cache and the original MongoDB cases respectively. On the other hand, those of the regular expression matching query are 44.0 msec and 28198.8 msec for the DistGPU Cache and the original MongoDB cases respectively as the regular expression matching query is quite compute intensive.

Figure 13 shows write query throughputs of DistGPU Cache and the original MongoDB. B+tree database indexes are used in the original MongoDB case. In this experiment, write queries that add new documents are performed on both the data stores (i.e., DistGPU Cache and the original MongoDB) where ten million documents have been already stored. Here, each document has _id field and five string fields filled with randomly-generated eight characters. As shown in Figure 13, the write throughput of the original MongoDB is degraded as the number of indexed fields increases, while the write throughput of DistGPU Cache is almost constant even when the number of cached fields in DistGPU Cache increases.

5 Summary

In this paper, we proposed DistGPU Cache, in which a host and GPU devices are connected via PCIe over 10GbE so that GPU devices that cache and process a document-oriented data store can be added and removed dynamically. We also proposed a bucket partitioning method that distributes ranges of documents to GPU devices based on a hash function. The buckets of DistGPU Cache can be dynamically divided and merged when the GPU devices are added and removed, respectively.

In the evaluations, we compared local and remote GPU devices on DistGPU Cache in terms of regular expression match query throughput. We also compared the DistGPU Cache with remote GPU devices and the original document-oriented data store in terms of performance. We showed that although the local GPUs case outperforms the remote GPUs case by 1.08x, the remote overhead is quite small and can be compensated by a high horizontal scalability to add more GPU devices via a network. We also showed that DistGPU Cache with GPU devices significantly outperforms the original data store.

Acknowledgements This work was partially supported by Grant-in-Aid for JSPS Research Fellow. H. Matsutani was supported in part by JST PRESTO.

References

1. Apache Couch DB. `http://couchdb.apache.org`
2. MongoDB. `http://www.mongodb.org`
3. Duato, J., Pena, A., Silla, F., Mayo, R., Quintana-Orti, E.: rCUDA: Reducing the Number of GPU-based Accelerators in High Performance Clusters. In: Proc. of the International Conference on High Performance Computing and Simulation (HPCS'10). pp. 224–231 (Jun 2010)
4. Morishima, S., Matsutani, H.: Performance Evaluations of Graph Database using CUDA and OpenMP-Compatible Libraries. ACM SIGARCH Computer Architecture News 42(4), 75–80 (Sep 2014)
5. Morishima, S., Matsutani, H.: Performance Evaluations of Document-Oriented Databases using GPU and Cache Structure. In: Proc. of International Symposium on Parallel and Distributed Processing with Applications. pp. 108–115 (August 2015)
6. Naghmouchi, J., Scarpazza, D.P., BereKovic, M.: Small-ruleset Regular Expression Matching on GPGPUs: Quantitative Performance Analysis and Optimization. In: Proc. of the International Conference on Supercomputing (ICS'10). pp. 337–348 (Jun 2010)
7. Shukla, D., et al.: Schema-agnostic Indexing with Azure DocumentDB. In: Proc. of the International Conference on Very Large Data Bases (VLDB'15). pp. 1668–1679 (Aug 2015)
8. Sitaridi, E.A., Ross, K.A.: GPU-accelerated String Matching for Database Applications. The VLDB Journal pp. 1–22 (Nov 2015)
9. Suzuki, J., Hidaka, Y., Higuchi, J., Yoshikawa, T., Iwata, A.: ExpressEther - Ethernet-Based Virtualization Technology for Reconfigurable Hardware Platform. In: Proc.of International Symposium on High-Performance Interconnects. pp. 45–51 (August 2006)
10. Vasiliadis, G., Polychronakis, M., Ioannidis, S.: Parallelization and Characterization of Pattern Matching using GPUs. In: Proc. of the International Symposium on Workload Characterization (IISWC'11). pp. 216–225 (Nov 2011)
11. Zu, Y., Yang, M., Xu, Z., Wang, L., Tian, X., Peng, K., Dong, Q.: GPU-based NFA Implementation for Memory Efficient High Speed Regular Expression Matching. In: Proc. of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'12). pp. 129–140 (Feb 2012)