OurRocks: offloading disk scan directly to GPU in write-optimized database system

Won Gi Choi, Doyoung Kim, Hongchan Roh, and Sanghyun Park

Abstract— The log structured merge (LSM) tree has been widely adopted by database systems owing to its superior write performance. However, LSM-tree based databases face vulnerabilities when processing analytical queries due to the read amplification caused by its architecture and the limited use of storage devices with high bandwidth. To flexibly handle transactional and analytical workloads, we proposed and implemented OurRocks taking full advantage of NVMe SSD and GPU devices, which improves scan performance. Although the NVMe SSD serves multi GB/s I/O rates, it is necessary to solve the data transfer overhead which limits the benefits of the GPU processing. The primary idea is to offload the scan operation to the GPU with filtering predicate pushdown and resolve the bottleneck from the data transfer between devices with direct memory access (DMA). OurRocks benefits from all the features of write-optimized database systems, in addition to accelerating the analytic queries using the aforementioned idea. Experimental results indicate that OurRocks effectively leverages resources of the NVMe SSD and GPU and significantly improves the execution of queries in the YCSB and TPC-H benchmarks, compared to the conventional write-optimized database. Our research demonstrates that the proposed approach can speed up the handling of the data-intensive workloads.

Index Terms—Database architectures, High performance computing, LSM-tree, Query processing

1 Introduction

LOG structured merge (LSM) tree [1] has been widely owing to its write-friendly features. As LSM-tree is also designed to derive high bandwidth of the underlying hardware platform, including non-volatile memory express (NVMe) SSD, it has gained wide acceptance for the storage engine of various database management systems (DBMS).

As an example of the LSM-tree, RocksDB, which is an embedded open-source key-value store [2] optimized for fast, low latency storage and developed by Facebook, is used in numerous applications to handle write-intensive workloads. Several DBMSs, including MyRocks [3], MongoDB [4], Sherapa [5] that manage their data files with RocksDB, have been implemented. Furthermore, Facebook has migrated their existing database, which stores the social activities, to MyRocks, to take advantage of write and space efficiency [6]. However, several issues remain because studies for leveraging LSM-tree as storage engine of existing databases are still underway.

Real-time analytics applications that deal with large-scale data require a new generation of data management systems. For example, real-time recommendation and fraud detection services dealing with data from mobile and IoT devices require a functionality that handles fast concurrent transactions (OLTP) and analytics (OLAP). Various vendors and academic groups have implemented systems like VoltDB, HekaTon, and H²TAP to support Hybrid Transactional/Analytical Processing (HTAP). These systems maintain a single type of data organization by exploiting modern hardware to avoid lag caused by data conversion [7]. While LSM-tree has the potential to be used as an engine for HTAP solutions because of its optimization for transactional operations, it is not designed friendly to process the analytical queries for large-scale data.

First, the existing LSM-tree based DBMS, which are driven by CPU, cannot fully utilize the resource of high-performance storage such as NVMe SSD, especially during the processing of long-term queries. NVMe SSD serves multi-GB/s I/O rates and is now being used as primary storage units in the enterprise. Nonetheless, the general CPU processor makes limited use of the throughput and iops of NVMe SSD even if it leverages several optimization techniques, including multi-threading, advanced vector extension (AVX), and asynchronous I/O.

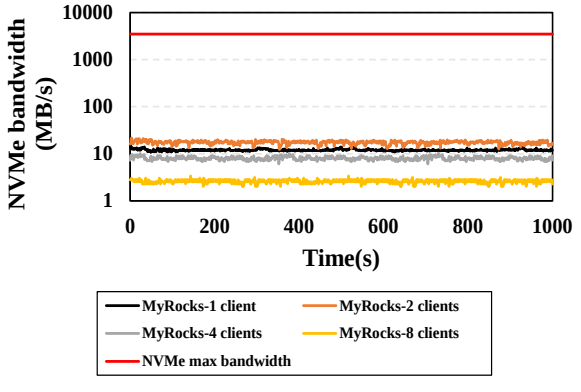


Fig. 1. NVMe bandwidth during the processing of an analytic query by MyRocks (TPC-H query 6)

- *W. G. Choi is with the Department of Computer Science, Yonsei University, Seoul, Republic of Korea. E-mail: cwk1412@yonsei.ac.kr.*
- *D. Kim is with Department of Computer Science, Yonsei University, Seoul, Republic of Korea. E-mail: kem2182@yonsei.ac.kr.*
- *H. Roh is with the Machine Learning Infra Lab, SK Telecom, Seongnam-si, Republic of Korea. E-mail: hongchan.roh@sk.com.*
- *S. Park is with the Department of Computer Science, Yonsei University, Seoul, Republic of Korea. E-mail: sanghyun@yonsei.ac.kr.*

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2020.3027671, IEEE Transactions on Computers

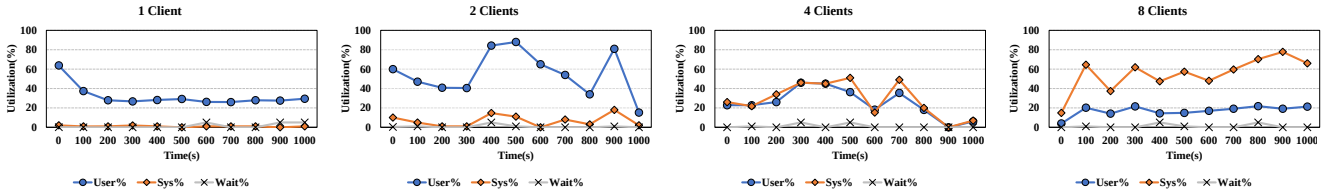


Fig. 2. CPU utilization per core during the processing of an analytic query by MyRocks (TPC-H query 6). User (%) denotes the CPU usage by user space processes. Sys (%) denotes the CPU usage by the kernel space. Wait (%) denotes the CPU idle time because of I/O waiting.

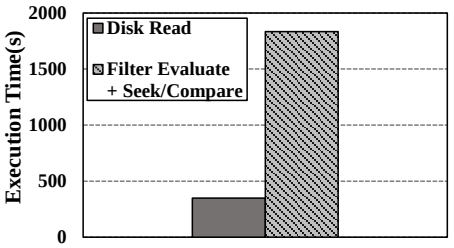


Fig. 3. Execution times of the operations during the execution of an analytic query by MyRocks (TPC-H query 6)

Fig 1. depicts the variance of NVMe SSD's bandwidth when MyRocks, which stores its files on the device, performs an analytic query. Compared to the maximum bandwidth of NVMe SSD, the result shows that the original approach of MyRocks for query processing poorly utilize the NVMe SSD's bandwidth (only 1% of maximum bandwidth). Further, NVMe utilization decreases as the number of clients is increased, even if multiple client sessions are executed simultaneously. Fig 2. depicts CPU utilization per core for the corresponding cases. As the number of clients is increased, the CPU overhead from context switching sharply increases and prevents the database from optimally utilizing the NVMe bandwidth.

Moreover, as depicted in Fig 3, MyRocks requires a considerable amount of time to seek a record's key by comparing it with the others and to evaluate filter conditions in a query, compared to the disk read. This indicates that the CPU-driven query processing performed by MyRocks cannot keep pace with the data loading speed offered by the NVMe SSD device. As shown in Fig 2, higher CPU usage occupied by the user space and context switching compared to I/O wait time demonstrates that the database requires a novel architecture to resolve the CPU intensive tasks. The bottleneck due to the limited capability of the computation leads to poorly exploiting the throughput of the NVMe SSD, which degrades its overall performance when processing analytic queries.

Second, read amplification related to the central processing unit (CPU) occurs whenever the database processes the analytic queries because of the structural characteristics of LSM-tree architecture. The amplification can induce significant performance degradation in large-scale databases.

The main contribution of this study is resolving the concerns by constructing an optimal system architecture to accelerate the analytic computation with a graphics processing unit (GPU). We present OurRocks, a new LSM-tree based database system architecture forked from the MyRocks branch. OurRocks leverages NVMe SSD and GPU resources to improve the analytic query processing while maintaining existing write-friendly features. An intuitive manner to accelerate the scan performance of the LSM-tree based database is utilizing a filtering predicate pushdown approach with bulk loading. This approach offloads the tasks of checking whether the data satisfies the condition of the queries to the available computing resources.

In addition, we implemented an OurRocks architecture that supports pipelined scan and direct memory access (DMA) so that it hides or resolves the bottleneck and derives full potential of the processing ability of the GPU. Eliminating the bottleneck from transferring data is necessary to fully benefit from the GPU device. The delay from loading the data in the disk to the GPU memory space causes the device to be idle, which degrades the overall throughput. The burden for I/O requests exacerbates the bottleneck because the LSM-tree is a persistent structure.

We verified the efficiency of OurRocks with widely-known benchmarks, including the analytic queries.
In summary, the contributions of this paper are as follows:

- To the best of our knowledge, OurRocks is the first academic project that leverages the LSM-tree engine and supports hybrid transactional/analytical processing. Existing LSM-tree based database has limited use of storage devices with high bandwidth during the processing of analytic queries because of its structural characteristics. In particular, core operations for table scan in LSM-tree generate significant CPU overhead and prevent the database from keeping pace with the data loading speed offered by NVMe SSD device. Our study primarilly focuses on the construction of an optimal architecture to utilize GPU device, thereby improving the scan performance of the LSM-tree and resolving the underutilization of NVMe resource.
- Our approach can be widely applied to the data management systems that leverage the LSM-tree engine.
- OurRocks is implemented based on MyRocks database and still benefit from all the features of MyRocks.
- Our evaluation result demonstrates that OurRocks exhibits noticeable performance improvement in the scan operation, including the simple queries in YCSB benchmark and the complex queries of TPC-H benchmark.

The remainder of this paper is organized as follows. Section 2 briefly reviews the background information,

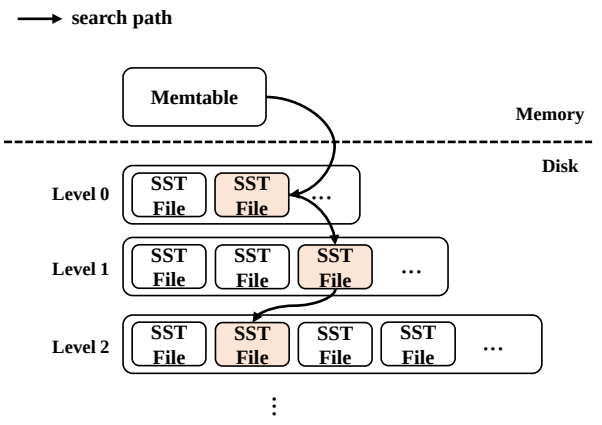


Fig. 4. Search procedure in LSM-tree

including LSM-tree, DMA and MyRocks database. In Section 3, we describe the basic idea of OurRocks with the motivation and implementation details. In Section 4, we present the experimental results on OurRocks. In Section 5, we introduce previous studies related to our work. Finally, we present the conclusions and directions for future works in Section 6.

2 Background

2.1 LSM-tree and RocksDB

LSM-tree is an underlying data structure for several systems, including BigTable [8], HBase [9], LevelDB [10], and RocksDB. As the demand for the efficient handling of the large-scale data increases, studies on LSM-tree have emerged. In this section, we briefly explain the architecture of LSM-tree employed in our implementation.

Fig 4. illustrates the overall structure of the LSM-tree and the sequence of the search operation. The LSM-tree uses an in-memory write buffer called the Memtable in addition to the disk-level structure to utilize sequential writes. However, it is a persistent data structure that stores most of the data on the disk in the format of sorted sequence table (SST) file. SST files store data in key-value form sorted by key. Each level except for 0, which is adjacent to the Memtable, possesses SST files with non-overlapping data key ranges.

The search operation for a key begins from the Memtable to the last level until the key is found or considered not present. Although binary search is performed referring to the file's metadata during the search operation, additional CPU and I/O cost is required to find files that contain the desired keys. Range queries require searching through all levels for finding keys that fall within the desired key range. This incurs read amplifications for physical and logical reads. As the data keys are scattered throughout the levels of the LSM-tree, seek operations are required during a table scan query to search the initial position at each level. Similarly, compare operations are also required during this stage to verify whether or not a record is included in a table and to retain the order of the data key. These operations generate significant CPU overhead.

RocksDB is an LSM-tree based key value store optimized for fast, low latency storage, similar to the flash memory storage. It is forked from the LevelDB, which was implemented by Google as a basic LSM-tree structure. However, RocksDB provides features to derive better performance and can be applied to real environments. RocksDB is widely used as a storage engine for various database systems owing to its superior and consistent write performance.

2.2 PCIe storage to GPU DMA

In general, two duplicate operations are necessary to enable the GPU to handle the file contents in the disk; the contents from the disk are first read into an intermediate CPU buffer and subsequently transferred to the GPU memory. The CPU controls the operations that load the data from the storage device to GPU memory. The intervention of CPU for data transfer disturbs the overall system operation, particularly when a large amount of data is required to be transferred.

Nvidia-devised GPUDirect RDMA (remote direct memory access) [11] enables a direct path for data communication between the third-party peer device, specifically network interface card, and GPU, exploiting standard features of the PCI Express (PCIe). The use of GPUDirect RDMA allows for the significant burden of CPU to be reduced for data transfer.

Similar to GPUDirect RDMA, the implementations of DMA approach between PCIe storage devices and GPU have been suggested. Project Donard [12], [13], NVMMU [14], SPIN [15], and GPUDirect Storage [16] are examples of architectures that leverage copy engines to asynchronously move large blocks of data over PCIe rather than via loads and stores. The DMA approach between the storage and GPU enables the offloading of I/O jobs to the DMA controller so that the processors can perform other tasks simultaneously to achieve the high throughput. The approach also enables the I/O-intensive application to benefit from GPU resources. We implemented our table scan operation of the LSM-tree, which exhibits a write-friendly structure, by making optimal use of direct data transfer technique.

2.3 MyRocks

MyRocks is a database system that uses RocksDB as a backend storage while retaining all the features of MySQL. The MySQL-structured relational information requires encoding in plain key-value format for storing it in the LSM-tree structure. Each database table in the RocksDB storage engine comprises a primary key index, and potentially, a set of secondary key indices. The primary key index includes the record data of the entire row and the secondary key index contains information of the values of the indexed columns.

Fig 5. illustrates how to encode the structured information to a key-value format. Each index based on the primary and secondary keys has its unique id automatically generated in four bytes. The MyRocks uses the id of the index to search the location which is a target of the point query or a starting point of the range query. If the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2020.3027671, IEEE Transactions on Computers

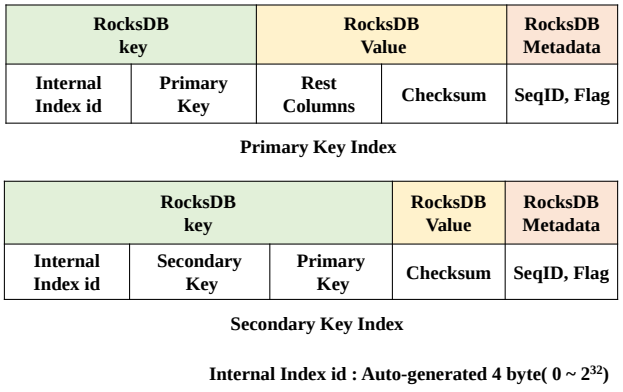


Fig. 5. MyRocks's physical record format

data are stored based on the primary key index, a unique id and value of the primary key constitute the key of RocksDB. The value of the RocksDB that corresponds to the key is composed of the checksum and the sequence of values of the remaining columns. When the secondary key index is stored in the RocksDB, a unique id of secondary index and the values of secondary keys and primary key are grouped for the key of the RocksDB. The value of the RocksDB corresponding to the key contains only the checksum because the secondary index does not require any information from other columns.

MyRocks executes the input queries based on a nested loop join algorithm with the table scan operations. A table scan operation reads the data blocks from the storage and returns the row records of the table, successively. The nested loop join algorithm joins the records retrieved by the table scan operations. The query planner and optimizer determine the join order of the tables.

RocksDB storage engine performs a table scan operation by searching the records based on the primary and secondary key indexes constructed for the table. The storage engine reads the SST files to acquire the data blocks that contain the records of the table. The storage engine handler of MyRocks converts the physical format of the records in the data blocks to a logical format. The logical format is that the values in RocksDB are mapped to each column entry in the table, indicating the format that the database can recognize.

The handler checks whether the logical record satisfies the conditions of the query. If the values of the record satisfy the condition, then the scan operation continues on the subsequent tables to execute the join operation. Otherwise, the record is excluded, and the next record in the table currently in progress becomes a new candidate for the join operation.

In other words, MyRocks constructs an iterator that retrieves and merges the records from the data files in each level of RocksDB in the order of the keys of records. MyRocks' iterator repeatedly retrieves the records one by one to evaluate the filter condition. This sequential procedure requires an excessive amount of CPU resources and repeatedly calls small read I/Os, which cannot utilize the NVMe's full bandwidth.

3 OurRocks

This section describes the architecture called OurRocks proposed in this study. The motivation of our study is discussed in subsection 3.1. In the subsections 3.2, 3.3, and 3.4, we present detailed accounts of the techniques utilized in the implementations. The primary framework and a naive attempt to utilize GPU resources in an LSM-tree-based database are outlined in subsection 3.2. Solutions for the framework are then suggested in subsections 3.3 and 3.4, thereby alleviating the heavy cost from data transfer and enabling the architecture to fully benefit from GPU resources.

3.1 Motivation

3.1.1 Scan performance degradation

Compared with the databases that manage data files with B+tree based storage engines, such as InnoDB, LSM-tree based databases exhibit superior performance in terms of write efficiency. However, it has a relatively weak scanning performance under filter conditions and requires more CPU resources and storage I/O due to the read amplification of the LSM-tree. The read amplification denotes that more data blocks are required to read than the actual number of blocks storing the data. These additional block reads lead to the requirement of additional computing resources for validating the contents of the unnecessary blocks with the predicates of the query as well as storage I/O.

MyRocks supports the software filtering approach which can early filter the unnecessary read for data blocks based on secondary key indices. However, the approach requires additional storage I/O for storing secondary indexes and cannot cover the columns that are not designated as secondary keys. We mainly target the performance improvement of the table scan. OurRocks offloads the additional computing overhead to the GPU to alleviate the performance degradation in the table scan.

3.1.2 NVMe SSD to GPU performance

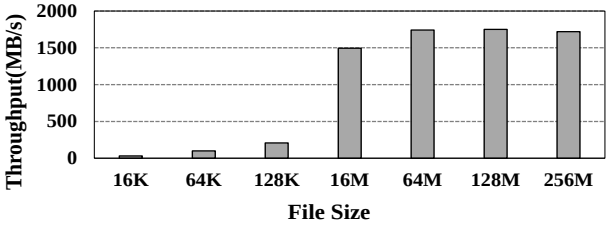


Fig. 6. Throughput of DMA transfer

Fig 6. depicts the results after the evaluation of the DMA transfer throughput. The throughput was calculated by measuring the execution time taken to copy the files written in NVMe SSD devices to the GPU memory space. As the size of the file increases, the throughput also increases and then converges. The DMA transfer exhibits better performance when the size of the file is relatively large, and the maximum throughput is reached.

The OurRocks files, which are created in megabytes to induce sequential writes and higher bandwidth of the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2020.3027671, IEEE Transactions on Computers

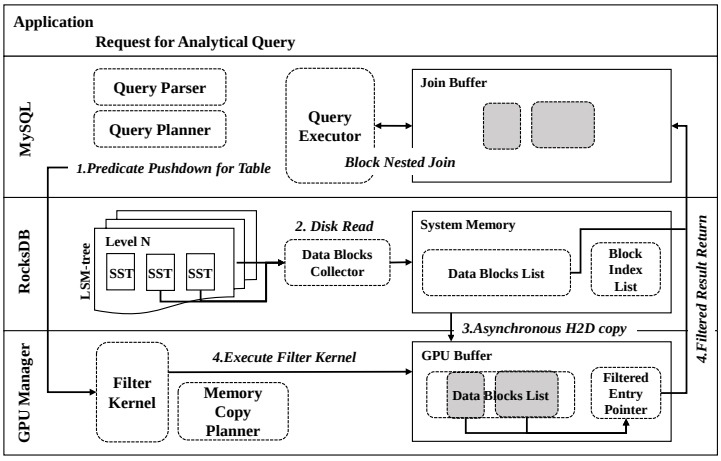


Fig. 7. Overall structure of OurRocks with filtering predicate pushdown

storage device, can also derive the optimal throughput of the DMA transfer.

3.2 Filtering predicate pushdown with GPU acceleration

OurRocks comprises three layers: MySQL query engine, RocksDB storage engine, and GPU manager. The query and storage engines are derived from existing MyRocks modules. The GPU manager module is newly implemented to manage the GPU resources and transfer the data from RocksDB files to the GPU memory to execute a kernel function.

Fig 7. depicts the basic design of OurRocks, following the fundamental procedure of the GPU accelerated application. OurRocks leverages the GPU to boost the table scan performance of LSM-tree based storage engine by implementing the filtering predicate pushdown (FPP) approach. This approach offloads the computational overhead of decoding the required value from the record's physical format and comparing the data with the predicate's pivot values. Subsequently, GPU distributes the overhead to the computing units which are abundantly available.

The query parser and planner modules in MySQL engine parse queries to obtain information about the filtering predicates and determine the target table to be scanned via the FFP approach. OurRocks populates the predicates and table information on the successive layers. By comparing the unique id for the table's primary index with the key range of the files, the RocksDB engine traverses the LSM-tree and collects SST files that contain the records of the target table. The data blocks of the files are read from the disk to the system memory, and the records and record indices in the data blocks are constructed in the form of arrays to be duplicated to the GPU memory. The record index indicates the record's offset in a block. The GPU threads refer to the record index to access the address of the record. A single thread may need to handle multiple records because the index is sparsely constructed with respect to the records. OurRocks read data files in batch and processes the operations that verify whether the table contains the record and evaluate the record based on the condition by utilizing GPU resources. Filtered results are returned to the query engine in sorted order.

Fig 8. depicts the details of the OurRocks GPU manager which utilizes CPU-driven data transfer. The GPU manager plans to transfer the records and record index array to the GPU memory. It leverages CUDA stream [17] to overlap data transfer and kernel execution to alleviate significant overhead for handling the large amount of data. The GPU manager distributes the contents of the record data and record indices to the designated stream. The data and indices in the stream are also divided by the unit of the GPU block size. The GPU block size indicates the unit of execution that operates on the same GPU resource. The threads in a GPU block refer to their own record indices. The thread traverses the region in the record array designated by the record index, while decodes the records and checks whether they satisfy the filter condition. If the condition is satisfied, the array that consists of the record's key and the pointer and size of the values is asynchronously duplicated to the host memory utilizing CUDA stream. GPU manager forms the logical record by referencing the record array in the host memory stored for data transfer and the duplicated array. It also propagates the batch of the logical format of records to the query engine so that it executes the remaining parts of the query.

3.3 Scan pipelining

In applications exploiting GPU, overhead for data transfer between host memory and GPU device memory limits the overall system performance. Large amounts of data transfer aggravate the burden of the GPU copy engine so that other tasks of the applications can be delayed.

As OurRocks also handles the heavy task of transferring large amounts of data in the disk to the GPU memory, the throughput is bounded by the latency of data transfer. To avoid this, OurRocks performs other query executions and the table scan in parallel, a process known as scan pipelining. For example, OurRocks can execute join or aggregate operations while performing the table scan for the next batch of records. Scan pipelining

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2020.3027671, IEEE Transactions on Computers

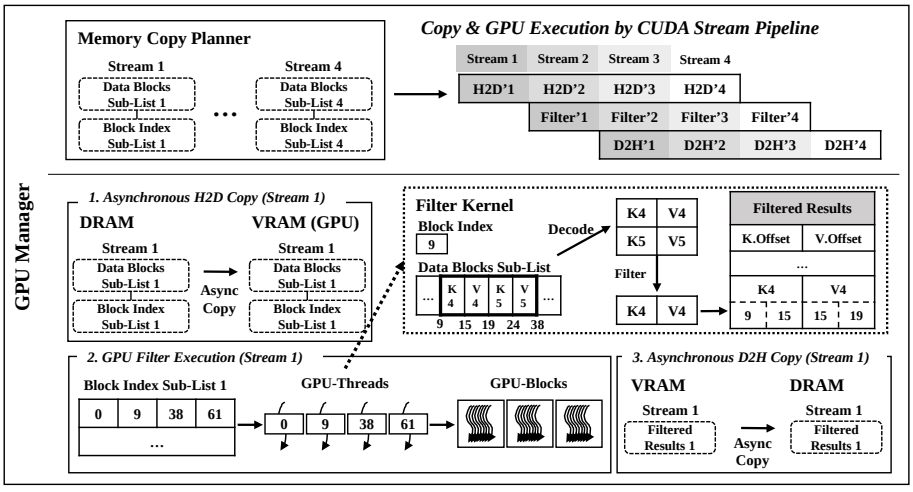


Fig. 8. Detailed description of GPU manager of OurRocks

can hide the latency for data transfer and improve the overall throughput.

3.4 Direct scan to GPU memory

The contents of the files require two separate data copy operations to be processed by the GPU. First, the contents are read from the disk to the system memory. Then, the data from the system memory are transferred to the GPU device memory. These two instances of data transfer cause a large amount of operational delay that is severe enough to counteract the benefits gathered from parallel processing by the GPU. Further, the transfer cost is a dominant factor in the determination of the overall throughput of the database as OurRocks stores most of its data on the disk. DMA transfer shortens the copy path by sharing memory space to be designated for PCIe devices. This allows the files in NVMe SSD devices to be asynchronously copied to the GPU memory by the PCIe controller without the intervention of the host CPU.

Fig 9. depicts the structure of OurRocks with DMA transfer. This architecture also begins parsing and populates the predicates and table information on the successive layers. Then, OurRocks reserves the memory space required for its next process. Compared to the version of OurRocks that exploits host CPU-driven data propagation, the current structure requires only small portions of memory for metadata blocks and a pinned space for the DMA transfer. OurRocks maintains the consistency of these metadata blocks to support mixed workloads, which include read and write operations. The new SST files are created by following out-place updates without modification of the existing files. OurRocks limits isolation level to read-committed and tracks versions of SST files. The SST files that are referenced by metadata are not updated and are garbage collected after the scan query execution if the files are determined to be invalid.

DMA transfer enables OurRocks to map the files asynchronously to the GPU memory via the pinned memory. Computing units of the GPU device can access the files mapped in the device memory. For the units to interpret the file logically, OurRocks requires metadata about the physical format of the files including the size of the data blocks. OurRocks reads the relevant metadata from the files and transfers it to the GPU memory. This process also generates I/O costs, however, these are negligible as only a small block is required per file.

OurRocks executes a kernel function that accesses the exact address of the records in a file's data blocks after the preliminary operation is completed. Data blocks in files are grouped together for the block unit of GPU processing called the GPU block. In other words, a GPU block contains the set of data blocks and the threads in a GPU block are distributed to the data blocks to perform the kernel function. Each thread distributed to a data block is allocated its own device memory containing multiple data records for processing. Its memory is determined based on the record index entries stored in a data block.

Algorithm 1 describes the details of the kernel function used by OurRocks to exploit the DMA transfer, called the DMAFilterKernel. The function requires input parameters, including the address list of the files mapped by the DMA transfer. The kernel function requires several metadata to designate memory region to be accessed by the thread. The metadata include the granularity of the GPU block that indicates the number of data blocks included in it. The number of data blocks and GPU blocks assigned to the file are included in the metadata. The schema variable includes the predicate information and the length of the column's value. The kernel function leverages the schema to decode the column's value from the physical format of the RocksDB entry. The kernel function, equipped with the parameters, transfers the key-value array of filtered records as a result to the query engine. GPU threads identify the memory region to be accessed by referring to the GPU address of a file, the data block offsets in a file, and the record offsets in a data block. DMAFilterKernel begins by selecting an address that the thread should access from the list of all file addresses. As the id is tagged to the GPU blocks in the files, the function can determine the file address based on GPU block id (Line 3 to 4). Following this, the offset of the data block in the file is calculated

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2020.3027671, IEEE Transactions on Computers

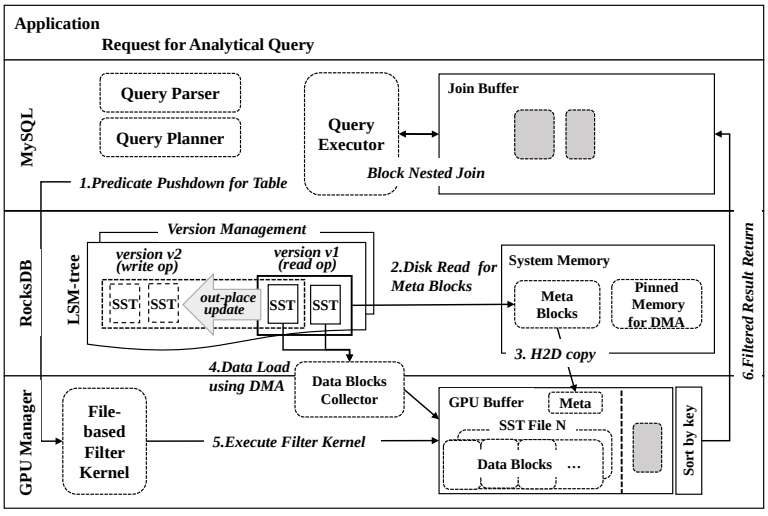


Fig. 9. The overall architecture of OurRocks that support direct data transfer (DMA)

Algorithm 1 DMAFilterKernel

Input
- file_ptr_list : list of device memory address for a file
- block_num_list : list of number of data blocks in a file
- g_block_num_list : list of number of GPU blocks in a file
- g_block_unit : number of data blocks in a GPU block
- block_size_info : information for all data blocks' size
- schema : information for table schema and predicates

Output
-d_results_idx : number of filtered records
-d_results : key-value format for filtered records

```
1  /** obtain GPU address of a file
2  containing the space covered by a GPU block **/
3  idx ← getFileIdx(blockIdx.x, g_block_num_list);
4  file_ptr ← file_ptr_list[idx];
5  /** obtain GPU address of a data block
6  covered by the GPU block **/
7  block_offset
8     ←getBlockOffset
9     (block_num_list, g_block_num_list, block_size_info);
10 block_ptr ← file_ptr + block_offset;
11 /** obtain the number of index in a data block
12 to allocate the index to a thread **/
13 num_restarts ← getNumIdx(block_ptr, block_size_info);
14 /** allocate the start offset and number of task
15 for a thread to process **/
16 start_offset, num_task
17    ← allotTask(threadIdx.x, num_restarts, g_block_unit);
18 /** execute filtering function **/
19 d_results_idx, d_results
20    ← DecodeNFilterOnSchema
21       (block_ptr, start_offset, num_task, schema);
22
```

based on the metadata related to data and GPU blocks (Line 7 to 10). The tail of the data block stores the number of record indices that is the basis for allocating the work of the thread. As the function holds the block size information and the offset of the data block, it can access the address that stores the number of record indices and obtain the corresponding value. The number of indices is denoted by num_restarts (Line 13). Based on the id of thread, num_restarts, and the GPU block granularity, the function assigns corresponding region of the memory to each thread. The record indices in the data block are distributed to the threads. The region of the thread is decided by the index that indicates the start offset of the record in a data block and the number of records to be considered (Line 16 to 17). Finally, the device function DecodNFilterOnSchema is executed, including the address of the data block, start offset, end offset, and the schema (Line 19 to 21).

Algorithm 2 presents the details of the DecodeNFilterOnSchema executed by one thread. DMAFilterKernel designates the region of the memory to be accessed by each thread. It is represented by the data block address and start offset in the data block, and the number of records to be processed. The end offset can be calculated based on the number of records to process. The function designates the starting and end points of the memory region to be accessed by each thread, which are called subblock and limit. Subblock and limit can be calculated by referring the record indices stored in the data block. The thread traverses the memory from the subblock to the limit (Line 2 to 3). The function sequentially decodes the record entries from the starting point. Each record entry is structured as a sequence of the size and data of the corresponding key and value. As RocksDB stores the key with a prefix key encoding that avoids the storage of redundant parts, the thread constructs a key by combining the newly emerging parts and the shared parts stored via tracking during the traverse (Line 6 to 8). The thread verifies whether or not the record to be decoded is included within the target table by confirming the first four bytes of the constructed key, which indicates the unique id of the primary key index (Line 10 to 12). If the record is adjudged to be included within the target table, the values of the required columns are decoded from RocksDB's value pointer by referring the schema information that contains the value's length (Line 14). After the values of the column are decoded, the validation function is executed by combining the values with the condition operator and the pivots present in the schema information (Line 16). If a record's value is determined to be valid for the condition, its key and value addresses and its size are

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2020.3027671, IEEE Transactions on Computers

Algorithm 2 DecodeNFilterOnSchema

```
    Input
    - block_ptr : device memory address for a data block
    - start_offset : thread's start offset in a data block
    - num_task : number of records for a thread to process
    - schema : information for table schema and predicates

    Output
    -d_results_idx : number of filtered records
    -d_results : key-value format for filtered records
 1  /** allocate the space range for a thread to process **/
 2  subblock ← startPtr (block_ptr, start_offset);
 3  limit ← endPtr(block_ptr, start_offset, num_task);
 4  while subblock < limit do
 5  |  /** decode key from the designated pointer **/
 6  |  key_ptr, shared, non_shared, value_ptr, value_size
 7  |     ← decodeEntry(subblock)
 8  |  key ← makeKey(key_ptr, shared, non_shared)
 9  |  /** check whether the table contains the record **/
10  |   if first4Byte(key) != schema.table_key then
11  |   |continue;
12  |   endif
13  |  /** decode values from record's value pointer **/
14  |  decoded_values ← convertRecord(schema, value_ptr);
15  |  /** compare values with predicate's pivots **/
16  |  bool match ← condValid(decoded_values, schema);
17  |  if match == true then
18  |  |idx ← atomicAdd(d_results_idx, 1);
19  |  |d_results[idx] ← (key, value_ptr, value_size);
20  |  endif
21  |  /** move to the next record's pointer **/
22  |  subblock = value_ptr + value_size;
23  endwhile
24
```

atomically inserted in the result array. An array of the complete records is constructed by referring to the result array, and it is subsequently delivered to the query engine (Line 17 to 20). The thread continues to execute the same process until it reaches the end of the designated memory region (Line 22).

4 Evaluation

4.1 Experimental setup

We conducted experiments to verify the effectiveness of OurRocks. We used two benchmarks: Yahoo! cloud serving benchmark (YCSB) [18] and TPC-H. First, YCSB provides several types of workloads that simulate large-scale cloud services. It is difficult to reproduce real RocksDB storage I/O using YCSB because it ignores key space localities [19]. However, YCSB is still one of the most widely used standard key-value store benchmarks. We evaluated OurRocks by performing workloads C, D, and E of the YCSB. OurRocks was found to significantly improve the performance of the existing database system. Sequentially, we used TPC-H, which is widely known as a decision support benchmark that comprises a suite of business oriented analytic queries and datasets, to confirm the effect of our implementation in relatively complex queries. We used 100 GB TPC-H datasets. To evaluate the scan performance which is our main target to improve, we loaded the TPC-H dataset and performed the representative analytic queries that contain the scan operations. Our experiment results demonstrated that OurRocks outperforms the conventional database system by up to 6.2 times.

TABLE 1
Experiment Environment

SPECIFICATION	
CPU	Xeon Processor (8-core) 4110
RAM	32GB DDR4 PC4 2666
Storage	1TB Samsung NVMe SSD 960 Pro
GPU	NVDIA Tesla V100 (base) NVDIA Tesla K80
OS	Ubuntu 14.04
File System	ext4
GPU Driver	384.183 version
CUDA	9.0 version

Table 1 specifies the details of the environment used in our experiments. We also implemented the CPU acceleration version of MyRocks with AVX intrinsic instructions to compare the performance with OurRocks.

Similar to GPU processing, AVX can perform concurrent filter operations on the data records stored in the CPU register. Using AVX, MyRocks can decode the value of the target column by traversing the key-value pairs in the data blocks and store the value to the register. The AVX instructions simultaneously process few record values and mark the ones that satisfy the filter condition.

We evaluated the performance using five types of databases: MyRocks, MyRocks-AVX, OurRocks-GPU, OurRocks-ASYNC, OurRocks-DMA. MyRocks denotes the original version, which is a baseline. MyRocks-AVX denotes MyRocks accelerated with AVX instructions. OurRocks-GPU denotes OurRocks with CPU-driven data transfer. OurRocks-ASYNC denotes OurRocks-GPU with scan pipelining. OurRocks-DMA denotes OurRocks accelerated with DMA approach.

4.2 YCSB

We used java database connectivity package implemented in YCSB project to evaluate the performance of OurRocks. A Java client accesses the instances of databases and executes the queries of the workloads. As OurRocks does not support the update query, we used the default workloads C, D, and E to generate point and range queries with the insert queries. YCSB workloads simulate specific circumstances in cloud systems. Workload C is a read-only benchmark, workload D is composed of point queries and insert queries, and workload E is composed of range queries and insert queries.

We created a table with six fields including 64-bit random integer and 32 bytes random strings. We loaded 50 million of row entries to the table to execute the workloads. We performed the workloads with a single thread because OurRocks does not support multi-threaded query execution yet. The multi-threaded execution requires the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2020.3027671, IEEE Transactions on Computers

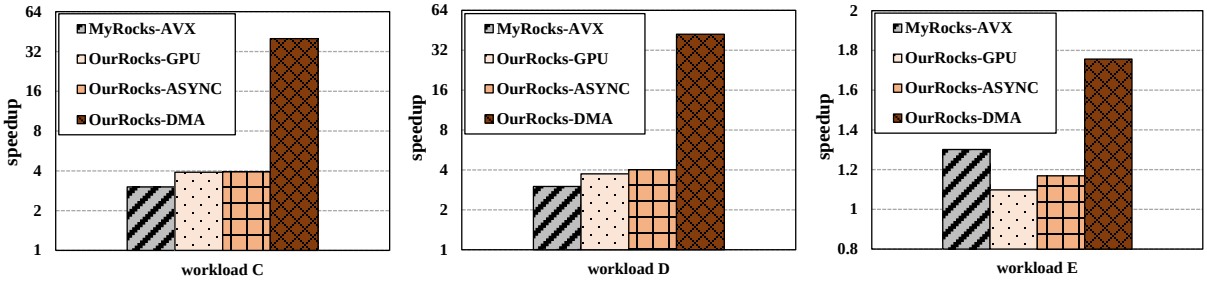


Fig. 10. Experimental results for the workloads of the YCSB benchmark

implementation of management modules for GPU resources to allow the threads to access the GPU simultaneously. Instead, we clearly confirmed the performance gain of OurRocks in the query execution by the single thread.

Fig 10. shows the speedup ratio of implementations with the acceleration modules as compared to MyRocks. We demonstrated that by offloading the filter operations to the sufficient computing resource, the performance of the workloads can be improved. In all cases, including OurRocks-DMA, implementation using the acceleration module outperforms the baseline.

The result exhibits different tendencies depending on the type of workload. In workloads C and D, OurRocks-GPU and OurRocks-ASYNC outperform MyRocks-AVX. In workload E, all the cases have lower improvements as compared to the baseline. The scan queries in the workload E include the external sorting procedure with file I/O, which is a heavy load enough to minimize the effect of computational offload. In addition, the approaches exploiting GPU except OurRocks-DMA are slower than those using AVX, although they outperform MyRocks.

The performance gain with GPU utilization depends on the selectivity of the query's filter conditions. OurRocks-GPU and its optimization, which intervenes query execution and scan operation, require data transfer between the system and GPU memories to process the filter kernel functions. If the number of entries returned by the filtering operations is large, the data transfer cost from the GPU to the system memory is increased. Furthermore, as the GPU cannot utilize the data format of the query and storage engine, the cost for transformation between the raw data and the format is also not negligible. Although we implement the pipeline that asynchronously copies the data to alleviate the overhead from data transfer, additional cost for leveraging the GPU exceeding the performance gain can occur based on the selectivity of the scan query. Experimental results in the case of workload E shows that the average execution time per query of GPU acceleration is slower than that with AVX acceleration.

The results demonstrate that data transfer including disk I/O is a bottleneck in the database system. The DMA resolves this bottleneck by the asynchronous transfer from the disk to the GPU memory, thereby utilizing the GPU processing capabilities. OurRocks-DMA significantly reduces the execution time of queries in all the cases. It also accelerates the database by up to 40 times in workloads with point queries and 1.7 times in workload with scan queries.

4.3 TPC-H query execution

We evaluated the performance of the database in cases when the simple scan queries are requested before the evaluation of the TPC-H queries. Evaluation of the queries can help in the effective verification of improvements in the scan operation because of the exclusion of other costly operations. The example queries are performed against a part table of the TPC-H dataset. The table comprises nine columns, including the primary key. The size of the part table is roughly 2 GB. The size of the table is relatively small and therefore, the frequency of disk read or data transfer to the GPU memory is relatively low. The queries comprise a single filter predicate on an integer type and a variable-sized column, and an operation that counts the number of rows satisfying the condition.

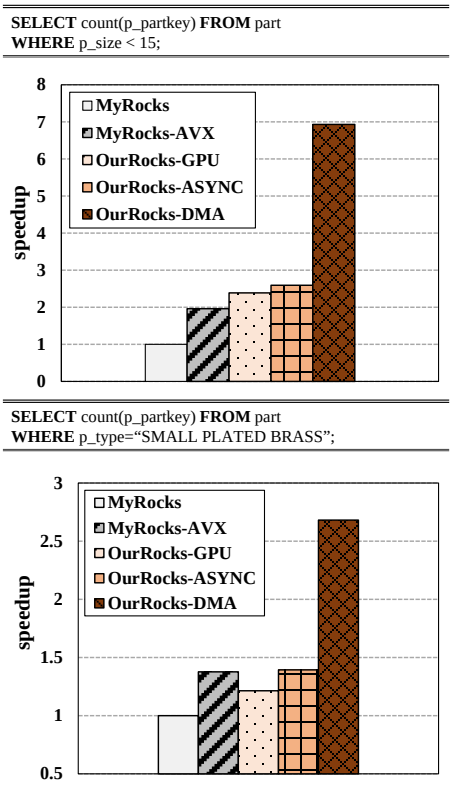


Fig. 11. Experimental results for the simple scan queries

Fig 11. depicts the experimental results when the queries are performed. The experimental results show that offloading scan filtering operations to the LSM-tree based

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2020.3027671, IEEE Transactions on Computers

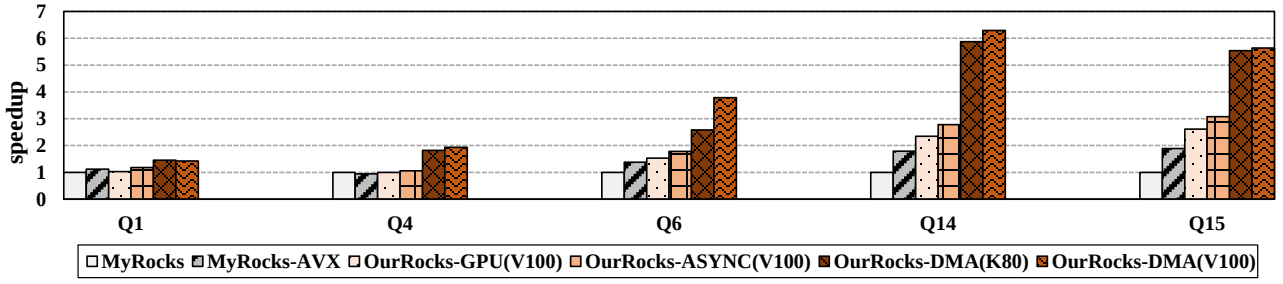


Fig. 12. Experimental results for TPC-H scan queries

storage engines leverages the abundant processing resources, thereby improving query performance. Compared with the baseline, MyRocks-AVX can speed up performance by 1.9x in the case of the experiment with a integer column. When using the GPU device, execution time is reduced even further. The database with GPU devices can handle more data records at once than the CPU vector processing. When the database performs the queries on simple filters, the GPU resources can be effectively utilized. OurRocks-GPU improves the performance by up to 2.3x and OurRocks-ASYNC by up to 2.5x. As the DMA alleviates the performance degradation from the disk read, OurRocks-DMA significantly accelerates the query execution. The performance of OurRocks-DMA increases by 6.9x compared with the baseline. In the case of the experiment with a variable-sized column, the result shows a similar trend, and OurRocks-DMA has up to 2.6x performance improvement.

To evaluate the effect of our implementation in more complex analytic queries, we performed the TPC-H queries for pre-loaded dataset. We selected representative queries that induce the full scan operation on a table sufficiently large to fully utilize the GPU resources. The query optimizer of the databases performs the table scan operation when the target table does not have any suitable indexed columns in filter conditions. While the databases can avoid the table scan with the index search, considerable storage cost may be required to create and access the index. Although OurRocks only focuses on accelerating the table scan operation, the improvement of the table scan operation noticeably contributes to processing analytic queries.

Fig 12. denotes the speedup ratio as compared to MyRocks. MyRocks-AVX has competitive or slightly better performance, 1.3 times – 1.8 times better compared to the original version of MyRocks. Although MyRocks-AVX accelerates the conditional statement checking by utilizing the vector processing, the size of the vector registers is not sufficient to store the large number of data entries, which limits the capability of parallelism. In addition, MyRocks-AVX still has difficulty in resolving the burden to read the data from the disk.

OurRocks-GPU and OurRocks-ASYNC improve the performance of the query execution compared to MyRocks-AVX. In OurRocks-ASYNC case, the speed-up ratio is up to 3 times compared to the baseline, MyRocks. The experimental results demonstrate that the performance gain of leveraging GPU grows as the amount of data to process increases. However, the disk-based databases have difficulty in drawing the full capability of GPU owing to the high cost as a result of the data transfer.

In all cases, OurRocks-DMA shows significant improvement compared to the other implementations regardless of the GPU type because DMA approach enables the database to resolve the bottleneck of the data transfer, thereby fully benefitting from the GPU. The OurRocks-DMA with Tesla V100 device improves the execution by 1.4 to 1.9 times in queries 1 and 4, which are rarely improved by other implementations. As for the other queries, the speed-up ratio increases by 3.7 times to 6.2 times. OurRocks-DMA allows the database to keep up with the fast data load from NVMe SSD device by fully utilizing GPU parallelism.

4.4 Time breakdown analysis

We separated the characteristic operations during query execution. Data transfer denotes the time taken for data movement from the NVMe device. Data transfer in MyRocks denotes the time required for disk read. Data transfer in OurRocks denotes the sum of direct data transfer time from the NVMe to the GPU and transfer time from the GPU to the host memory to return the filtered results. In MyRocks, we measured seek and compare times by calculating the time required by the iterator to seek the key and return the record to the query engine, excluding the disk read time. In OurRocks, we measured the time by calculating the time required by the database to check whether the SST files contain the table's records and return the filtered results to the query engine in sorted order. Evaluate denotes the execution time of evaluation of the filter conditions. In OurRocks, it denotes the execution time required by filter kernel. Etc. denotes the other operations, such as aggregation, grouping, and ordering operations, except for operations of table scan. We marked each execution time as a ratio to clearly indicate the overhead of the operations in the query processing.

As depicted in Fig 13, OurRocks exhibits similar trends on Query 6, Query 14, and Query 15. The ratio of Data transfer, Seek and Compare, and Evaluate in terms of total execution time is relatively high in the case in these cases. OurRocks significantly improves the corresponding times and leads to overall performance improvement. On the other hand, a different trend is observed for Query 1 and Query 4. The ratio for operations independent of table scan is dominantly high. In Query 1, aggregation operations such as average, sum, presented as Etc., account

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2020.3027671, IEEE Transactions on Computers

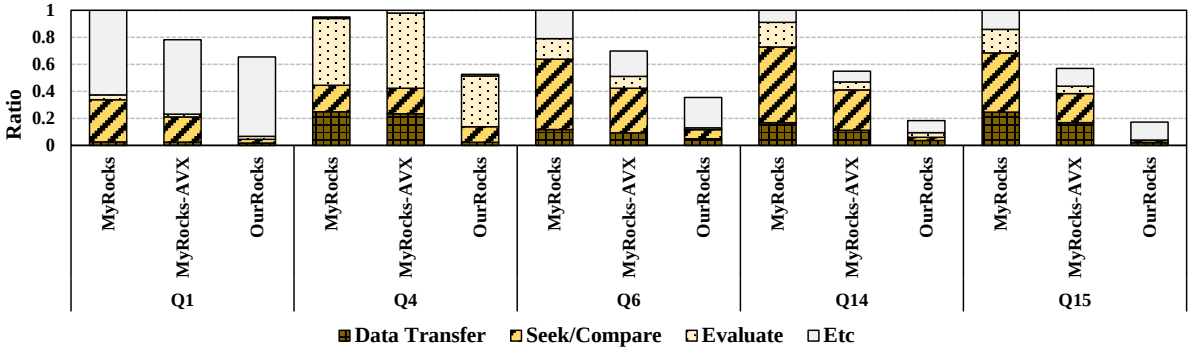


Fig. 13. Time breakdown analysis for TPC-H scan queries

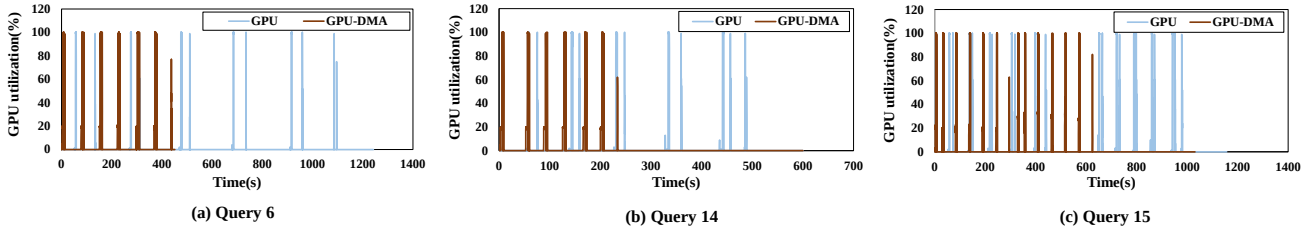


Fig. 14. Comparison of GPU utilization in TPC-H scan queries

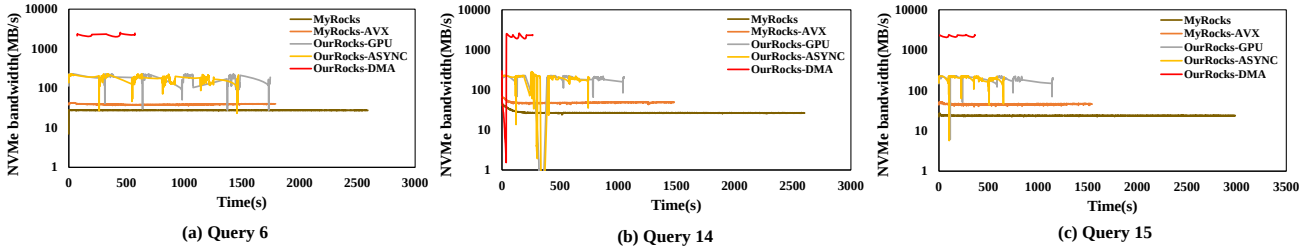


Fig. 15. Comparison of NVMe bandwidth in TPC-H scan queries

for a significant enough portion to reduce the effectiveness of OurRocks. In Query 6, while the Evaluate operation accounts for a large amount of total time, the conditional statement of Query 6 includes a subquery for a table that is outside the scope of OurRocks' table scan operation and the Evaluate time includes the execution time for this subquery.

OurRocks may be the best choice when processing the queries that have a large proportion of table scan operations and if the target table to be scanned is large enough to incur CPU-intensive tasks. Further, the case when the number of results returned by evaluating the condition is small is also advantageous as the data transfer time for filtered results is reduced. As shown in Fig 13, MyRocks-AVX also utilizes the vector processing so that it boosts evaluate performance and disk read slightly and reduce the burden of seek and compare operations. However, the vector registers' size is not sufficient to store the large number of data entries, which still incurs iterative seek and compare operations and limits the performance improvement.

4.5 Resource utilization

Fig 14. shows the effect of DMA on GPU utilization when performing three queries that OurRocks significantly improves. The utilization is calculated by measuring the percentage of time over the past second during which the kernel function is executed on the GPU. GPU-DMA represented by a brown line denotes the GPU utilization in OurRocks-DMA. GPU represented by a sky-blue line denotes the GPU utilization in OurRocks-ASYNC. Similar pattern is shown in three cases. The total time of kernel function processing occupies the small portion in total execution time. The preparation time for GPU processing, including data load from the disk and data transfer to the memory of GPU, is a dominant factor that determines the overall execution performance. The delay from the preparation prevents the GPU from being continuously utilized. The period between kernel executions in the sky-blue line is longer than in the brown line. OurRocks-DMA eliminates this bottleneck such that the database fully utilizes the high throughput of GPU per unit time. As OurRocks exploits the GPU to accelerate only filtering operations in

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2020.3027671, IEEE Transactions on Computers

TABLE 2
PERFORMANCE PER COST

	COST (US$)	PERFORMANCE PER COST (TOTAL AVERAGE - PROCESSING UNIT ONLY)	PERFORMANCE PER COST (TOP 3 - PROCESSING UNIT ONLY)	PERFORMANCE PER COST (TOTAL AVERAGE - TOTAL SERVER DEPLOYMENT)	PERFORMANCE PER COST (TOP 3 - TOTAL SERVER DEPLOYMENT)
CPU	546 $	1x baseline	1x baseline	1x baseline	1x baseline
CPU(AVX)	546 $	1.36x baseline	1.72x baseline	1.36x baseline	1.72x baseline
CPU(Tesla V100)	6825$	0.18x baseline	0.38x baseline	1.12x baseline	2.43x baseline
GPU(Tesla K80)	699 $	1.01x baseline	1.94x baseline	2.03x baseline	3.90x baseline

SQL queries and designates remaining operations to CPU, the overall utilization of GPU is low. The extension of OurRocks to support other operations, including aggregation, will be targeted in future works.

Fig 15. presents the bandwidth of NVMe SSD device while the databases perform the three queries. The graphs denote the peak points of the bandwidth of NVMe SSD which present the amount of I/O per unit time required by the databases. The figure shows that MyRocks and MyRocks-AVX constantly request I/Os to NVMe SSD. MyRocks-AVX achieves the higher bandwidth of NVMe SSD than MyRocks, because it accelerates the filtering operations, increasing the amount of disk read per unit time. However, the databases driven by CPU do not meet the maximum bandwidth of the NVMe SSD device.

OurRocks-GPU and OurRocks-ASYNC show a higher level of bandwidth as compared to CPU-driven databases, because OurRocks using GPU simultaneously read several data files to populate the GPU memory. OurRocks-ASYNC has slightly shorter interval of the peak time as compared to the OurRocks-GPU because it intervenes CPU execution and disk read. However, the OurRocks using GPU also does not meet the maximum degree because data transfer between the host memory and GPU memory still limits the bandwidth of NVMe SSD. OurRocks-GPU and ASYNC require software overhead of RocksDB for synchronously reading files to memory, which prevents the databases from efficiently exploiting NVMe bandwidth. Additional overhead is consumed for preparing GPU utilization, such as the computation process for thread distribution tasks. OurRocks-DMA shows the maximum bandwidth of NVMe SSD with the use of DMA and fast computation with the use of the GPU. Fig 15. demonstrates that OurRocks-DMA fully utilizes the resource of NVMe SSD, which improves the overall performance of the database.

4.6 Performance-per-Cost

We measured performance-per-cost by performing TPC-H experiments with NVDIA Tesla K80 which is similar in price to the CPU used. As depicted in Fig 14, the overall GPU utilization factor is low in query execution. Filtering kernel that doesn't include iterative aggregate operations doesn't require a lot of GPU resources such as memory bandwidth and core frequency. Further, the total number of threads executed in each GPU is as same as the number of block indices because of the block structure of RocksDB. Accordingly, if the size of the GPU device memory for batch reading is similar, there is no significant performance difference between the two Tesla V100 and Tesla K80 devices although Tesla V100 has better device capabilities, as depicted in Fig 12. However, the type of filtering condition that works can influence the experimental results according to the GPU type. Query 6 has a higher GPU utilization rate per unit time compared to Query 15, and this causes a difference in overall performance due to the difference in GPU performance. Further, the performance difference between two GPUs also depends on the filtering ratio because a kernel which copies the result array from data blocks is executed in GPU.

We measured the performance-per-cost metric by dividing the average performance by the server deployment cost. We calculated the deployment cost in two cases, including the processing units only that indicate CPU and GPU, and the total server cost. The cost of OurRocks includes the cost of CPU and GPU.

As depicted in Table 2, OurRocks-DMA with Tesla K80 exhibits meaningful improvement compared to the baseline by factors of 1.01 to 3.9. As Tesla V100 has the highest price, performance-per-cost is significantly low when the server deployment cost is calculated based on only processing units. However, if the other components' sum price is a considerable degree, Tesla V100 also shows remarkable improvement in performance-per-cost. Consequently, the experiments demonstrate that OurRocks is more cost-effective than the general LSM-tree based database that performs table scan operations driven by CPU.

4.7 Energy consumption

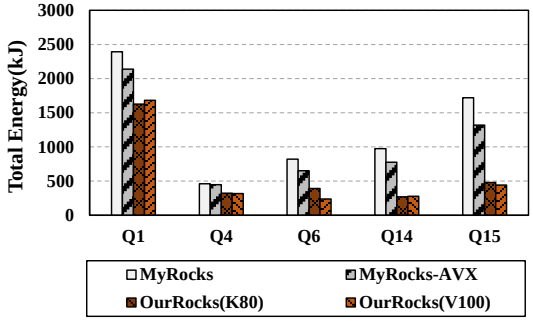


Fig. 16. Total Energy consumption when processing analytic queries

We measured the overall power consumption to demonstrate the benefit of OurRocks by connecting the server via an external power measurement device. We utilized

SJPM-C16 [20], which is capable of measuring power up to 3520 Watts. We evaluated the total energy consumed by MyRocks and OurRocks-DMA after executing the analytical queries. Fig 16. depicts the energy consumptions of the server in kJ when it operates each database. The results demonstrate that OurRocks would be advantageous in energy saving than MyRocks and AVX-optimized MyRocks because of its significantly improved processing time. OurRocks is observed to consume less energy by a factor between 1.4 and 3.8 compared to MyRocks.

5 Related works

5.1 Database techniques for GPU

GPU has received a significant amount of attention in both academic and industrial domains owing to its processing capability. Zhang et al. [21] designed a novel in-memory key-value store, Mega-KV. With a GPU-optimized cuckoo hash table, the Mega-KV offloads index data structure and operations to the GPU, thereby achieving high performance and high throughput. Ashkiani et al. [22] developed a dictionary data structure for the GPU called GPU LSM, which improved the insertion and deletion performances by parallelizing the sort and merge process in the LSM-tree.

Paul et al. [23] implemented a novel hash join for GPU utilizing the characteristics of modern GPU including shared memory atomic instructions, CUDA stream and unified memory. Wang et al. [24] supported coordinated multi-query execution by implementing a query scheduler and a device memory manager, thereby allowing GPU to be shared. GPL [25] improves GPU utilization by leveraging features of modern GPU, including channels and concurrent kernel execution, to construct a pipelined query execution engine. Li et al. [26] suggested HippogriffDB, which is a scalable GPU-accelerated OLAP system. It utilizes compression methods to fit into the GPU architecture, and improves kernel efficiency using the operator fusion and double buffering technique. Further, it resolves data transfer overhead by extending Hippogriff [27] that supports peer-to-peer communication between SSD and GPU.

MapD [28] and Kinetica [29] are commercial products widely known for GPU-accelerated databases. They provide high-performance query processing and an analytic visualization module. SQream [30] is also one of the widely known GPU-powered database products. It can handle large-scale datasets by utilizing columnar and compressed data format.

Previous studies have typically focused on maximizing the utilization of GPU resources during the processing of data loaded onto the main memory by taking advantage of the characteristics of modern GPU. This study primarily focuses on optimizing the procedure of the table scan in LSM-tree which stores most data on the disk, by leveraging the characteristics of modern GPU, thereby maximizing the utilization of the resources of NVMe device.

5.2 Storage device with near data processing

With the advent of intelligent devices, several studies utilizing such devices have emerged to handle big data. Based on near-data processing [31], smart or intelligent SSDs [32], [33], [34] extend flash-based storage to offload the database tasks, leveraging the high internal bandwidth of the devices. YourSQL [35] presented the product-strength database building on commodity NVMe-SSDs, which supports early filtering to avoid unnecessary data traffic. There are similar approaches [36], [37] to offload the query processing to FPGA of high-end products.

OurRocks offloads selection queries with filter condition to GPU to leverage sufficient computing resources and boost performance.

6 Conclusion and Future Works

This study shows that effective use of the GPU and NVMe SSD device can resolve the vulnerability of a scan operation in write-optimized databases. The proposed idea involves the offloading of scan operations with filtering predicates to the GPU and leveraging the DMA to eliminate the bottleneck originating from data transfer.

While OurRocks still benefits from all the features of a write-optimized database, it accelerates analytic queries. Experimental results demonstrate that OurRocks significantly improves the performance of the analytic queries, as compared with the conventional database.

There are several avenues for future works. To support the consistent and better performance of OurRocks, the query optimizer needs to be implemented in order to select the join order of tables optimal for the GPU. Further, we intend to implement a data-caching policy that can store the filtered results of hot data in memory to reduce additional transfer cost.

Acknowledgment

This research was supported by the MSIT (Ministry of Science and ICT), Korea, under the SW Starlab support program (IITP-2017-0-00477) supervised by the IITP (Institute for Information & communications Technology Promotion). This research also received support from the Next-Generation Information Computing Development Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT (NRF 2015M3C4A7065522). The corresponding authors are Hongchan Roh and Sanghyun Park.

References

[1] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The log-structured merge-tree (LSM-tree)," *Acta Inform.*, vol. 33, no. 4, pp. 351–385, Jun. 1996.
[2] RocksDB [Online]. Available: http://rocksdb.org/.
[3] MyRocks [Online]. Available: https://myrocks.io/.
[4] MongoDB [Online]. Available: https://www.mongodb.com/.
[5] Sherpa [Online]. Available: https://yahooeng.tumblr.com/post/120730204806/sherpa-scales-new-heights/.
[6] S. Dong, M. Callaghan, L. Galanis, D. Borthakur, T. Savor, and M. Strum, "Optimizing Space Amplification in RocksDB," in *CIDR*, 2017, vol. 3, p. 3.

[7] F. Özcan, Y. Tian, and P. Tözün, "Hybrid Transactional/Analytical Processing: A Survey," in *Proceedings of the 2017 ACM International Conference on Management of Data*, Chicago, Illinois, USA, May 2017, pp. 1771–1775, Accessed: Jun. 04, 2020. [Online].

[8] F. Chang *et al.*, "Bigtable: A Distributed Storage System for Structured Data," *ACM Trans. Comput. Syst.*, vol. 26, no. 2, pp. 4:1–4:26, Jun. 2008.

[9] A. S. Aiyer *et al.*, "Storage infrastructure behind Facebook messages: Using HBase at scale," *IEEE Data Eng. Bull.*, vol. 35, no. 2, pp. 4–13, 2012.

[10] LevelDB [Online]. Available: http://leveldb.org/.

[11] GPUDirect RDMA [Online]. Available: https://docs.nvidia.com/cuda/gpudirect-rdma/index.html/.

[12] S. Bates, "Donard: NVM Express for Peer-2-Peer between SSDs and other PCIe Devices," *Retrieved July*, 2018.

[13] S. Bates, "Project donard: Peer-to-peer communication with nvm express devices." 2014.

[14] J. Zhang, D. Donofrio, J. Shalf, M. T. Kandemir, and M. Jung, "NVMMU: A Non-volatile Memory Management Unit for Heterogeneous GPU-SSD Architectures," in *2015 International Conference on Parallel Architecture and Compilation (PACT)*, Oct. 2015, pp. 13–24

[15] S. Bergman, T. Brokhman, T. Cohen, and M. Silberstein, "SPIN: Seamless Operating System Integration of Peer-to-Peer DMA Between SSDs and GPUs," *ACM Trans. Comput. Syst.*, vol. 36, no. 2, pp. 5:1–5:26, Apr. 2019.

[16] GPUDirect Storage [Online]. Available: https://devblogs.nvidia.com/gpudirect-storage/.

[17] CUDA Stream [Online]. Available: https://devblogs.nvidia.com/how-overlap-data-transfers-cuda-cc/.

[18] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking Cloud Serving Systems with YCSB," in *Proceedings of the 1st ACM Symposium on Cloud Computing*, Indianapolis, Indiana, USA, 2010, pp. 143–154.

[19] Z. Cao, S. Dong, S. Vemuri, and D. H. C. Du, "Characterizing, modeling, and benchmarking RocksDB key-value workloads at Facebook," in 18th ${USENIX} Conference on File and Storage Technologies ({FAST}$ 20), 2020, pp. 209–223.

[20] External power measurement device https://seojunelectric.com/

[21] K. Zhang, K. Wang, Y. Yuan, L. Guo, and R. Lee, "Mega-KV: a case for GPUs to maximize the throughput of in-memory key-value stores," *Proceedings VLDB Endowment*, 2015.

[22] S. Ashkiani, S. Li, M. Farach-Colton, N. Amenta, and J. D. Owens, "GPU LSM: A Dynamic Dictionary Data Structure for the GPU," in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2018, pp. 430–440.

[23] J. Paul, B. He, S. Lu, and C. T. Lau, "Revisiting hash join on graphics processors: a decade later," Distributed and Parallel Databases, Jan. 2020, doi: 10.1007/s10619-019-07280-z.

[24] K. Wang et al., "Concurrent analytical query processing with GPUs," Proceedings VLDB Endowment, vol. 7, no. 11, pp. 1011–1022, Jul. 2014.

[25] J. Paul, J. He, and B. He, "GPL: A GPU-based Pipelined Query Processing Engine," in Proceedings of the 2016 International Conference on Management of Data, San Francisco, California, USA, Jun. 2016, pp. 1935–1950, Accessed: Jun. 04, 2020. [Online]

[26] J. Li, H.-W. Tseng, C. Lin, Y. Papakonstantinou, and S. Swanson, "HippogriffDB: balancing I/O and GPU bandwidth in big data analytics," Proceedings VLDB Endowment, vol. 9, no. 14, pp. 1647–1658, Oct. 2016.

[27] Y. Liu, H. Tseng, M. Gahagan, J. Li, Y. Jin, and S. Swanson, "Hippogriff: Efficiently moving data in heterogeneous computing systems," in 2016 IEEE 34th International Conference on Computer Design (ICCD), Oct. 2016, pp. 376–379.

[28] T. Mostak (2017) MapD Open Sources GPU-Powered Database. [Online]. Available: https://www.omnisci.com/blog/mapd-open-sources-gpu-powered-database/.

[29] Kinetica [Online]. Available: https://www.kinetica.com/.

[30] SQream [Online]. Available: https://sqream.com/.

[31] R. Balasubramonian *et al.*, "Near-Data Processing: Insights from a MICRO-46 Workshop," *IEEE Micro*, vol. 34, no. 4, pp. 36–42, Jul. 2014.

[32] D.-H. Bae, J.-H. Kim, S.-W. Kim, H. Oh, and C. Park, "Intelligent SSD: a turbo for big data mining," in *Proceedings of the 22nd ACM international conference on Information & Knowledge Management*, 2013, pp. 1573–1576.

[33] J. Do, Y.-S. Kee, J. M. Patel, C. Park, K. Park, and D. J. DeWitt, "Query Processing on Smart SSDs: Opportunities and Challenges," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, New York, New York, USA, 2013, pp. 1221–1230.

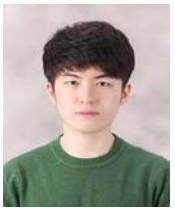
[34] Y. Kang, Y. Kee, E. L. Miller, and C. Park, "Enabling cost-effective data processing with smart SSD," in *2013 IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST)*, 2013, pp. 1–12.

[35] I. Jo *et al.*, "YourSQL: A High-performance Database System Leveraging In-storage Computing," *Proceedings VLDB Endowment*, vol. 9, no. 12, pp. 924–935, Aug. 2016.

[36] P. Francisco, "Ibm puredata system for analytics architecture," *IBM Redbooks*, pp. 1–16, 2014.

[37] L. Woods, Z. István, and G. Alonso, "Ibex: An Intelligent Storage Engine with Support for Advanced SQL Offloading," *Proceedings VLDB Endowment*, vol. 7, no. 11, pp. 963–974, Jul. 2014.

Won Gi Choi received his BS degree in 2014 from the Yonsei University, Seoul, Korea. He is currently a Ph.D. candidate at the Yonsei University. His current research interests include database systems, flash memory, and NVRAMs.



Doyoung Kim received his BS degree in Computer Science from Yonsei University, Seoul, Korea, in 2018. He received his MS degree in Computer Science from Yonsei University, Seoul, Korea, in 2020. His current research interests include key-value stores utilizing non-volatile memory or graphic processing unit.

Hongchan Roh received his BS degree in 2006, MS degree in 2008, and Ph.D. degree in 2014, all from the Yonsei University, Seoul, Korea. He is currently a research fellow for SK Telecom. His current research interests include network processors, database systems, flash memory, and SSDs.

Sanghyun Park received his BS and MS degrees in Computer Engineering from the Seoul National University in 1989 and 1991, respectively. He received his Ph.D. degree from the Department of Computer Science, University of California at Los Angeles (UCLA), in 2001. He is currently a professor at the Department of Computer Science, Yonsei University, Seoul, Korea. His current research interests include databases, data mining, bioinformatics, and flash memory.