Buffer Management in Cross Device Processing

Karthikesh Varma Chintalapati
Faculty of Informatik
Otto von Guericke University
Magdeburg, Germany
karthikesh.chintalapati@st.ovgu.de

Sai Teja Thalluri
Faculty of Informatik
Otto von Guericke University
Magdeburg, Germany
sai.thalluri@st.ovgu.de

Sreepradh Dingari
Faculty of Informatik
Otto von Guericke University
Magdeburg, Germany
sreepradh.dingari@st.ovgu.de

Vasudev Raghavendra Bidarkar
Faculty of Informatik
Otto von Guericke University
Magdeburg, Germany
vasudev.raghavendra@st.ovgu.de

***Abstract*—Disk based heterogeneous co-processing database systems have come into trend to execute database queries with good speed and accuracy. In disk based database systems, data has to be fetched from the disk to the main memory, which is a time consuming task. This is further increased when the data has to be transferred to the GPU memory. Additionally, due to the memory limitations of the GPU it is difficult to cache the entire data into the device memory. To overcome this problem, we have used a buffer manager which stores relevant data such as the buffer flags in a shared memory, whereas the input data is stored in the device.**

In this work, we have executed conjunctive predicate queries on the GPU. There is a shared space called Shared Virtual Memory (SVM) which stores flags. Both the host and the device have access to this SVM. The device updates the flags in SVM when the execution is completed, whereas the host updates the flags and evicts the pages which are no longer useful. For this project, we have worked with two eviction strategies: Greedy and FIFO. We have tested on two devices and then compared the executions times of the queries running with and without buffer manager. This comparison helps us to know how effectively a buffer manager works in achieving good performance while executing queries on the GPU. After the evaluation, it has been found that buffer manager can not be useful in catalysing the query execution.

***Index Terms*—Buffer manager, Replacement strategy, Heterogeneous systems, Shared Virtual Memory, OpenCL**

I. Introduction

Some research projects have explored running database query operations using heterogeneous devices like CPU and GPU to attain good performance and consistency [3]. Generally, GPUs are used for their higher query processing rates compared to a CPU. We use the massive computational power and parallelisation of GPU to operate on our data. CPUs have a large main memory, they are only suitable in performing serial instructions, incurring delay in execution time. Unlike the CPU, a GPU can have thousands of cores. The biggest disadvantages of using a GPU are

1) Memory limitations and the speed at which data can be copied into the GPU's memory.
2) Memory concerns are the only other primary concern outside of ability-to-be-parallelized if the amount of data needed to do the calculations doesn't fit in your GPUs memory.
3) If the code is already fast enough that it takes less time to execute on a CPU than to copy your data to the GPU's memory, then the GPU is going to do you very little good in practice.

CPUs are optimized for performing a small number of simple calculations on large amounts of data, GPUs excel in parallelising many computations on small amounts of data [3]. Hence, we use Buffer managers to allocate and prioritize the data within the limited GPU memory space. In our work, CPU copies all the data needed for a computation from the main memory to the GPU RAM, launches the GPU kernel, and instructs to copy the result data back to CPU main memory.

The aim of the project is to build a buffer manager module which prioritizes the data into data that needs to be stored in the device and data that needs to be stored in the main memory. As the space is limited in the device, buffer manager is used to make the best out of the data based on the run time behaviour and allocate the data effectively. Then, implement the module in OpenCL framework and test whether the buffer module is allocating the data effectively. This focuses mainly on

(a) Buffering data into GPU
(b) Managing and tracking flags for replacement strategy using SVM across devices.
(c) Evaluating the performance of buffer manager with FIFO and Greedy.

II. Background

In this section, we provide an overview of the typical GPU architecture and its memory spaces, and heterogeneous co-processing systems. In the next section, we discuss about the types of query executions, buffer manager, and its components.

GPUs typically have their own memory. This ranges from a few GBs to tens of GBs of GDDR5 memory [1]. Compared to main memory sizes which have been increasing as predicted by the Moore's Law, GPU memory is smaller. On the other hand, the memory bandwidth of GPUs is much higher when compared to CPUs. Generally, host and device do not share the same address space, meaning that neither the GPU can directly access the main memory nor the CPU can directly access the device memory [4].

Today heterogeneous co-processing systems i.e, CPU-GPU computing systems have come into trend to increase parallelism and speedup the various types of computations. *Robust query processing in co-processor accelerated databases* [3] which has CPU as host and GPU as device makes most of the data to run on GPU to use its computational power. But, due to the very limited memory space in GPU, if the data of a particular operation can not be cached into the GPU then the data is again moved to CPU and the host executes the operation. In this system there is no buffer manager involved but the query processor itself uses a replacement strategy to evict the unwanted data. As they were using data driven approach, if the data does not fit in the memory then the other processor takes charge of the operation. Without a standard buffer manager, problems like inefficient data distribution and query execution takes place which results in increase in execution time and transfer times and ineffective storage management.

Generally in-memory database systems doesn't have a buffer manager as the entire data will be readily available and managed from the main memory (which is itself a buffer now). The data in the device memory has to be cached effectively in order to avoid the extra transfer time from the host. We need a module which can keep a track on how often a particular data has been accessed by the device to execute a particular operation. This can either be done by having pointer counters or flags which keeps on incrementing and lets us know how many times particular data has been accessed. This record helps in prioritizing the data from most frequently used to less frequently used and evict the less frequently or less recently used based on the replacement policy. A standard buffer manager with proper replacement policy is useful in disk based systems to prioritize the data as it helps in minimising the number of input and output operations.

GPUs have global memory space which is generally the entire GPU memory where the data will be fetched from CPU to operate on GPU. The GPU main memory is considered as a buffer pool which is called as CL buffer pool where the entire buffer pool is subdivided into fixed size pages. These pages are chunked based on the total buffer pool size. The chunk size must be optimal for transfer and execution of the data. If the chunk size is too large than the ideal size then the execution time is more and transfer time is less whereas if the chunk is smaller than the ideal chunk size then the transfer time is more and execution time is less. This problem has to be resolved by selecting the chunk which has nearly equal transfer and execution times. Shared Virtual Memory(SVM) has to be created where the flags are being placed. These flags are the pointers of the data and gets updated when data is being accessed by the GPU. Both the host and the device has the access on the SVM. The host accesses the flags in SVM and decides which pages to be stored in the CL buffer and which pages to evict back to the CPU memory. Whereas GPU access the flags in SVM to update the flags when the pages are being accessed by the GPU. Both the device and host has read and write access on SVM which is called cross device processing.

[1]https://medium.com/@ashanpriyadarshana/cuda-gpu-memory-architecture-8c3ac644bd64

III. Concepts

A. *Types of executions*

How a query is executed plays a significant role in determining how effective of a parallelism can be implemented. If the scheduling is done improperly, there might be adverse effects on the data such as data corruption or data loss. This is a reason why effective query scheduling and execution is important. For our project, we have considered two types of query execution models.

1) Operator-at-a-time execution
2) Iterative execution

1) Operator-at-a-time execution: As the name suggests, in this type of execution model, the data is processed operator wise. All the data corresponding to a certain operator is pushed to the device, executed, and the results are retrieved. Then the execution moves on to the next operator. This seems streamlined at first glance but the number of writes and reads required is high.

For example, if two operators need to be applied to two inputs, then initially the first input is written, processed, and its partial output is read back. This is repeated for the second input. Then the first partial output is again written to the executor and the final output is read back after processing it through the second operator. This is done again for the second partial output. The amount of times data needs to be *written to* and *read from* the device for each input increases exponentially with the number of operators. And since data transfer time is a major bottleneck in multi-device execution, this needs to be minimized as much as possible. If the two inputs are mutually exclusive then multiple reads and writes are redundant.

2) Iterative execution: Iteratively executing a query takes fewer amounts of reads and writes than operator-at-a-time execution. This is because data is processed through all the operators at once. This advantage is even more significant when we look at the transfer times for data between CPU and GPU. Because of this reason, we have chosen to employ iterative execution for this project.

B. *Heterogeneous buffer manager*

Buffer management is a essential component in a traditional database architecture because it ensures that there is enough

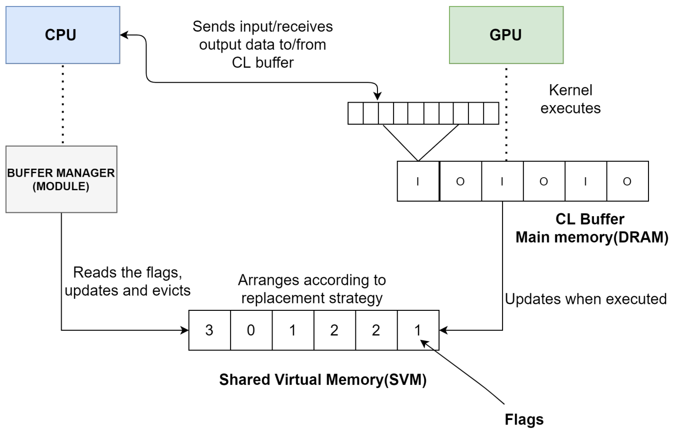
CPU
Sends input/receives output data to/from CL buffer
GPU
Kernel executes
BUFFER MANAGER (MODULE)
I
O
I
O
I
O
CL Buffer Main memory(DRAM)
Reads the flags, updates and evicts
Arranges according to replacement strategy
Updates when executed
3
0
1
2
2
1
Shared Virtual Memory(SVM)
Flags

Fig. 1. Block Diagram

memory available for the data which is important. In the classical design, all data structures are stored on fixed-size pages in a translation free manner [5]. The rest of the system uses a simple interface that hides the complexities of the I/O buffering mechanism and provides a global replacement strategy across all data structures. Runtime function calls to pin/unpin page provide the information for deciding which pages need to be kept in RAM and which can be evicted to external memory (based on a replacement strategy like LRU, LFU, and FIFO).

This buffer manager sits on the host (CPU) and controls the entire procedure. The buffer manager module which sits in the host is responsible for prioritizing the pages based on the data accessed from the host. This module keeps a background check on how often the data has been accessed by the device. When the CPU as a master sends the data to the slave (GPU), then the buffer manager module keeps track on data i.e how much data has to be allocated to the CL buffer and which data has to be sent to the device. When the data has been accessed by the GPU the device updates the flags in SVM and tells that the data has been received and about to execute. The data is cached in the sub-buffers and the respective operation takes place as instructed in the kernel. As soon as the data gets executed the device once again updates the flag in SVM and the host has both read and write access to SVM. Once the data is executed the device lets the host know that the data has been executed and it is ready to sent back to the host. The host checks the flags and fetches the output from device and updates the flags to the idle state and again sends another packet of data to the device. Buffer manager has components which work as a team to reduce the input/output accesses and effective data transfers among cross devices. The components of the buffer manager are

1) Replacement strategy
2) CL buffers
3) Shared Virtual Memory(SVM)
4) Flags

1) Replacement Strategy: To prioritize the data effectively we need a standard replacement strategy which allows the device to retain most frequently accessed data and evict the less frequently/recently/FIFO data based on our requirement.

LFU (Least Frequently Used) is a commonly used strategy where the page is evicted based on the least frequently used page. LRU is another commonly used replacement strategy where the least recently used page gets evicted from the list [2]. When the free pages in the buffer pool are running out, the buffer manager starts to keep a random subset of pages in the cooling state and most of the in-memory pages will remain in the hot state. Cooling pages are organized in a FIFO queue. The queue maintains pointers to cooling pages in the order in which they were unswizzled, i.e., the most recently unswizzled pages are at the beginning of the queue and older pages at the end. When memory for a new page is needed, the least recently unswizzled page is used. This elegant design is one of the pillars of classical database systems.

2) CL buffer: CL buffer is another important component of buffer manager. The CL buffer is nothing but the entire DRAM is considered as buffer pool and the buffer pool is divided into sub-buffers. The input data from the host is sent to the buffers.

These sub-buffers are chunked based on the optimal chunk size. The chunk size has to be optimal in transferring and executing the data for execution. We have found out that the optimal chunk size is 2 MB and you can check out why in the next section. This buffer pool stores both inputs and outputs. When the final output is obtained the output is sent back to the host which consequently clears the sub-buffer and the next set of data is sent to the device for another execution.

3) Shared Virtual Memory (SVM): SVM is another vital component of buffer manager. SVM gives the access to both device and host to read and write the flags in order to keep it updated. SVM has the equal number of sub-buffers as the CL buffers. This is because each time the data has been accessed it updates the flags and therefore there has to be equal number of chunks.

4) Flags: Flags are like a counters which keeps track on number of times the data has been accessed. This helps in getting to know which data is unnecessary which data is important to be cached in the memory. Initially, in the idle state the flags are ’0’. When the data is sent to the CL buffers then the flags gets updated to ’1’. CPU updates the flags from ’0’ to ’1’. Once the data is executed the status of the flag is updated by the GPU to ’2’. Also when the output is sent to the host then the GPU updates it to ’3’.

C. Non blocking operation

Threads are an important part of the operating system. They control the execution flow, interrupting and resuming operations as required.

By default, OpenCL runs on the CPU thread and halts it until GPU completes execution. This is called a *Blocking operation*. But for our purposes, flags need to be changed and data inserted and retrieved even when the GPU is busy executing a query. This is not possible through blocking

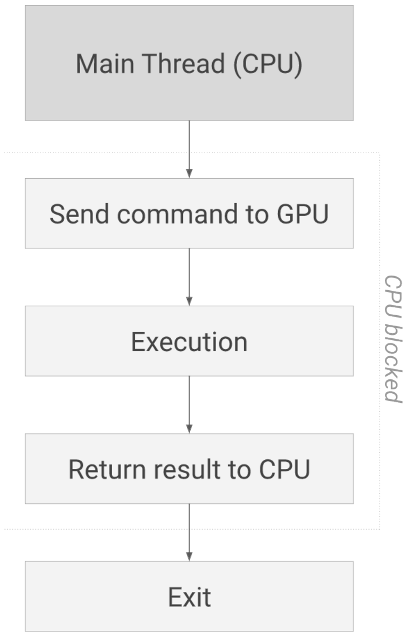

Fig. 2. Blocking operation

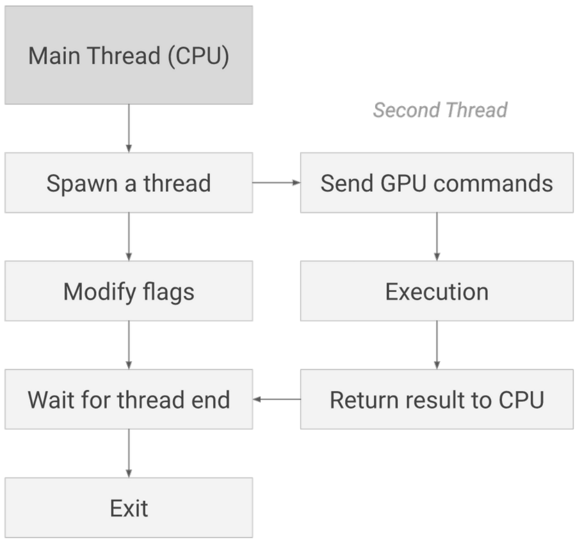

Fig. 3. Non-blocking operation

operation. So, we need to use a thread to allow CPU to continue independently of GPU.

Create a new thread for sending commands to the GPU. During the execution, the CPU can read and alter the data as necessary. This is used to send the next input data and update flags during the query execution (pipelining). This way two threads are created to make the host and the device work parallely independent of each other.

IV. Partial Implementation

A. *Creating an SVM Space*

Shared Virtual Memory (SVM) in OpenCL allows the host and device to share a memory space. This is useful in cases where frequent reads and writes of data between two devices is not preferred. Before storing any data in the shared space, a specific amount of it has to be allocated using the `clSVMAlloc()` function.

In a SVM, data consistency is managed manually with `map` and `unmap` commands to make sure any information is synchronized. *Mapping* enables the host and the device to share a singular region of virtual memory space containing pointer based data. Compared to the basic buffers offered by OpenCL, this avoids duplication of data between the host and the device. A result of this is the freeing of extra memory and eliminating the need for transferring data and the overhead it brings. Pointers initialised on the host, in OpenCL SVM, can use the data normally without any extra effort.

The sample code executes according to the following scenario:

1) Allocate SVM memory by using appropriate functions from the OpenCL framework and map the data to a virtual space.
2) Pass a pointer to SVM memory to the device to enable the OpenCL C kernel to access SVM memory.
3) The OpenCL C kernel reads the shared memory and traverses the arrays using those shared pointers.
4) The OpenCL C kernel performs operations with values from the SVM buffers and writes to the dedicated output buffer.

a. Allocating SVM Memory

Create an array in the SVM memory with the `clSVMAlloc` function. This function returns a regular pointer in the host address space. This pointer can be passed to the kernel to be used like a regular OpenCL buffer.

```
inline int* createSvm(cl_context context,
    cl_svm_mem_flags flags, size_t size)
    {
    return (int*) clSVMAlloc(context,
    flags, size, 0);
}
```

b. Accessing SVM Memory in Host Code

For the SVM memory to be synchronized internally, the host needs to *map* the memory. This has to be done once after allocating the SVM space using `clEnqueueSVMMap`.

```
clEnqueueSVMMap(queue(), CL_TRUE, CL_MAP
_WRITE, inp, inpSize * sizeof(int), 0,0,0);
```

c. Passing SVM Pointers to a Kernel

Pointers to shared memory can be passed to the kernel as a kernel argument, albeit with a different function than what is used to pass in buffers.

```
cl::Kernel kernel=getKernel(context,&err);
```

```
clSetKernelArgSVMPointer(kernel(),0,inp);
clSetKernelArgSVMPointer(kernel(),1,inp);
```

Creating CL Buffer:

```
cl::Buffer buffer(context, CL_MEM_READ_WRITE,
inpSizeBytes, nullptr);
```

Creating Kernel and setting arguments(CL Buffer)

```
cl::Kernel kernel = getKernel(context,&err);
err = kernel.setArg(0, buffer);
err = kernel.setArg(1, buffer);
```

Writing input data to Buffer:

```
queue.enqueueWriteBuffer(buffer, CL_TRUE,
0, inpSizeBytes, inp);
```

Executing Kernel:

```
queue.enqueueNDRangeKernel(kernel, 0,
cl::NDRange(numInts));
```

Reading output data from buffer:

```
queue.enqueueReadBuffer(buffer, CL_TRUE,
0,inpSizeBytes, inp);
```

A randomly generated array of integers is then passed to the kernel to measure the transfer and execution times of SVM and compare it to CL Buffers.

B. *Choosing an Optimum Chunk Size*

In the GPU, the buffer pool is subdivided into chunks for the effective transfer of data. The buffer pool has to be subdivided into equal number of chunks where choosing the optimal chunk size is important. If the chunk size is too small the transfer time is high which delays the execution time and if the chunk size is too large the transfer time is too slow.

V. Implementation

In the previous section we concentrated more on setting up the background. We created an SVM space which can be accessed both by host and the device and passed the SVM pointers into the kernel. We have also created CL buffer and wrote the input data to buffer, executed the kernel and read the output data from the buffer. We got to know that CL buffer is best fit for our input data when compared to SVM. To split the buffer into chunks an optimal chunk size of 2 MB has been chosen as it has the minimal difference in transfer and execution times.

After setting the background we have

1) Created a buffer of the maximum size and created sub buffers in it according to the chunk size that we have calculated (2MB) in the previous section.
2) Created an SVM space with a size that is equal to the number of sub buffers, to store the flags.
3) Decide the input data into the calculated size and put it in the sub buffer, and then update the flags.

In this section we will provide you how the query execution takes place. We are focusing on two operations i.e "Selection less than" and "Logical AND" operations. These are the operations that the device executes from the input data given by the host.

A. *"Selection less than" Implementation*

In this section, we talk exclusively about "Selection less than" operation and its implementation. The input data is given by the host to the device and this entire data is said to be stored in the device memory. The entire memory of the device is considered as buffer pool and split as input and output buffers. This split is based on the number of inputs and outputs.

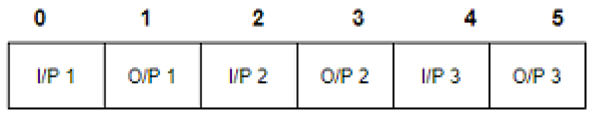

Fig. 4. Interleaved buffers for selection less than

In this selection less than operation there is one input and one output. Hence we consider the split to be equal. One half being allocated for input storage and the other being allocated for the output data. In the buffer pool we check sequentially whether the sub buffer is empty. If the sub buffer is empty then the input is stored and the consecutive sub buffer is blocked for the output of this data. Similarly another input is allocated skipping every alternative sub buffer . The shared virtual memory will also have the equal size of the output buffers. For instance, if there are 'n' number of output buffers then there has to be 'n' number of SVM buffers which has 'n' number of flags.

Flag value	Operation
0	Buffer is empty
1	Input is inserted
2	Input is processed
3	Output is inserted

TABLE I: Flag states

Now, when the data is transferred to the device from the host then the data is stored in the device memory. In the idle state the flags in shared virtual memory (SVM) are marked 'O'. When the input data is transferred to device the flags in the SVM are turned as '1' by the host. The device checks the SVM and recognises the flags which are turned '1', fetches the data and starts executing the data by sending it to the kernel. After executing the data, the GPU, as the device, has the access to the SVM to write, updates the flags to '2' so that the data can be replaced by the CPU. Once the data is executed, it is written to the output buffer and the flag is updated to '3'. Now, as the CPU has the access to the SVM and checks the entire SVM in FIFO fashion where it checks if there are any flag updates. If any buffer whose corresponding flag is '3' is found, then the related data is evicted and sent back to the host. The evicted data is replaced by a new input data and the GPU continues its execution until the query is finished.

B. "Logical AND" Implementation

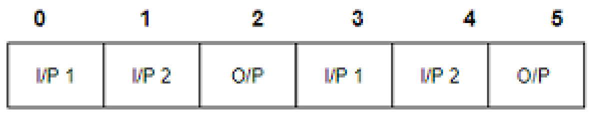

Fig. 5. "Interleaved buffers for logical AND

In the "Logical AND" operation there are two inputs and one output. In the buffer pool we check sequentially whether the sub buffer is empty. If the sub buffer is empty then the first input is stored and then checks again if any other buffer is empty. If an empty buffer is found then the second input is stored and the consecutive sub buffer is blocked for the output of this data. Similarly another input is allocated skipping every two sub buffers.

VI. Experimental Setup

As evaluation platform, we evaluated on two devices. One machine with an AMD Ryzen 5 3500 U with four cores @2.1 GHz and 8 GB main memory, and a Radeon Vega 8 graphics GPU with 2 GB of device memory. The other machine is Intel core i7 9750h with 6 cores and a Nvidia RTX 2060 graphics GPU with 6 GB memory. On the software side, we use Ubuntu 20.04 (64 Bit) as operating system and OpenCL being the framework with Clion IDE.

VII. Evaluation Results

A. Preliminary evaluation

In the GPU, the buffer pool is subdivided into equal number of chunks where choosing the optimal chunk size is important for the effective pipelining of data. If the chunk size is too small, the transfer time is high which delays the execution time, and if the chunk size is too large, the transfer time is too slow.

This is a problem while we are trying to pipeline the data. Hence, an optimal chunk size is required for the effective transfer and processing of data. Therefore the minimum difference between the transfer time and execution time of a particular data size gives us the optimal chunk size. After ten iterations, we have found out that 1 MB and 2 MB data sizes are optimal for chunk size as their difference of execution time and transfer time is minimal when compared to other data sizes.

In Fig.6 the time taken to transfer the data from CPU memory to GPU is CL Buffer transfer time and the time taken to transfer the data from SVM to GPU. The results are shown in the form of graph that has data sizes taken from 0.25 MB to 64 MB and transfer time taken is in ms.

In Fig.7 the graph depicts the optimal chunk size where the minimum difference between transfer time and execution time is observed at 2^{21} bytes (2 MB). This is almost same for all the iterations. So, we have picked this data size as a split for our input.

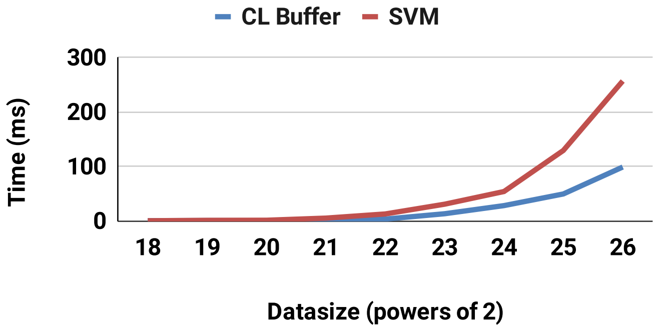

Fig. 6. Transfer Time- CL Buffer vs SVM

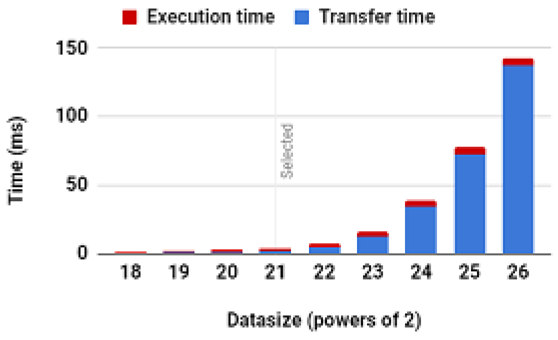

Fig. 7. No execution operation

B. Overall performance evaluations:

As mentioned in the previous section, we have evaluated the performance of our buffer manager on two devices - AMD Radeon Vega 8 and Nvidia RTX 2060. We have used FIFO and greedy as buffer manager replacement strategies. In greedy execution, the buffers are reused. For instance, inputs are allocated to buffers and when the operation is executed, the outputs are stored in adjacent buffers. As soon as the outputs are obtained, the previous input buffers are reused to store the new inputs. This way the buffers are reused. The performance of the greedy execution is nearly equivalent to that of FIFO, with a difference of only about 5%.

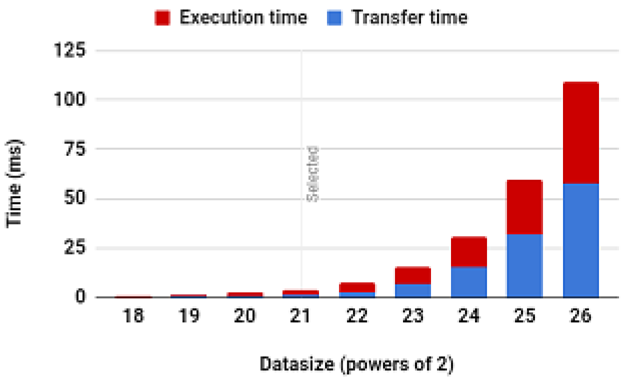

Fig. 8. Selection less than operation

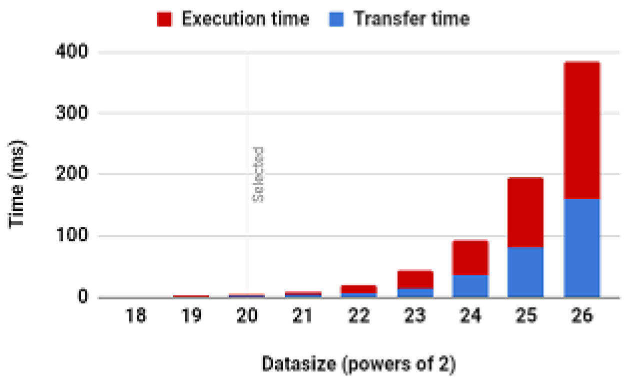

Fig. 9. Logical AND operation

As the input, depending on the input data size, an array of numbers are stored in the buffers to be sent to the GPU. As we are performing selection less than operation, we have randomly taken a number 43242 which is close to half of the range of numbers. So, the two selection less than operations are `A < 43242` and `B < 43242`. Then, a logical AND operation is performed on the outputs of the two selection less than operations to obtain the final output. So, the entire query looks like `A < 43242 && B < 43242`. These operations are performed using buffer manager strategies on both the GPUs, AMD and Nvidia, and the execution times are calculated. We also evaluate the performance of the device without the buffer manager by measuring its execution time. This way we can compare how effectively our buffer manager is performing comparatively with no buffer manager.

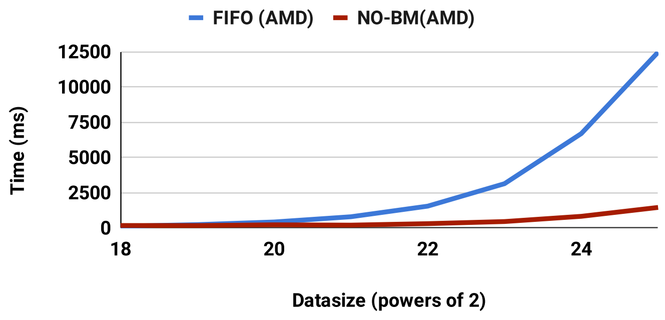

Fig. 10. Performance evaluation of FIFO and No buffer manager on AMD

As observed in the Fig. 10 we can see how the buffer manager is under performing on both the devices - AMD and Nvidia. We can depict from the graph that the AMD GPU with the buffer manager is performing worse by a factor of 8 than without a buffer manager. This is slightly better in the case of the Nvidia GPU, where Nvidia with a buffer manager (FIFO) is comparatively performing better by a factor of 5 to AMD with buffer manager. However, Nvidia with buffer manager is still under performing by a factor of 2 without the buffer manager.

There may be several reasons which are responsible for the poor performance.

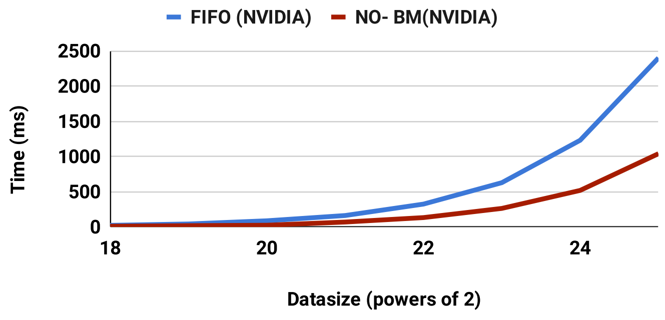

Fig. 11. Performance evaluation of FIFO and No buffer manager on Nvidia

1) AMD has a very bad SVM manager. This is because of the unsatisfactory transfer time of SVM in AMD. The penalty in the SVM manager increases the penalty of the buffer manager.
2) Another reason may be due to the replacement strategy. Using FIFO, the performance drops by a factor of 2 in Nvidia when compared to performance on Nvidia without a buffer manager. However, we have used greedy strategy as an alternative to test, but it is similar to FIFO with a minute difference of around 5%.
3) There may be performance degradation and more than expected penalties while threading in non blocking operation.

These may be some of the factors contributing to the lower-than-expected performance of the buffer manager among cross device query processing. We can conclude by saying that a buffer manager can not be used in the heterogeneous systems, in its current state, for query processing because of its poor performance.

VIII. Summary

In heterogeneous co-processing systems, due to the memory limitations of the GPU, it is difficult to cache the entire data into the device memory. To overcome this problem we have used a buffer manager. Both the host and the device have access to a Shared Virtual Memory (SVM) and they update the flags from different threads. We have tested on two devices, one being AMD Radeon Vega 8 and the other is Nvidia RTX 2060. We have found that AMD with buffer manager is performing worse by a factor of 8 when compared to AMD without a buffer manager. The reason being, AMD has a very bad SVM manager. This is because of the poor transfer time of SVM in AMD. With an input size of 2^{25} bytes using a buffer manager, it takes 12 seconds to execute the query whereas having no buffer manager completes it in 1.4 seconds.

There might be an improvement in the efficiency if some other replacement strategies such as LRU (Least Recently Used), or LFU (Least Frequently Used) are employed.

Thus, we can summarize by saying that a buffer manager can not be used in the heterogeneous systems, in its current state, for query execution because of its sub-optimal performance.

References

[1] Neumann, Thomas, and Michael J. Freitag. “Umbra: A Disk-Based System with In-Memory Performance.” CIDR 2020, 10th Conference on Innovative Data Systems Research, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings.

[2] Leis, Viktor, et al. “LeanStore: In-Memory Data Management beyond Main Memory.” 2018 IEEE 34th International Conference on Data Engineering (ICDE), IEEE, 2018, pp. 185–96.

[3] Breß, Sebastian, et al. “Robust Query Processing in Co-Processor-Accelerated Databases.” Proceedings of the 2016 International Conference on Management of Data - SIGMOD ’16, ACM Press, 2016, pp. 1891–906.

[4] Breß, Sebastian, et al. “GPU-Accelerated Database Systems: Survey and Open Challenges.” Transactions on Large-Scale Data- and Knowledge-Centered Systems XV: Selected Papers from ADBIS 2013 Satellite Events, edited by Abdelkader Hameurlain et al., Springer, 2014, pp. 1–35.

[5] Effelsberg, Wolfgang, and Theo Haerder. “Principles of Database Buffer Management.” ACM Transactions on Database Systems, vol. 9, no. 4, Dec. 1984, pp. 560–95.