Memory Management Strategies in CPU/GPU Database Systems: A Survey

Iya Arefyeva(✉), David Broneske, Gabriel Campero, Marcus Pinnecke, and Gunter Saake

University of Magdeburg, Universitätsplatz 2, 39106 Magdeburg, Germany
{iia.arefeva,dbronesk,campero,pinnecke,saake}@ovgu.de

Abstract. GPU-accelerated in-memory database systems have gained a lot of popularity over the last several years. However, GPUs have limited memory capacity, and the data to process might not fit into the GPU memory entirely and cause a memory overflow. Fortunately, this problem has many possible solutions, like splitting the data and processing each portion separately, or storing the data in the main memory and transferring it to the GPU on demand. This paper provides a survey of four main techniques for managing GPU memory and their applications for query processing in cross-device powered database systems.

Keywords: Cross-device query processing
GPU memory management · Divide-and-conquer · Mapped memory
Unified Virtual Addressing · Unified Memory

1 Introduction

In-memory database systems first were proposed in 1980s [7], and nowadays are becoming more and more popular, as RAM sizes grow and prices are dropping. The main copy of the data in these systems, in contrast to traditional disk-based database systems, is stored in main memory and can be directly accessed by the processor. Such design enables these systems to achieve significant performance improvements due to higher memory access speed. This is especially important in high-throughput applications which require fast response time.

Another emerging trend, that has gained a lot of attention of researchers in recent times, is the usage of co-processors (e.g. GPUs) in database management systems. Utilization of co-processor spans a range of system-related tasks, such as query optimization or query execution. In fact, co-processors are applied to accelerate both online analytical processing (OLAP) [3–5,9,11,15,18,20,23] and online transaction processing (OLTP) [2,10]. Furthermore, there is ongoing co-processor acceleration research [1,17] for the combination of these both processing types, called hybrid transactional/analytical processing (HTAP).

However, GPU-accelerated computing involves a significant challenge: the memory size of a GPU is limited to several gigabytes and is often smaller than

S. Kozielski et al. (Eds.): BDAS 2018, CCIS 928, pp. 128–142, 2018.
https://doi.org/10.1007/978-3-319-99987-6_10

the data to process, while the main memory nowadays can consist of hundreds of gigabytes. Storage of the whole data on the GPU can be very beneficial, since it eliminates overheads introduced by transferring the data to the GPU and back. Unfortunately, today's device memory capacity is far too limited to hold real-world databases completely. Even high-end GPUs like Nvidia Titan Xp and Nvidia Tesla V100, released in 2017, have 12 GB and 16 GB of memory correspondingly, which does not even come close to the amount of RAM that can be installed on a server machine. Hence, there is the need for strategies considering the fact that the database can only be partially moved to the GPU.

Usually, there is no shared memory between a CPU and a GPU, since their memory spaces are physically separated. The data has to be transferred from the host to the device memory over the PCI-E bus, whose bandwidth is much smaller than the GPU memory bandwidth. This problem, however, is getting less severe as bandwidths are getting higher. For instance, PCI-E 4.0, announced in 2017, provides a bandwidth of 32 GB/s which is two times higher than the bandwidth of PCI-E 3.0 (16 GB/s) that is likely installed on current machines. PCI-E 5.0, which is planned to be released in 2019, is expected to further double the bandwidth, reaching 64 GB/s.

We are optimistic that it is only a matter of time until both capacity limitations and transfer bandwidth problems degenerate to insignificance due to advantages in technological development. However, unless these issues are removed, both must be considered for memory management in a heterogeneous database system consisting of both CPU and GPU.

In the last decade, the research community suggested a variety of solutions to face these issues, e.g. splitting data into chunks or storing it in pinned host memory. In this paper, we survey the state of the art in GPU memory management, providing insights into how different approaches attempt to approximate an ideal GPU memory management model, that should be able to (i) allow for GPU memory oversubscription, (ii) utilize the GPU efficiently by overlapping transfers and computations, hence minimizing the idle time of the GPU (iii) avoid unnecessary transfers via the PCI-E bus and (iv) keep the data coherent.

The remainder of the paper is structured as follows. Section 2 provides some information about GPUs' architecture and execution model, that is necessary for understanding the rest of the paper. The existing basic approaches for GPU memory management are described in Sect. 3 along with their advantages, disadvantages, and details of their implementations in different systems. Section 4 compares the approaches and discusses how their properties help to face different challenges. Finally, Sect. 5 concludes the paper by summarizing the described techniques.

2 Background

A CPU (central processing unit) usually consists of a few powerful cores that share the same cache and control unit, and can handle a few software threads at a time. CPUs are designed to execute an instruction on a single unit of data as

quickly as possible. They are well suited for tasks, that cannot be parallelized and/or depend on each other.

In contrast, a GPU (graphics processing unit) is composed of several streaming multiprocessors, each consisting of multiple cores. Therefore, a GPU is able to handle multiple threads (sequences of instructions, that process one data element at a time) simultaneously, in a SIMD fashion.

2.1 GPU Memory Types

Each thread, or work item, has its own *registers* and *local* memory that is visible only to this thread. Threads are grouped into *thread blocks* or *work groups*, where each thread block has its own *shared* memory that is visible to all threads within a block and can be used for communication between threads. Blocks themselves are grouped into a *grid*. In contrast to the other memory types, *global*, *constant*, and *texture* memories are shared across all thread blocks. The number of threads per block and blocks per grid is defined by the programmer.

The right choice of memory types can have a significant impact on the performance and increase the speed of memory operations. However, the choice between memory types has smaller impact on the problem of limited GPU memory for co-processing. Therefore, in the techniques' description in Sect. 3 we assume that global memory is used, unless specified otherwise.

2.2 SIMD-Fashioned Thread Execution

Threads are executed in *warps* - batches of 32 threads each - which fetch data from memory together. To optimize executing behavior by exploiting spatial locality of memory accesses, threads within a warp should access sequential blocks of memory. This way, reads from global memory are performed in as few transactions as possible. This access behavior is called *coalesced memory access*. Moreover, to avoid a loss in performance, all threads in a warp should follow the same path in case of if-else statements.

To sum up, CPUs are designed to perform a few complex tasks at a time, and GPUs are good at performing one small task on many units of data. The latter is referred to as *data parallelism* - the case when the same operation is performed by threads on different units of data. OLAP queries are usually well suited for the GPU processing style, since they require applying the same operation to a lot of fields. For instance, ranking users visiting a website, grouped by their residence, requires reading fields in every tuple of a table that stores the visitors' data. OLTP, however, runs many small and different tasks (e.g., small insert, update, and delete operations), therefore usage of GPUs for OLTP might be challenging. Additionally, OLTP queries, unlike OLAP ones, change the data, and thus a good synchronization mechanism is required.

2.3 Programming and Execution Models of GPUs

A program executed on a GPU (device) is called a *kernel*, it performs operations on one element of the data and forms a basic unit of parallelism. A kernel is

invocated by a CPU (host), but cannot be controlled by the CPU after the invocation and cannot communicate with other kernels that are executed on the GPU. In case of query processing, a data element can be represented by one value of an attribute, by a tuple (in row-wise storage), a column (in column-wise storage), or by a whole table. Obviously, the latter does not allow to exploit the GPU efficiently, because it leads to a set of operations being performed by one thread on a very large chunk of data.

A GPU needs to communicate with the CPU through the PCI-Express (PCI-E) bus, whose bandwidth is much lower than the bandwidth of the GPU memory. This difference limits the performance by causing a bandwidth bottleneck, and often [8] data transfer takes much more time than processing of this data.

The data to process is either sent to the GPU prior to the kernel execution, or the GPU directly accesses pinned CPU memory during the execution. In general, there are two memory management models: *programmer-managed GPU memory* and *pinned host memory* [12]. Programmer-managed GPU memory consists of the following steps:

1. Allocating memory blocks of GPU, that are big enough to contain the input and output data.
2. Transferring the data to GPU over the PCI-E bus.
3. Calling kernels that perform operations on the data.
4. After the execution is finished, transferring the output back to the host.

The memory blocks, allocated on the GPU, should be small enough to fit into the device memory. In case the size of the data is larger than the GPU memory, the data should be split into several blocks, that are small enough to not cause a memory overflow.

Usage of pinned memory allows GPU to directly access the data in the main memory, without the need to transfer all the data to the device prior to kernel execution. Transfers to GPU are overlapped with the execution and performed on-demand and implicitly, therefore the data size is limited only by the main memory size. The two resulting memory management models are explained in details in the next section.

The most popular frameworks for GPU programming are CUDA and OpenCL. OpenCL (Open Computing Language) is an open standard for parallel programming of heterogeneous systems and is supported by several platforms, including Nvidia, AMD and Intel. CUDA is a parallel computing platform and programming model which supports only Nvidia GPUs.

3 GPU Memory Management

As mentioned in the previous section, approaches used to manage GPU memory can broadly be classified into the divide-and-conquer approach (Sect. 3.1) and usage of pinned CPU memory (Sect. 3.2). Unified Virtual Addressing Sect. 3.3 and Unified Memory (Sect. 3.4) extend these approaches by simplifying their

usage and, in case of Unified Memory, making data transfers transparent to the user. This section describes these approaches and discusses their benefits and drawbacks.

3.1 Divide-and-Conquer

The divide-and-conquer approach is used in systems like GPUQP [9] and MultiQx-GPU [21] to manage GPU memory.

When a system is issued with a query (e.g., a selection), the data is split into multiple chunks, which are sent to the GPU and processed there separately. Once a chunk is fully transferred to the GPU, the kernel is started for this chunk. After the kernel execution is finished for all the data elements, the result for the processed chunk is copied back from the GPU global memory to the main memory and is stored there until the results for all the chunks are returned. This process is repeated for each of the chunks, and the memory, taken by previous chunks and their results, is either overwritten or freed beforehand in order to make enough space in the GPU memory for the new chunks. The process ends when the execution is finished for all the chunks. All the partial results are then merged in the main memory, e.g. the union of the results is performed.

Divide-and-conquer approaches can be divided into serial and asynchronous processing. In serial processing a data chunk is transferred to the GPU, a query is executed over this data, and the result is transferred back to the CPU. Then, the next chunk is transferred and processed. Figure 1 illustrates the process of transferring one data chunk and performing operations on it. As can be seen, when processing occurs only after the whole chunk is transferred to the GPU, a big amount of time is spent waiting for transfers, while the GPU remains idle.

Asynchronous processing, as shown in Fig. 2, overlaps data transfers with execution, so that data transfers occur when the kernel is executed. Overlapping is possible on devices with compute capability $\geq$1.2, and devices with compute capability $\geq$2.0 support bidirectional (host-to-device transfers and device-to-host) simultaneous transfers occurring concurrently with computations[1].

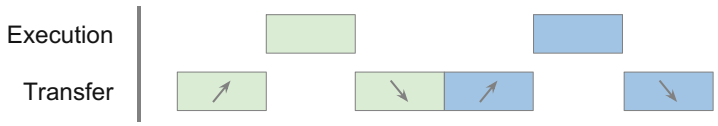

Fig. 1. Serial processing; the second chunk is transferred only after the result of the processing of the first chunk is returned.

The degree of overlap and the achieved gain in performance depend on many factors, including the complexity of the kernel, chunk sizes and the GPU used for

[1] OpenCL Best Practices Guide. Online at https://www.cs.cmu.edu/afs/cs/academic/class/15668-s11/www/cudadoc/OpenCL_Best_Practices_Guide.pdf.

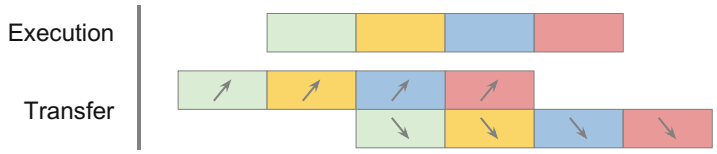

Fig. 2. Asynchronous processing; the transfer of a chunk is overlapped with the processing of the previous chunk.

computations. The biggest gain in performance can be achieved for workloads, where equal amount of time is spent for data transfer and data processing. In some cases [3], asynchronous processing of data is reported to be not much faster than serial transfers, however, in other cases [19,22] a better performance is achieved, when transfers and executions occur concurrently. In fact, even in the worst scenario, when no overlap at all occurs, the performance would be the same as if the chunks were transferred and processed serially. Therefore, the rule of thumb is to process the data asynchronously, as in the worst case there would simply be no speedup compared to the serial processing.

Benefits and Drawbacks. Although divide-and-conquer is a simple and straightforward approach, that allows to process the data resident in the fast GPU memory, it has several drawbacks.

First of all, the size of a chunk should be big enough to utilize GPU efficiently, but small enough to not cause a memory overflow. Selecting the right chunk size or adjusting the current size considering the amount of available GPU memory might be a difficult problem. Second, since the GPU works with a copy of the data rather than with the original data stored in the main memory, explicit synchronization is required when the data is changed.

Usage in DBMS. Ideally, the data should be split in a way that allows to utilize the GPU efficiently, without keeping the GPU idle or producing unnecessary intermediate results.

GPUQP [9] divides an operator into multiple independent suboperators, the amount of memory used by a suboperator executing on GPU is limited by the amount of available GPU memory. After the splitting, suboperators are executed either on CPU or on GPU, and the results are merged on the CPU.

In MultiQx-GPU [21] a table is partitioned into data chunks that are then passed from one operator to another. HippogriffDB [14] processes chunks asynchronously and, like MultiQx-GPU, passes them between operators.

Overlapping Transfers and Executions in CUDA and OpenCL

In CUDA, to overlap kernel executions with data transfers, several streams should be created using the `cudaStream_t` type. Then, the three following steps are performed iteratively, where the number of iterations equals the number of streams:

1. The `cudaMemcpyAsync` function is used to copy a data chunk to the device.
2. A kernel is launched.
3. The data is copied back to the host using the `cudaMemcpyAsync` function.

These steps can also be performed in three separate cycles, i.e. first all the N chunks are copied to the device, then the kernel is launched N times, and finally the chunks are copied back to the device. The performance of the two options might differ and depends on the utilized GPU. If the GPU has only one copy engine, the first option might have the same performance as the serial processing, because tasks are issued in the order, that does not allow to achieve any overlap. Two copy engines allow to perform host-to-device and device-to-host transfers at the same time, which leads to a high degree of overlap. For devices with compute capability 3.5 both options lead to the same performance[2].

The number of streams does not need to be too large. Shirabata et al. [19] use three CUDA streams in order to efficiently overlap data transfers to the device, kernel execution, and transfers back to the host. Wu et al. [22] report, that the best performance is achieved while using two streams, and increasing their number does not provide any additional gain in performance.

In OpenCL asynchronous processing is achieved by creating several command queues. The main steps, when two command queues are used, are as follows:

1. The first chunk is transferred to the GPU using the `clEnqueueWriteBuffer` function for the first command queue.
2. A kernel for the first chunk is launched.
3. The second chunk is sent to the GPU using the `clEnqueueWriteBuffer` function for the second command queue.
4. The kernel for the second chunk is launched.
5. The first chunk is copied back to the host using the `clEnqueueReadBuffer` function.
6. The second chunk is copied back to the host using the `clEnqueueReadBuffer` function.

3.2 Mapped Memory

By default, memory allocated on CPU is pageable, i.e., it can be swapped out to the disk. GPU cannot access the data directly from pageable host memory, which means that first a temporary page-locked (or pinned) host array must be allocated, and the data should be copied to this array. Then the data is transferred from the pinned array to the device memory.

主机上的内存被map到GPU后（pinned），kernel可以直接访问这部分内存。

When mapped memory is used, a portion of main memory is mapped onto GPU, and this memory is then declared as pinned (i.e. guaranteed to be at a certain location). A kernel then can directly access the pinned data in the main memory, however, data transfers are still performed implicitly and are automatically overlapped with kernel execution. The accessed data is transferred

[2] How to overlap data transfers in CUDA C/C++. Online at https://devblogs.nvidia.com/parallelforall/how-overlap-data-transfers-cuda-cc/.

pinned内存上的数据会直接放入thread local memory，不经过global memory。

directly to local memory of threads, as shown on Fig. 3, without using the GPU's global memory as an intermediate step.

By default, no guarantee of coherence is provided, e.g. the host can access the content of the pinned memory, while the device processes it. This can be beneficial when read-only operations are performed either by the CPU, the GPU or by both devices, because no device has to wait until the other device finishes the execution. However, it is necessary to ensure coherence when both the CPU and the GPU change the data stored in the pinned memory.

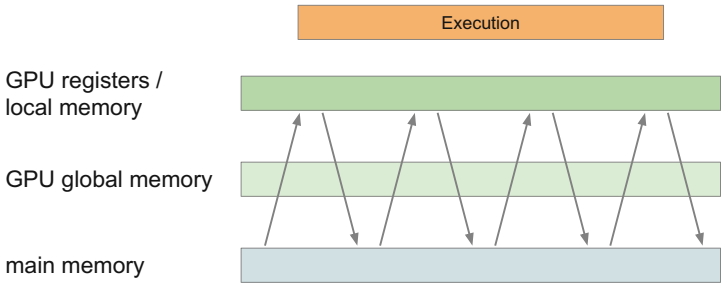

Fig. 3. Mapped memory; the data goes directly from the main memory to the GPU's local memory during the execution.

When a system receives a query, the kernel is simply run for all the data, as if it was small enough for the GPU memory. The result can either be transferred to a block of pinned memory or, if enough GPU memory is available, be stored directly on the GPU and transferred to the main memory, when the execution is finished.

Benefits and Drawbacks. Despite the benefits of mapped memory, its usage does not always speed up the execution. Allocation and deallocation of pinned memory is more expensive than simply copying the data, therefore usage of mapped memory might not be beneficial for a small amount of data. Additionally, transferring the data over the PCI-E bus takes more time than reading the data from the GPU's global memory. Therefore, usage of mapped memory is not recommended when host memory is accessed repeatedly [16].

One must note that allocating pinned memory reduces the amount of physical memory available to the system, which might have negative impact on the performance.

Mapped memory is used in the systems described in [3,23], and is reported [3] to be faster than the divide-and-conquer approach. Additionally, [23] states, that usage of mapped memory leads to higher transfer bandwidth compared to the usage of pageable memory, because data is directly transferred using GPU DMA engine without the overhead of being copied to a pinned buffer first.

适用场景：拥有大数据量(不需要重复拷贝)，其中少量部分才会用于计算(可减少访存延迟)

Considering the properties of mapped memory, one can conclude, that its usage is beneficial in cases, when the data is big enough, and a small portion of it is accessed by the GPU without a lot of repetitions. If a lot of computations need to be performed on the data, it is better to move it to the GPU memory.

Usage in DBMS. Due to the GPU processing style, the best performance can be achieved if the data in the main memory accessed by GPU is coalesced. To make writing the result of the execution back to mapped memory coalesced as well, Bakkum and Chakradhar [3] use a two-step procedure:

1. The `atomicAdd` operation (an atomic operation, that is ensured to write a value to the given location without any interference from other threads) is used to give each thread of a block an area in shared memory, where it will write the result. The number of rows that should be written to the result block is counted, and a block of global memory is allocated for the result of this thread block.
2. Each thread writes to the area assigned to it. The data is ensured to be written to global memory by using the `threadfence` function and counting the writes. When an area, allocated for a thread block, is filled with the output data, each thread copies the data from this area to the mapped memory.

Mapped Memory in CUDA and OpenCL

In CUDA, pageable memory is allocated using the `cudaMalloc` function, the data is then sent to the GPU using the `cudaMemcpy` function. A block of page-locked memory is mapped into the address space of the device by using the `cudaMallocHost` function. Since, in the case of mapped memory, there is no explicit data transfer between the host and the device, calling `cudaMemcpy` is not needed. The pointer to the block in the device memory can just be retrieved using `cudaHostGetDevicePointer` and then used to access the data from within a kernel.

In OpenCL, a block of pinned memory can be allocated by passing the flag `CL_MEM_ALLOC_HOST_PTR` to the function `clCreateBuffer` instead of the flag `CL_MEM_COPY_HOST_PTR` that is used for pageable memory. The functions `clEnqueueMapBuffer` and `clEnqueueUnmapMemObject` are used to map a region of the buffer object and unmap a previously mapped region correspondingly.

3.3 Unified Virtual Addressing

Unified Virtual Addressing (UVA) is supported by CUDA starting from the version 4.0 and requires a Fermi-class GPU with compute capability 2.0 or higher. This technique is used in Caldera [1] - a database engine for HTAP. UVA allows to have identical host and device pointers for pinned host memory (Fig. 4), so that the pointers can be accessed from the GPU no matter where the data really resides.

从一个设备拷贝到另一个设备不再需要从CPU中转。但CPU无法直接访问GPU中的数据。

The location of the data is determined from the value of a pointer, therefore explicitly requesting the device pointer with `cudaHostGetDevicePointer` is no longer necessary. Copies from one device to another can be performed without using the host (CPU) as an intermediate stage, data stored on one GPU can directly be accessed by a different GPU. The CPU, however, still can not access the data that resides on a GPU.

Benefits and Drawbacks. UVA, in general, has the same benefits and drawbacks as mapped memory. The speed of memory accesses in UVA depends on where the data resides: in the GPU or the main memory. Therefore, it would be beneficial to store the most frequently accessed data in the GPU memory.

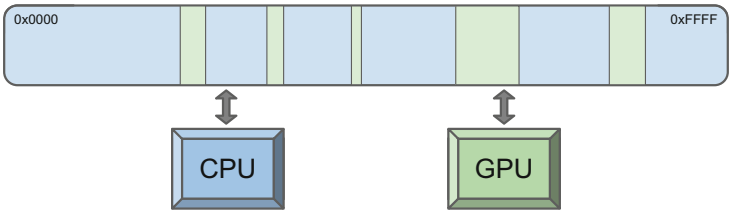

Fig. 4. Unified Virtual Addressing; the CPU and the GPU share the address space.

3.4 Unified Memory

CUDA 6.0 introduces Unified Memory (UM), which is supported starting with the Kepler GPU architecture and requires compute capability 3.0 or higher. UM further simplifies programming by automatically managing device memory allocations and data transfers. The concept of UM is shown in Fig. 5. UM allows to access the memory of both CPU and GPU using a single pointer and, unlike UVA, initially allocates managed memory on GPU and automatically migrates the allocated data between GPU and CPU. By default, the size of transferred pages are the same as the OS page size (4 KB) [16]. A programmer, however, is still able to manage memory allocations and data transfers explicitly.

To keep the data coherent, the host and the device are not allowed to operate on the same memory at the same time. As long as the GPU is executing a kernel, the CPU cannot access the managed memory, and an explicit call of any of the functions, that guarantee that the execution is finished (e.g. `cudaDeviceSynchronize`), is required.

Besides migrating the data, UM eliminates the necessity to create deep copies of structures, that contain pointers, before passing them to the GPU. Additionally, UM makes possible to share linked lists (lists, where each element contains a pointer to the next and/or previous element) between CPU and GPU, whereas, when pageable memory is used, passing a linked list to GPU requires complex operations.

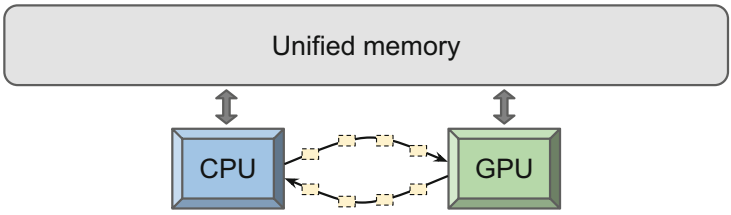

Fig. 5. Unified Memory; the data migrates to the device that accesses it.

Benefits and Drawbacks. A big disadvantage of UM used with CUDA 6.0 and Kepler architecture is that is does not allow to oversubscribe the GPU memory: the maximum amount of the memory, that can be allocated, is limited to the smallest of the available device memories. CUDA 8.0 and the Pascal architecture eliminates this limitation, allowing to use the entire system memory.

UM simplifies memory management and provides the benefit of UVA: one single pointer to the data. Memory accesses in UM are faster than in case of UVA or mapped memory, because the data resides on the device that processes it. However, overlapping transfers and executions has to be enabled by the programmer by using several streams.

Although UM makes programming easier, it does not always lead to a significant performance improvement [16], and, in some cases [13], is greatly outperformed by a non-UM approach, where the data is transferred to the GPU prior to the execution.

3.5 Other Solutions

Some systems use the fallback strategy: if the data is too big for the GPU, it is processed on the CPU instead. CoGaDB [5] stores table's columns on the GPU, but operators also require additional memory for their results and temporary data. In case of memory overflow, CoGaDB first removes cached data from GPU and then, if the available memory is still insufficient, aborts the operator and processes the data on CPU. Clearly, aborting an operator is expensive, and there are two alternatives: pre-allocate enough memory for the operator or wait until enough GPU memory is available. The first option, however, might allocate more memory that the operator actually needs, and therefore unnecessarily reduce the amount of the available GPU memory. The second option might involve significant waiting time, or the required amount of memory might never become available.

Some other systems [11,15] store only some data on the GPU, e.g. the most frequently or the most recently accessed columns. TripleID [6] minimizes the GPU memory usage by storing identifiers instead of elements themselves and avoiding storage of redundant data.

Another option is to compress the data [20]. Although even the compressed data might still be too large for the GPU memory, compression allows for faster data transfers [14].

4 Discussion

Table 1 contains a comparison of main characteristics of the divide-and-conquer approach, mapped memory, UVA and UM. When selecting a technique that would allow to achieve the best performance in a given case, it is important to consider the following:

Data Location: The main copy of the data can either reside in main memory, in the GPU memory, or be shared between both devices.

For the divide-and-conquer approach and mapped memory the data initially resides in main memory. In case of the divide-and-conquer approach it is fully transferred to the GPU for processing, while usage of mapped memory allows to transfer the data only when it is accessed. In UVA the data can be stored in both the main memory and the GPU memory, in UM the data is transferred to and becomes resident on the device that accesses it.

Asynchronous Processing: Data chunks can be processed either sequentially or asynchronously.

The divide-and-conquer approach allows to overlap data transfers and kernel executions, with mapped memory and UVA (when the data is in the main memory) it happens automatically. UM does not provide an overlap by default, but allows it to be implemented by a programmer.

Explicit Allocations and Transfers: Generally, only the divide-and-conquer approach requires explicit memory allocations and data transfers, e.g. the size of a data chunk should be defined and enough memory for it should be allocated on the GPU.

Synchronization: When the data is sent to the GPU for processing, and the GPU changes the data, it is necessary to also apply these changes to the data that resides in main memory.

Unlike other approaches, that work with only one copy of the data, the divide-and-conquer approach requires synchronization.

Explicit Coherence Maintenance: The situation, when the CPU and the GPU work with the same data at the same time should be avoided, because it might lead to one of the devices processing data that is no longer valid.

The divide-and-conquer approach sends a copy of the data to the GPU, thus this data cannot be changed by the CPU. UM blocks access to the data from the CPU, while a kernel is being executed. Mapped memory and UVA require the programmer to manually ensure the data coherence.

Unnecessary Data Transfers: Sometimes only a small part of a table is accessed by a kernel, which removes the need to transfer all the data to the GPU.

While with mapped memory, UVA and UM the accessed data is transferred on-demand, the divide-and-conquer approach might suffer from transferring a lot of data that is never accessed and therefore not needed.

Unified Address Space: UVA and UM allow the GPU to access the data using one single pointer independently of the data's physical location. The divide-and-conquer approach and mapped memory do not provide such benefit.

Speed of Memory Accesses: The location of the data determines how quickly this data can be accessed by a thread: when the data resides in the fast GPU memory, the speed of memory accesses is much higher, than if it needs to travel through PCI-E bus first.

The divide-and-conquer approach and UM use the fast GPU memory, since the data accessed by the GPU is located in the GPU memory. Mapped memory requires the data to be transferred over the PCI-E bus every time it is accessed, which makes the access speed slow. In UVA the speed depends on the physical location of the data.

Memory Oversubscription: Some approaches are able to automatically handle situations, when the allocated memory is too big, without causing memory overflow.

In case of mapped memory, no memory overflow can occur, since the data is transferred only when accessed by a thread. UM, starting from CUDA 8.0, allows to oversubscribe the GPU memory. Usage of the divide-and-conquer approach and UVA (for the data in the GPU memory) requires to be aware of the GPU memory size and adjust the data size accordingly.

Table 1. Comparison of the approaches

	Divide-and-conquer	Mapped memory	Unified Virtual Addressing	Unified Memory
Data location	Main memory	Main memory	Both	Migrating
Transfers and executions are overlapped	Yes	Yes	Yes	No
Requires explicit allocations and transfers	Yes	No	No	No
Requires synchronization	Yes	No	No	No
Requires coherence maintenance	No	Yes	Yes	No
Avoids unnecessary data transfers	No	Yes	Yes	Yes
Unified address space	No	No	Yes	Yes
Speed of memory accesses	Fast	Slow	Depends on data location	Fast
Allows for memory oversubscription	No	Yes	No	From CUDA 8.0

5 Conclusion

As one might see, no technique for GPU memory management is the best ultimately, and the choice of the right one for a particular application should be influenced by multiple factors, e.g. the pattern and the type of access to the data.

The divide-and-conquer approach is well suited for situations, when a table either is not changed by the GPU (OLAP case) or is handled by the GPU exclusively, else a synchronization mechanism would be necessary. However, it requires explicit memory management.

Mapped memory is good for situations, when the data is big, each data element is accessed only once and the accesses are coalesced. A lot of repeated and/or uncoalesced accesses lead to a performance drop. The same applies to UVA, when the GPU is working with data that is located in the main memory.

UM significantly simplifies the programming by removing the need to transfer the data and maintain coherence explicitly. From the simplicity perspective, UM surpasses other approaches. However, the performance might suffer in cases, where the same data is accessed often by both the CPU and the GPU.

To the best of our knowledge, existing DBMS use only one of the approaches. A mixture of several techniques (e.g. UM for frequently accessed and changed data and the divide-and-conquer approach for historical data) could be used to achieve the best possible performance. Therefore, there is a need for future research that would investigate the usage of a hybrid approach in GPU-accelerated DBMS.

Acknowledgment. This work was partially funded by the DFG (grant no.: SA 465/50-1).

References

1. Appuswamy, R., Karpathiotakis, M., Porobic, D., Ailamaki, A.: The case for heterogeneous HTAP. In: CIDR (2017)
2. Arefyeva, I., Broneske, D., Pinnecke, M., Bhatnagar, M., Saake, G.: Column vs. row stores for data manipulation in hardware oblivious CPU/GPU database systems. In: GvDB, pp. 24–29. CEUR-WS (2017)
3. Bakkum, P., Chakradhar, S.: Efficient data management for GPU databases. Technical report, High Performance Computing on Graphics Processing Units (2012)
4. Bakkum, P., Skadron, K.: Accelerating SQL database operations on a GPU with CUDA. In: GPGPU, pp. 94–103. ACM (2010)
5. Breß, S.: The design and implementation of CoGaDB: a column-oriented GPU-accelerated DBMS. Datenbank-Spektrum **14**(3), 199–209 (2014)
6. Chantrapornchai, C., Choksuchat, C., Haidl, M., Gorlatch, S.: TripleID: a low-overhead representation and querying using GPU for large RDFs. In: Kozielski, S., Mrozek, D., Kasprowski, P., Małysiak-Mrozek, B., Kostrzewa, D. (eds.) BDAS 2015-2016. CCIS, vol. 613, pp. 400–415. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-34099-9_31
7. DeWitt, D.J., Katz, R.H., Olken, F., Shapiro, L.D., Stonebraker, M.R., Wood, D.A.: Implementation techniques for main memory database systems. In: SIGMOD, vol. 14, pp. 1–8. ACM (1984)
8. Gregg, C., Hazelwood, K.: Where is the data? Why you cannot debate CPU vs. GPU performance without the answer. In: ISPASS, pp. 134–144. IEEE (2011)
9. He, B., et al.: Relational query coprocessing on graphics processors. TODS **34**(4), 21 (2009)

10. He, B., Yu, J.X.: High-throughput transaction executions on graphics processors. VLDB **4**(5), 314–325 (2011)
11. Heimel, M., Saecker, M., Pirk, H., Manegold, S., Markl, V.: Hardware-oblivious parallelism for in-memory column-stores. VLDB **6**(9), 709–720 (2013)
12. Kim, Y., Lee, J., Jo, J.E., Kim, J.: GPUdmm: a high-performance and memory-oblivious GPU architecture using dynamic memory management. In: HPCA, pp. 546–557. IEEE (2014)
13. Landaverde, R., Zhang, T., Coskun, A.K., Herbordt, M.: An investigation of unified memory access performance in CUDA. In: HPEC. pp. 1–6. IEEE (2014)
14. Li, J., Tseng, H.W., Lin, C., Papakonstantinou, Y., Swanson, S.: HippogriffDB: balancing I/O and GPU bandwidth in big data analytics. Proc. VLDB Endow. **9**(14), 1647–1658 (2016)
15. Mostak, T.: An overview of MapD (massively parallel database). Technical report, MIT (2013)
16. Negrut, D., Serban, R., Li, A., Seidl, A.: Unified memory in CUDA 6: a brief overview and related data access. Technical report, TR-2014-09, University of Wisconsin-Madison (2014)
17. Pinnecke, M., Broneske, D., Durand, G.C., Saake, G.: Are databases fit for hybrid workloads on GPUs? A storage engine's perspective. In: ICDE, pp. 1599–1606. IEEE (2017)
18. Pirk, H., Manegold, S., Kersten, M.: Waste not... efficient co-processing of relational data. In: ICDE, pp. 508–519. IEEE (2014)
19. Shirahata, K., Sato, H., Matsuoka, S.: Out-of-core GPU memory management for MapReduce-based large-scale graph processing. In: CLUSTER, pp. 221–229. IEEE (2014)
20. Sitaridi, E.: GPU-acceleration of in-memory data analytics. Ph.D. thesis, Columbia University (2016)
21. Wang, K., et al.: Concurrent analytical query processing with GPUs. Proc. VLDB Endow. **7**(11), 1011–1022 (2014)
22. Wu, R., Zhang, B., Hsu, M.: GPU-accelerated large scale analytics. Technical report, HPL- 2009–38, HP Laboratories (2009)
23. Yuan, Y., Lee, R., Zhang, X.: The Yin and Yang of processing data warehousing queries on GPU devices. VLDB **6**(10), 817–828 (2013)