University of Magdeburg

School of Computer Science

Master's Thesis

Processing OLTP Workloads on Hybrid CPU/GPU Systems

Author:

Mudit Bhatnagar

November 04, 2016

Advisors:

Prof. Dr. rer. nat. habil. Gunter Saake

M.Sc. David Broneske

Department of Computer Science

Bhatnagar, Mudit:
Processing OLTP Workloads on Hybrid CPU/GPU Systems
Master's Thesis, University of Magdeburg, 2016.

Abstract

In recent times there have been a plethora of researches done on the utilization of co-processors like GPU and FPGA in database management system (DBMS). The reason for this trend is that modern processors have reached a performance threshold. Two major factors that have led to this behaviour are Memory Wall and Power Wall. This has forced hardware vendors to come up with specialized processors that focus on speeding up computation in specialized areas. Hence, we are moving towards the age of heterogeneous computing where an efficient co-processor can be used along with the traditional CPU to meet our performance requirements.

Various researches in the recent past have shown that database systems can effectively use specialized processors, especially GPU to speed up query processing. This use of GPUs for acceleration of traditional computing system is called GPGPU (General Purpose GPU) based computing. To this end, we have been able to use GPUs as an effective coprocessor in OLAP scenarios with increased performance. The support for OLTP scenario is still under research with DBMS like GPUTx that executes bulk OLTP transactions as a single task on GPUs only system.

In our work we are going to study the processing capabilities of heterogeneous (CPU-GPU) systems in an OLTP scenario. For this we will implement the TPC-C Benchmark in a bulk query execution model. Our work will also compare the Row Store and Column Store based storage models on TPC-C Database and find out the most efficient storage mechanism for query execution in a CPU/GPU based heterogeneous system.

Acknowledgement

This thesis marks the end of my two years bonding with Otto-von-Guericke University Magdeburg for graduating with a Master degree. At this moment, I would like to express my gratitude to all the people who made it possible.

First and foremost, I would like to thank Professor Gunter Saake for giving me an opportunity to undertake this thesis work in his department. I would also like to thank for all his encouraging lectures on database topics as part of my degree that really motivated me to approach this work.

I would like to express my sincere thanks to David Broneske for being a big support throughout my research work. He has always been available for any help and gave his valuable inputs whenever needed. It is only because of his support that I am able to deliver my thesis in due time. I would also like to extend my sincere gratitude to Dr.-Ing. Martin Schäler for being the second reviewer of my thesis and providing his valuable feedback.

Lastly I would like to thank Sebastian Breß for his guidance at the start of the thesis. His insights at the topic paved the way for choosing the right set of ideas for the work.

Contents

List of Figures

List of Tables

List of Code Listings

1. Introduction

The ever growing OLTP market through industries like banking, credit card, and online retail have led to a significant increase in the amount of transactions. Also, the advent of Web 2.0 based web technologies have further added to this number. Most OLTP based application require the system to perform tens of thousands of transactions within a short period of time which leads to the requirement of high throughput oriented systems [HY11]. With the limitation in scaling the processing power of modern processors, it has become important to use new age hardware like Graphical processing Units(GPUs) to fully leverage the computing power at hand so that we can cater to the ever growing need of the OLTP market [BBHS14].

It is evident from recent researches that GPU can serve as an efficient co-processor for OLAP based scenario. The significance of using GPU as a co-processor for an OLTP based scenario is still being investigated. This can be attributed to the nature of OLTP workload which is believed to be non-suitable for GPU style of processing.

GPUs are more suitable for SIMD (Single Instruction Multiple Data) style execution model where a single operation is performed on a bulk of data. This style of processing greatly harness the architectural benefits of GPUs. The three major problem with the OLTP workload that makes it non fit for GPU based acceleration are:

a. OLTP systems need to handle many small transactions with various read and update operation on the database. In a multi user scenario, the transactions must adhere to isolation and consistency for correctness when they perform concurrent updates to the database. The massive thread parallelism of the GPU comes with various technical challenges related to the correctness and efficiency of transaction executions [KHL15][HY11].

b. OLTP queries are throughput oriented which requires executing tens of thousands of transactions in a short time, instead of response time oriented as in case of OLAP [CD97]. High throughput requires multiple tasks to be executed in a small time duration. Hence,

a system performing multiple tasks in parallel can give high throughput. GPUs are not preferred for task parallelism, instead it is considered a good fit for data parallelism which makes it unsuitable for executing OLTP workload.

c. The OLTP query processing engine mostly depends on the ad-hoc execution model (due to user interaction) which poses a serious challenge for parallel GPU architecture which can be exploited for parallelism only for bulk execution model otherwise the GPU can not be utilized to its full potential [KHL15].

With the ever increasing OLTP workload it has become possible to bulk a large number of transaction and execute it as a single unit, this has set ground for some of the recent works like HSTORE [KKN+08] and GPUTx [HY11]. HSTORE is an OLTP based query processing engine which focuses on high throughput. The important aspect of HSTORE is that the complete workload is required to be specified in advanced as it assumes that there will be no ad-hoc query. This leads to the removal of stalls due to user interaction [SMA+07]. GPUTx further uses this bulk query processing model and tries to execute bulk queries on a GPU based processor [KHL15].

Although GPUTx showed a significant improvement in throughput and achieved 4-10 times better performance than a CPU based query processing engine, there are still some limitations in the approach. GPUTx is a query processing engine aimed at using GPUs as an efficient query processor not as an efficient co-processor. Also the implementation of a task or stored procedure as a kernel makes GPUTx tightly coupled to the database schema and transactions. To use GPUTx with another set of transactions and schema, the kernels will have to be reimplemented.

Another important aspect for a database engine is the storage model. It has been proved that a column store model performs better on a CPU/GPU based in-memory database for an OLAP scenario [Gho12]. There is, to the best of our knowledge, no research that compares the Row Store and Column Store model for In Memory based OLTP engine using CPU/GPU based co-processor. Hence, through this research we will also do a comparative study between the two storage models and try to find out the most efficient storage model.

Hence broadly speaking below are the research questions that we will answer through our work:

- ***RQ1: Which is the most efficient storage mechanism for OLTP query processing on a hybrid CPU/GPU System?***
- ***RQ2: Is OLTP query processing on CPU/GPU based query processing engine faster than traditional CPU only system?***

To answer our research questions, we have implemented an in-memory database for TPC-C benchmark. The same TPC-C schema is implemented for both Row Store and Column Store storage scheme. We have also implemented various OLTP operators needed for

the TPC-C workload.Four out of five standard transactions of TPC-C workload have been executed using a bulk execution model. All the operators will be placed in their corresponding device i.e. CPU or GPU. Finally we compared the execution time of CPU/GPU based query processing vs CPU only query processing. Further we also compare the execution time of Row Store and Column Store implementation to find a better performing model.

Hence, our work will make the following contributions :

- Implementation of TPC-C benchmark in Column Store and Row Store
- Implementation of reusable set of OLTP operators as OpenCL kernels.
- Implementation of TPC-C transactions using a static query plan (No dynamic transaction management).
- Comparative study of storage mechanisms in a hybrid CPU/GPU based OLTP system.
- Comparative study between hybrid CPU/GPU based system and CPU only system for an OLTP workload.

The initials results showed that Column Store storage mechanism can speed up the execution of an OLTP based system by a factor a 4x. We also identified some of the benefits of GPU based computing that can be harnessed efficiently using a Column Store implementation. Our evaluation also suggested that a hybrid CPU/GPU based systems can speed up the execution time of an OLTP workload by a factor 2x. Our results also suggested that given a good load balancing mechanism the hybrid system's performance can be further improved.

Structure of the Thesis

Apart from this chapter, this thesis comprises seven more chapters. Chapter 2 presents background information about GPU based computing and will introduce all the aspects of GPU based query processing. This chapter will also give a brief description about TPC-C database and its transaction that we have used for evaluation of our work.

Chapter 3 will introduce some state of the art GPU based databases and present some related works in the field of GPU based OLTP systems. Chapter 4 presents some of the assumptions that have been used in this work. Chapter 5 presents the approach that has been used for implementation of the TPC-C database and will also show the methods used for implementing the transactions in a typical TPC-C workload. The evaluation done for this work has been covered in Chapter 6. This chapter will give the complete information about the evaluation set-up and the results of our work. This chapter will show the comparative study of OLTP workload performance on Row Store vs Column Store and CPU only vs Hybrid CPU/GPU based systems. This chapter

also summarizes the complete evaluation phase and will draw a conclusion from our evaluation. In this chapter we will try to answer the research question that we had put forward for this work.

Chapter7 will give a brief summary of our work and will draw the conclusion of this work. Finally, Chapter 8 will present the limitation of our work and scope for improvements. In this chapter we will also present some of the open areas that needs future research to come up with a fully functional hybrid CPU/GPU based OLTP system.

2. Background

This chapter provides background information needed to understand the basics of GPGPU computing and GDBMS (GPU based DBMSs). This chapter starts by introducing the basic architecture of GPUs. It will also present some of the challenges of GPU based computing. Further this chapter will talk about transaction processing on a GDBMS and introduce important aspects like operator placement and execution model. Finally, this chapter will introduce the programming model used for GPU based computing. Towards the end, this chapter will give information about the TPC-C benchmark that has been used in this research for testing OLTP workload.

2.1 GPU and its architecture

Graphical Processing Units are specially designed processors that were traditionally designed for gaming application. Recent works have shows that GPUs can significantly increase the processing power of general computing problems [HYF+08]. The use of GPUs in arbitrary workloads and problems instead of graphical processing only is referred as GPGPU (General-purpose computing on graphics processing units).

Figure 2.1 on the following page gives a high level comparison between CPU and GPU architecture. The ALU (Arithmetic Logical Unit) is responsible for computing tasks, hence it performs all the logical and arithmetic operation. The Control unit handles synchronization and is also responsible to direct the system for instruction execution. Cache keeps the frequently accessed data to save memory access time.

Architecturally, the CPU is composed of just few cores i.e. ALUs with a common Control unit. Also a big part of CPU transistor is used for control unit, this allows multiple cores to interact, perform out of order execution and have better flow control. A major part of the CPU chip is used for caches which gives it low memory latency as more data can be cached. All the thread in the CPU are heavy-weight and self-sufficient and can perform a task individually, this allows them to perform multiple tasks in parallel.

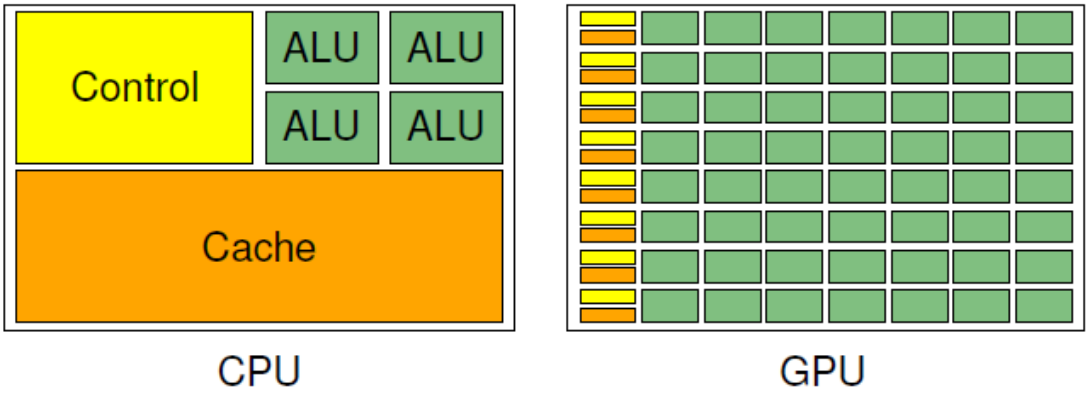

Figure 2.1: GPU Architecture

On the contrary the GPU contains 10,000s of cores which are scheduled in batches or thread blocks. Each thread block shares a common control unit and the memory cache. Hence, within a thread block, the threads can cooperate via shared memory and synchronize their execution. All the threads within the thread block execute same instruction in parallel, hence they execute the same operation for different set of data, which is referred as Single Program Multiple Data (SPMD) style of execution. Overall, we can see from the architecture thata major part of GPU chip space is used for ALUs and, thus, dedicated to computation which will lead to better throughput. In contrast, the cache size of GPUs are 10x smaller than CPU counterparts which will increase the memory latency [HYF+08].

Given the architecture we can say that, algorithms well-suited to GPGPU implementation are those that exhibit two properties: they are data parallel and data intensive. Data parallel means that a processor can execute the operation on different data elements simultaneously. Data intensive means that the algorithm is going to process lots of data elements, so there will be plenty to operate on in parallel. Due to these properties, GPUs can achieve high performance by using a lot of small processing units that can operate on different data elements in parallel [Cen].

However, individual processing units in a GPU cannot beat a CPU for general purpose performance. The CPUs have simpler architecture and has optimization techniques like long pipelines, out-of-order execution, branch prediction and instruction-level-parallelism.

2.2 GPU as a co-processor

Given the limitation in scaling the processing power of the CPUs we have to find alternative solutions like using specialized processors like GPU to aid CPU in its computing task. A GPU can be used as an efficient coprocessor in tandem with a CPU to increase the performance of various scientific and engineering problems([Bre15]) that can leverage the data parallel execution model of GPUs.

A CPU can offload various compute-intensive tasks to GPU and use it to speed up the execution. In the meanwhile the CPU can also execute tasks by itself. This style of

execution not only uses the computing power of GPU to speed up data parallel tasks but also gives us inter device parallelism between CPU and GPU. For end user the application will run faster. This style of computing is termed as Hybrid computing [Bre15].

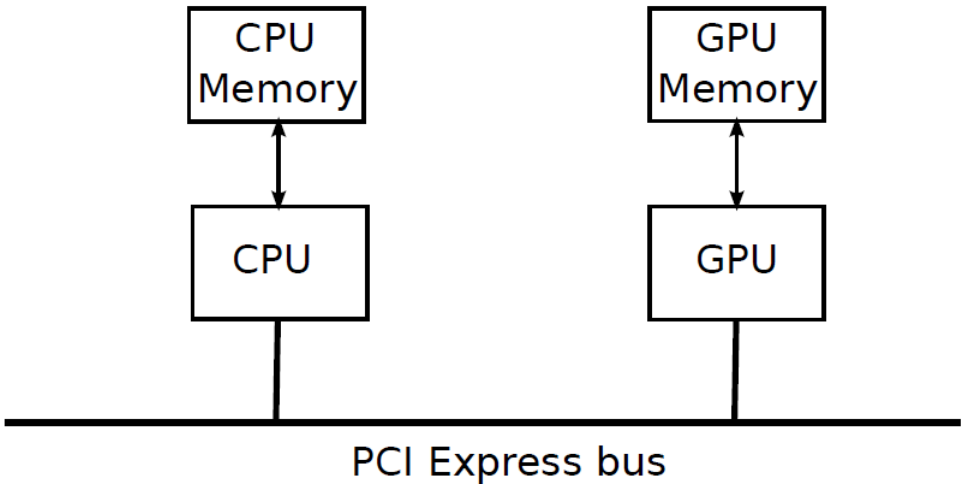

Figure 2.2: GPU Connectivity

Figure 2.2 illustrates the connectivity of GPU with the CPU. GPUs are a separate device which is connected to the CPU via a high speed PCIe express bus. All the data that needs to be executed on the GPU needs to be transferred from CPU memory over a PCIe bus to the GPU memory. After the processing the resulted data will then be transferred back to the CPU over the same PCIe bus.

2.3 Challenges in GPU computing

A GPU is architecturally a very different device as compared to the CPU. Also, unlike CPU it is an external device connected via a PCIe bus to a host device. Traditional computing techniques which gave good results cannot fare well on a GPU as they are not designed to leverage the architectural benefits of a GPU. Being a separate device and having architectural changes, GPU computing faces various challenges:

a. Communication Bottleneck: As GPUs are separate device with their own memory, all the data that needs to be processed on GPU has to be transferred over the PCIe bus. Also the processed data needs to be transferred back to the host [BGW+08]. This transfer overhead introduces high latency as the GPU remains idle during this process. This problem becomes more severe with data-intensive problems like query processing [MBS15].

b. Small Memory Size: The traditional GPUs like NVIDIA Titan X have 12 GB of device memory which is relatively small for Query Processing. The small memory capacities on GPU limits the amount of data that can be processes [BGW+08]. Due to

this limitation the data needs to be transferred to GPU in multiple chunks for subsequent processing which will have negative impact on the processing speed.

c. Bandwidth Bottleneck: The bandwidth of the PCIe bus is less as compared to the bandwidth of the GPU. This leads to a latency. The currently available PCIe 3.0 has a memory bandwidth of 16 GB/s, whereas the high-end GPUs like Titan X have bandwidth of 480 GB/s which is more than twenty times that that of the PCIe 3.0. Due to this limitation the PCIe bus is note able to feed GPU with sufficient amount of data and a major processing power of GPU goes in vain. This can affect the performance of GPU based task [BGW+08].

d. I/O Bound vs Compute Bound Problems: Due to the Communication bottleneck and Bandwidth bottleneck not all algorithms in computing can benefit from GPU acceleration. An I/O bound algorithm will have multiple stalls due to I/O device interaction, hence CPU will not be able to delegate the task to GPU and would require multiple data and instruction transfers over the PCIe bus. A compute intensive problem that can be parallelized to leverage GPU's architecture fares well on it as the GPU can be utilized for most of the time in computation rather than waiting for data and instructions.

e. Use of Specialized API: A traditional CPU based system cannot utilize the computing power of GPU by simple hardware extension. An application can only use the co-processing capability of hybrid systems when it is specifically programmed for GPU computing using a programming APIs, such as OpenCL or CUDA.

In a nutshell, to do efficient query processing on GPUs we need to handle the limitations like small device memory, communication bottleneck and bandwidth bottleneck. These limitations are mostly related to architecture. Hence we cannot remove them, but they can be hidden. The PCIe bottleneck and communication bottleneck can be hidden by providing sufficient tasks to GPU so that the majority of processing time is spent on computing. To hide the limitations due to local GPU memory we need to make sure that the maximum amount of data that can fit into GPU memory is sent in a single transfer and multiple communication between host and GPU is avoided. The algorithms like block nested join works efficiently with chunks of data and has shown performance gain on GPU.

2.4 Execution Model

The execution models which are mostly used in parallel computing are task parallelism and data parallelism.

2.4.1 Data Parallelism

A data parallel method is performed by distributing the data amongst computing units. In a DBMS a huge operation can be broken down into multiple small operations and performed in parallel using data parallelism. The same instruction is executed on

multiple data items in parallel. The framework that is used to realize this kind of parallelism is "Single Instruction, Multiple Data" (SIMD). This execution model is used in GPGPU computing, as a single instruction in a thread block is executed on multiple threads with different units of data. Given the small amount of space devoted for control logic, we cannot give each ALU a different instruction, but need to synchronize the execution of all ALUs. Due to the limited space, we only have one instruction decoder per Warp. This leads to a SIMD execution style, as a single instruction is performed on multiple chunks of data in parallel.

2.4.2 Task Parallelism

A task parallel approach is used to solve computation problems where multiple tasks can be performed in parallel. Each task in the computation problem is scheduled to respective cores for execution. Given the architecture of CPU where a lot of chip space is devoted to control unit, it is designed to perform different set of instructions on different cores [fc]. This makes CPU a good device for task parallel operations.

2.5 GPU Memory Model

The Memory model of GPU is different compared to the memory model of a CPU. Figure 2.3 on the following page shows the memory model of the GPU. We can see from the figure that there is a global memory which is shared amongst multiple work groups. Whenever a thread block (or work group) needs to access the Global memory, it needs to copy the memory into its shared memory for execution. Each thread block has a shared memory or local memory which is divided into memory banks. Each thread in a thread block can access the shared memory bank. The condition when two or more threads try to access the same shared memory bank is referred as bank conflict. Each thread in a work group has an exclusive memory which is called registers, hence each thread can access only its own register memory.

In a Nutshell we can say that:

- Each work item or thread has private memory named registers.
- Work items are grouped into a work group or thread blocks. Each work group has its own shared memory.
- Global memory is shared across all work groups

2.6 Coalesced Memory Access

As mentioned in GPU memory model, a GPU has an internal memory hierarchy that is used for efficient access of data. To make the most out of the available hardware the GPU work group should fetch data from the global memory using a coalesced memory

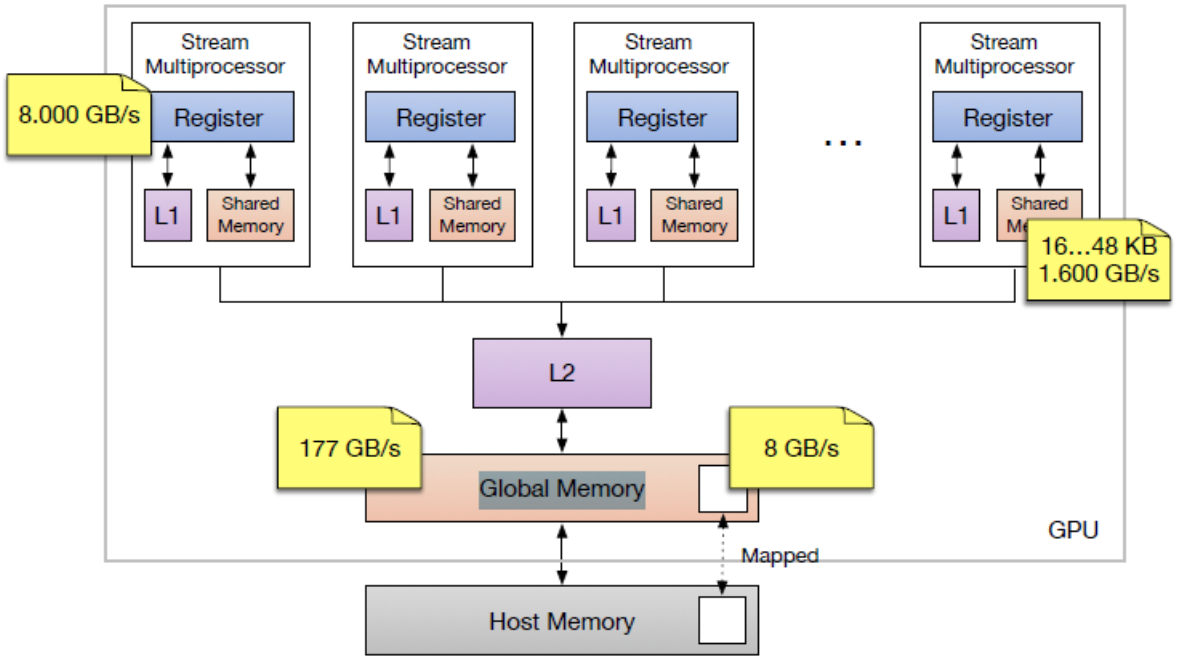

Figure 2.3: GPU Memory Model[SB]

access pattern. Whenever a work group fetches memory from the global memory a minimum number of elements are fetched together, hence the work group can work on this pre-fetched memory block using fast shared memory. To achieve this optimized execution behaviour of the GPU each thread within a work group should access sequential blocks of memory , this phenomenon where each work item within a work group accesses a sequential block of memory is termed as Coalesced Memory Access.

Listing 2.1 shows a kernel program to increment the value of each item in the int array by 10. This kernel will lead to coalesced memory access. This kernel increments the value of an int array which has been sent as an input from the host program. As we can see from the code listing that each ith work item is accessing the i-th index of the array.

Listing 2.2 on the facing page gives the example of a kernel with non coalesced memory access, this kernel performs the increment of every alternate element of an integer array. As we can see from the kernel half of the memory loaded into the local memory of work group goes waste due to alternate access pattern used.

Listing 2.1: Example Kernel: Coalesced Memory Access

```
1 __kernel void coalesced(__global int* arr)
2 {
3     int i = get_global_id (0);
4     arr[i]+=10;
5
6 }
```

Listing 2.2: Example Kernel: Coalesced Memory Access

```
__kernel void non_coalesced(__global int* arr)
{
    int i = get_global_id (0);
    arr[2*i]+=10;

}
```

2.7 OLTP vs OLAP

In this section, we differentiate between the OLTP and OLAP workloads. Also, this section shows the potentials of GPU acceleration on these workloads along with the challenges that GPU acceleration faces due to the nature of these workloads.

2.7.1 OLAP (On-line Analytical Processing)

OLAP mostly deals with historical or archival data. A typical OLAP workload has few number of transactions that are performed on a big chunks of data. Queries in an OLAP workload are complex and requires aggregations. For OLAP systems response time is an effectiveness measure of performance. OLAP applications are widely used for data mining techniques. A typical OLAP workload includes business reporting for sales, marketing, management reporting, business process management (BPM), budgeting and forecasting, financial reporting and similar areas [OvO]

2.7.2 GPU Acceleration in an OLAP Workload

Given the nature of the OLAP workload which require performing a limited number of instructions on a bulk of data, it becomes easy to exploit data parallelism. The data can be broken down to multiple chunks and the same instructions are applied on each chunk in parallel, this can be referred as performing a single instruction on multiple data or SIMD style of processing. Hence an OLAP Scenario fits perfectly to the GPU style of processing. Given the GPU architecture which has less chip space for control logic and more space for execution units like ALU, a small number of instructions can be performed on large number of data in parallel. The developer can schedule a heavy data-parallel task to the GPU, which leads to higher parallelism then its CPU counterpart.

2.7.3 OLTP (On-line Transaction Processing)

The OLTP workload is characterized by a large number of short on-line transactions (INSERT, UPDATE, and DELETE). In OLTP systems fast query processing and data integrity in multi user environment is of prime importance. The effectiveness measured performance is the number of transactions executed per second or throughput. A typical OLTP database consists of detailed and current data. The database tables used to store transactional databases are stored in 3NF form. An OLTP system should handle multiple users concurrently with low latency. It is interactive in nature, meaning that

latency impacts user experience. A typical example of an OLTP system is an ATM machine where multiple users issue transactions in parallel through multiple machines on the same database of the bank [OvO].

2.7.4 GPU Acceleration in an OLTP Workload

Due to ad-hoc queries, the traditional OLTP workload does not fit for GPU acceleration as we do not have bulk data that can be chunked and executed on different cores. However, an OLTP environment where the workload is pre-determined we can perform a bulk execution model that can leverage the GPU architecture just like in an OLAP scenario. The realization of ACID properties remains a challenge in OLTP workloads due to frequent update operations, this problem can be handled by efficient scheduling and locking mechanism.

2.8 Transaction Management on GDBMS

The transaction management on a GDBMS is completely different as compared to traditional DBMSs. All the thread blocks in a GPU work independently of each other with no inter-kernel communication. Since GPU is a separate device from CPU it is hard to realize a centralized locking mechanism that maintain consistent query processing between CPU and GPU. Also a lock-based transaction processing significantly breaks the performance of a GDBMS. Hence a lock free transaction processing is needed to fully utilize the potential of a GDBMS.

The approach used in GDB is widely used for lock free transaction management on GDBMS are based on finding the transactional dependency. In this approach a transaction is evaluated to find all conflicting operations which cannot be performed concurrently. All the non-conflicting operations can be scheduled in parallel whereas the conflicting operations needs to be scheduled serially to maintain data consistency. This can be achieved by having a helping data structure that stores the dependency between the transactions and using it for generating a query plan that avoids conflicts. A typical example of such data structure is a T-Graph.

2.9 Operator Placement

To efficiently utilize heterogeneous devices in a hybrid system a co-processor based DBMS should be able to place the operator on the device which is best suited for its performance. The method of placing operator on a suitable device during query execution by the database system is referred as operator placement. There are various operator placement schemes that have been used with varied results, these can be broadly divided into the types *compile-time operator placement* and *run-time operator placement*

2.9.1 Compile Time Operator Placement

This approach places the operator before the actual execution of the query is done. This operator placement scheme is data driven as the operator placement is decided based on the availability of the data in GPU's memory. Statistical measures are used to find the most accessed or the frequently accessed data in the system, this data is then loaded into the GPU memory given the condition that it fits in it. Based on the availability of the data on the GPU, the operator that needs to be performed on this data is placed. A background job is used to identify the access pattern of the workload and identify the frequently accessed data and place it in GPU memory. In case of the unavailability of the data in GPU, the corresponding operation is performed on the CPU as a fall-back option this leads to serious degradation due to the increased overhead of transferring data and operator to the GPU. The drawbacks of this strategy are:

- This approach can execute the complete operator chain only if all the data needed in the operator chain are available on GPU. Given the small size of GPU memory, this becomes a serious limitation as there is every possibility of complete input data for the operator chain not fitting in the memory. To overcome this problem only those inputs are processed on the GPU which can fit in its memory else the operation is performed on the CPU.

- The increased number of operators in this approach leads to performance degradation. During the execution of multiple operators there can be a case that their collective memory footprint exceeds that of GPU memory. In such a scenario the system goes to the fall-back scenario where the complete operator is restarted on CPU which leads to the wastage of processing time due to transfer of an unsuccessful operator and respective data.

- This approach tightly couples the query execution engine with the environment variables such as current load and memory usage. It is a challenging task to estimate these parameters before the actual execution of the workload.

2.9.2 Run Time Operator Placement

The run-time operator-placement strategy places operators at the run-time of the workload. This placement scheme do not require information of the workload as the operator placement is performed after all input data is available [BFT16]. A learning based cost model is constructed to estimate the execution time of a particular operator on the processing devices. The model observes the cost of executing an operator on input data for different processors and learns the correlation. After training with sufficient data sets and operators the models learns the optimal device for the operator [BS13]. Due to the dynamic nature of this approach it has various benefits over the compile time approach:

- As the operator placement scheme is fully aware of the input data, it can dynamically react to the shortcoming of the compile time approach like memory overflow for input data [BFT16]
- The transfer of the data and operator to the GPU is done at run-time unlike the compile time approach where data was pre-loaded. In case a leaf operator in an operator tree is aborted due to high memory footprint then we can stop scheduling other operators from the operator tree and save our self from transfer overhead of sending data for aborted transactions [BFT16].
- Unlike the compile time approach, this approach has low dependency on the knowledge of the environment variable as the cost model can be retained and used with a new workload.
- Using approach like query chopping along with the run-time operator placement can limit the number of executions running in parallel hence we can significantly reduce the problem of aborting operators due to memory overflow [BFT16].

2.10 Programming Model

A kernel programming model is the standard programming model used for GPU programming by many programming frameworks, such as OpenCL and CUDA [HSP+13, Bro15]. A kernel is a program which is executed on a single unit of data. A GPU device executes multiple instances of this kernel programs on different cores, each core executes the same kernel program for different unit of data. The CPU works as a scheduler of these kernels and is also referred to as a host. The primary function of the host program is to schedule tasks to the connected devices like GPUs through kernel invocation. The host also transfers raw unprocessed data over the PCIe bus to GPU's memory, finally when the data is processed by the GPU it is again transferred back to the host memory over the PCIe bus [HSP+13].

Figure 2.4 on the next page shows how the CPU (host) communicates with the GPU for executing kernels. Firstly the CPU (host) sends the data to be processed on the GPU over PCIe bus. After transferring data, the host invokes the kernel to be executed over the data. After the invocation of the kernel, the host has no control over the execution. Also a kernel program cannot interact with another kernel program in the course of their execution. The host is informed when the data processing is finished on the GPU and the final processed data is transferred back to the CPU (host) over the PCIe bus.

The APIs currently used in the market for GPGPU programming are CUDA and OpenCL:

CUDA (Compute Unified Device Architecture) was launched for the first time in 2006 by NVIDIA. It is a general purpose parallel programming API that uses the parallel architecture of NVIDIA GPUs to solve computation problems more efficiently than a CPU does.

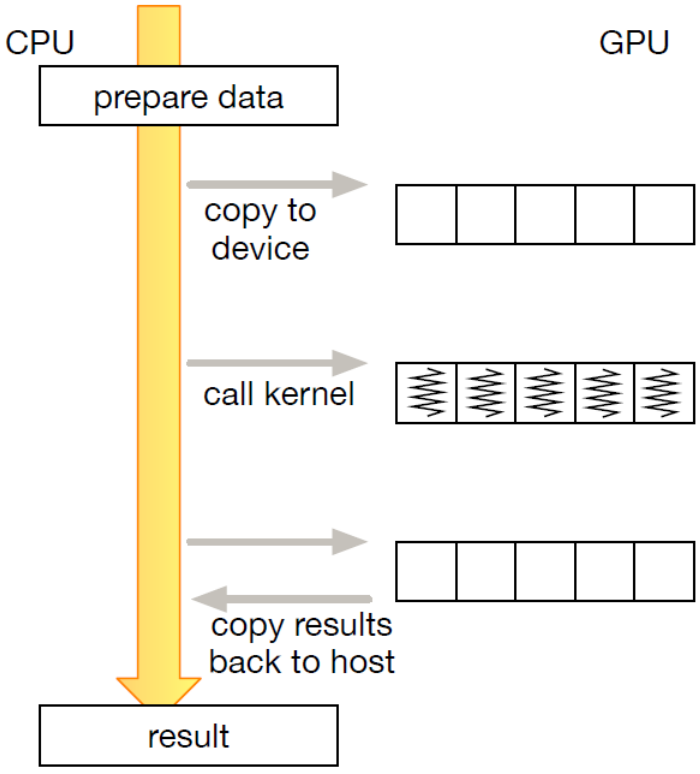

Figure 2.4: Execution Overview [SB]

OpenCL OpenCL (Open Computing Language) is the first open, standard for general-purpose parallel programming of heterogeneous systems. It tries to provide a unique programming environment for software developers to write portable code for servers, laptops, desktop computer systems and handheld devices using both multi-core CPUs and GPUs [OAD14]

In this work we have used OpenCL programming API for the following reasons:

- It perfectly fits our need to schedule kernels on multiple devices to have not only inter-device parallelism but intra-device parallelism.
- CUDA API is designed only for NVIDIA GPUs whereas OpenCL can be used with various hardware vendors which makes an OpenCL solution more device-independent and cross-platform.
- One major advantage of OpenCL over CUDA is that a single host application of OpenCL can host multiple kernels on multiple devices [Sca].

3. Related Work

This chapter presents some of the work that has been done in the field of GDBMSs. This chapter starts by introducing some of the state of the art databases which use GPU acceleration for both OLAP and OLTP Scenario. Further this chapter will present a research work which is related to some of the important aspects of GDBMS like operator placement and Transaction Management.

3.1 GPU Accelerated Systems for OLAP

He et al. presented the first hybrid CPU-GPU based In Memory database named GDB. They used a Selinger-style optimizer to create a heterogeneous query plan and used a learning based model for data placement and operator placement. They used split and sort primitives to execute some of the common relational operators using data-parallel approach. Their research showed that the performance of the query execution for OLAP based queries can be increased by 2-27 times despite some of the limitation of GPUs like transfer bottleneck [HLY+09].

To further improve the performance of hybrid query processing an efficient operator placement scheme needed to be devised. Breß et al. came up with HyPE which is a hybrid query plan generator that uses a learning based approach to generate an efficient query plan to fully utilize the processing power of CPU and GPU. The basic objective of HyPE is to find the right device for operator placement and generation of efficient query execution plan [BS13].

Breß et al. developed CoGaDB, a main-memory column store DBMS with built-in GPU acceleration for OLAP workloads. The operator placement in CoGaDB is done by using HyPE. CoGaDB showed that it could quickly adapt to the hardware and generate efficient query plans to be executed in a hybrid scenario [SMA+07].

Ocelot is a hardware-oblivious data processing engine that extends MonetDB. Breß et al. extended Ocelot by using HyPE for operator placement decisions, resulting in a hardware

adaptive database using heterogeneous processors [BKH+14]. The combination of HyPE and Ocelot proved to be an efficient query processing engine on different architectures that can learn device specific cost of the operator and make an efficient operator placement scheme.

3.2 GPU Accelerated Systems for OLTP

All the researches discussed so far focused on GPU acceleration for an OLAP environment. He et al. further tried to investigate the processing power of GPUs for an OLTP workload and developed GPUTx. GPUTx used bulk execution model to group multiple transactions into a bulk and to execute the bulk on the GPU as a single task to get high throughput [HY11]. GPUTx also developed a mechanism to identify the conflicting transaction to maintain database consistency by using T-graph (Transactional Graph). GPUTx was able to achieve 4-10 times higher throughput than its CPU-based counterpart. This research showed that GPU accelerated databases can outperform traditional DBMSs even for an OLTP workload.

The main limitations of GPUTx are:

a. GPU only Query Processing: GPUTx performs high throughput transaction processing for an OLTP workload using only GPU. This leads to research potentials of using GPU as an efficient co-processor along with CPU to further enhance the performance of query processing engine.

b. Transactional Dependency: The kernels written for GPUTx are similar to a stored procedures for a transaction i.e. GPUTx supports only a predefined set of transactions. For adding a new transaction to the system a new implementation of the kernel needs to be done.

c. Database Schema Dependency: Implementation of kernels depends on the schema design of the database. Any change in the schema calls for a change in the kernel. Also a new set of kernels has to be written for a generalized RDBMS.

Our work is aimed to further investigate the query processing capability of the CPU/GPU based query processing engine. We believe that there is potential of increasing the query throughput by using the processing capacity of both CPU and GPU as both of these devices have their sweet spots for different operations. Further our research is aimed to do operation based rather than task based query processing. This can be done by creating general operators needed for OLTP query processing and schedule it on corresponding device for execution. This will give our implementation freedom from transaction dependency and schema dependency.

To the best of our knowledge there is no research which investigate the most efficient storage scheme for GDBMS in an OLTP environment. Researches like GPUTx are inspired by OLAP based GDBMS to accept column store as the most efficient storage mechanism. This research will compare both Row Store and Column Store schemes to find out the most efficient storage scheme for OLTP based GDBMS.

4. Assumptions

This chapter will present the assumptions that we have made for the implementation of this work. This chapter will start by introducing all the assumptions that have been used for the implementation of a prototype database for OLTP workload execution. In the Second part of this chapter we will focus towards elaborating the TPC-C benchmark which has been used to simulate a real life OLTP database and workloads.

4.1 Assumptions in Implementation

This work is aimed at investigating the processing capabilities of a hybrid CPU/GPU systems for query processing. In a full-fledged GDBMS there are many factors like query plan generation, operator placements, transaction management and data placement which work in harmony for effective query execution. All of these topics are very wide in their scope and present their own challenges. To focus our work towards identifying the computing capabilities of the hybrid CPU/GPU systems, we have replaced many of these dynamic factors with manual or static techniques. This section is aimed to pre-declare all of these assumptions so that the reader can focus on the problems addressed in this research.

Here are the assumptions that we have used for the implementation of our work:

4.1.1 No GPU Memory overflow

In this work we have assumed that the data needed to be executed on the GPU can fit in GPU's memory. Hence all the required amount of data will be transferred to the GPU device in a single run. Due to this assumption the algorithms for implementing the operators do not consider any blocked approached which deals with chunks of data.

4.1.2 Static Transaction Management

Transaction Management is an important aspect of query processing in DBMS as during the parallel execution of multiple transactions it maintains all four ACID (Atomicity, Consistency, Isolation, and Durability) properties. Transaction management is out of the scope for this work as we have manually created the query plan and generated the workloads such that our system adheres to all ACID properties. The workload is created such that all the transactions are isolated from each other. The consistency of the critical data is maintained by executing multiple conflicting operations in parallel. Hence, instead of dynamically finding the transactional dependencies, we use the existing knowledge about the TPC-C workload to identify the conflicting operations or transactions and schedule them accordingly.

4.1.3 Static Operator Placement

As discussed in the previous section, operator placements has a major role in query processing in a hybrid CPU/GPU based systems. The state of the art hybrid databases uses either a learning based approach like HyPE or a statistical approach (Compile Time) to find out the best device for a particular operator to execute on. In our work we have scheduled operators with in a transaction to multiple devices such that the workload is evenly distributed between the devices. Hence, in this work we have scheduled the operators to devices based on the workload instead of scheduling preferably on a better performing device.

4.1.4 Static Query Plan Generation

A query plan is a blueprint for the query execution, it is an ordered set of all the operations to be executed so that the data adheres to ACID properties. In a full-fledged DBMS with a support for ad-hoc query execution, the query plan is generated dynamically.

Given the pre-defined set of transactions in TPC-C workload we have created a static query plan for each transaction. Hence, our query plans are hard bound to the TPC-C transaction types only and cannot be reused with any other transaction. For a fully functional OLTP database we need to create the mechanism to generate the query plan in the run time so that it can become independent of a particular workload.

4.1.5 Bulk Query Execution

In an OLTP environment the queries are mostly executed in ad-hoc model but in our work we have implemented a bulk execution model as we believe that with the ever increasing OLTP transactions and advent of Web 2.0 bulk execution can be supported for OLTP environment. Hence in our work, the complete TPC-C workload is executed as a single bulk. All the inputs needed for the bulk execution will be pre-loaded into the system. This execution model can leverage the computing power of a hybrid CPU/GPU based system as it will save the multiple data transfers from device to host, also bulk execution can leverage SIMD style of processing.

4.1.6 No Database Caching

One of the best ways to improve query performance is database caching as it caches some of the recently or frequently accessed data. In GPGPU computing ,caching can play a crucial role as it can reduce the overhead of data transfer from CPU memory to GPU memory. We consider this topic to be out of scope of this work and hence no cached data will be in consideration.

4.2 OLTP workload Assumptions

This section is aimed to describe the database and its corresponding workloads that have been used for simulating a functional OLTP system. This section gives an overview of the real life scenario that has been implemented in the TPC-C benchmark. Further this section will illustrate the complete database schema and transactions that we have used in this work.

4.2.1 Benchmark Overview

TPC-C is an OLTP benchmark published by the Transaction Processing Performance Council (TPC). TPC-C is different than some of the old benchmarks like TPC-A due to its multiple transaction types, more complex schema, and overall execution structure. The TPC-C benchmark is designed to simulate a general wholesale supplier structure. The workload is primarily a transaction processing workload with multiple queries executed as a bunch within a transaction.

Figure 4.1 on the following page, shows the company structure represented of TPC-C benchmark. On an abstract level, the company consists of one or more warehouses, each warehouse serves 10 districts and each district caters to 3000 customers each. The complete life-cycle of an order starts when a customer makes an order to its corresponding district and ends when the customer finally receives the order and issues the payment.

4.2.2 TPC-C Schema

The TPC-C schema needs 9 tables to represent the company structure and perform the life-cycle of a complete order. Figure 4.2 on page 23, illustrates the schema of TPC-C benchmark. It show all nine tables along with their relationships.

The warehouse table contains information about all the warehouses in the company. The district table represents all available districts along with their reference to a unique warehouse. The customer table maintains all the information about the customers registered in a company, each customer uniquely belongs to a district and warehouse. The Item table contains all available items in the company. The Stock relation maintains the stock level for each item in each warehouse. Each order placed by a customer is maintained in three relations. The Order table maintains a record of an order placed by a customer. The pending orders are maintained in New Order table which are deleted once the order is delivered. In Order-Line table, a separate entry is made for each item in a order. A history of the payment transaction is appended to the History table [LD93a].

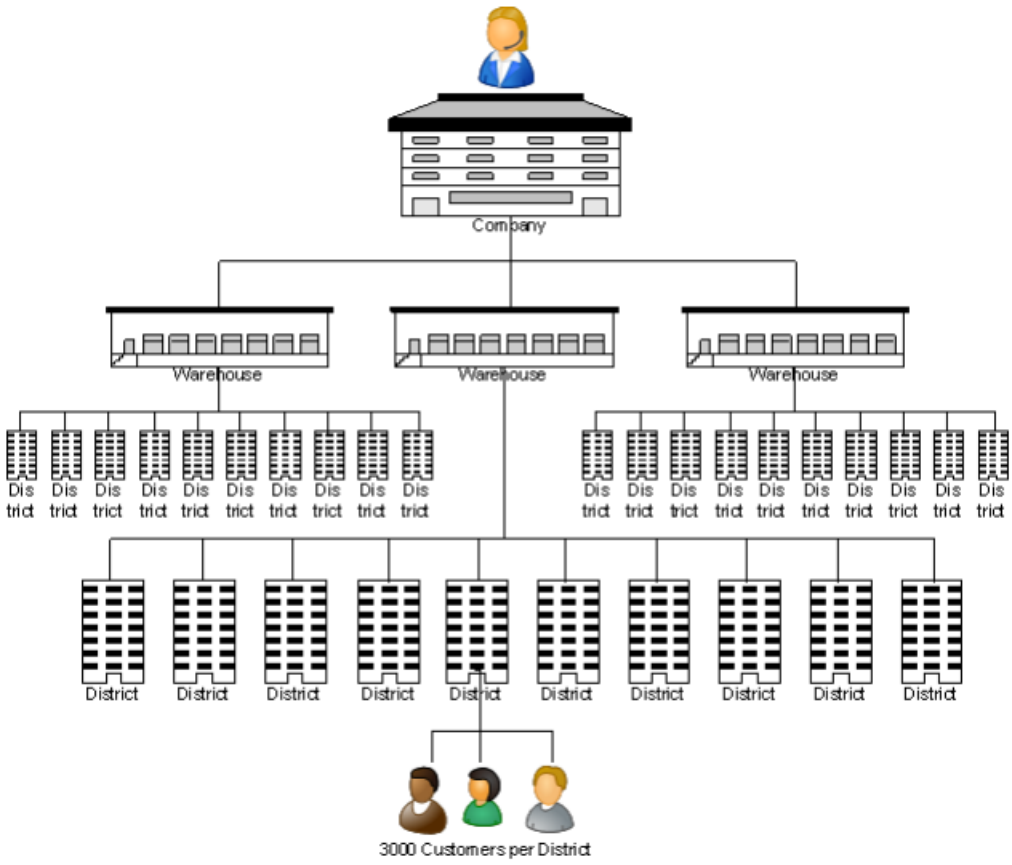

Figure 4.1: Company Structure

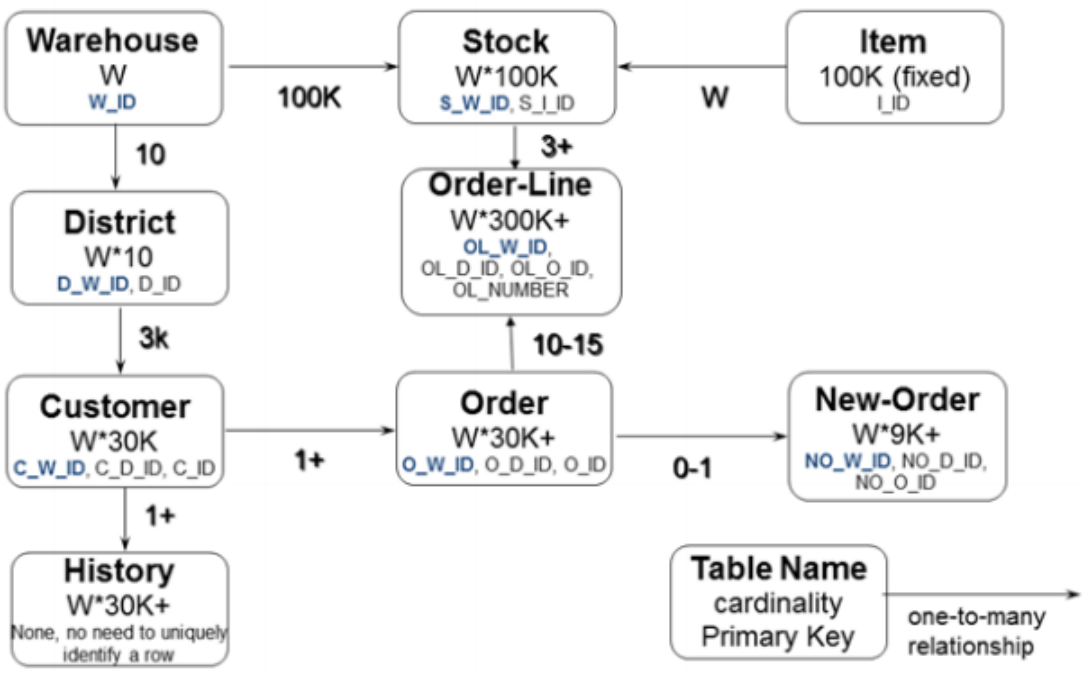

Figure 4.2: TPC-C Schema

4.2.3 TPC-C Workload

TPC-C benchmark is intended to model a medium complexity online transaction processing (OLTP) workload. It simulates an order-entry workload, with multiple transaction of varied complexity. The transactions ranging from simple transactions that are comparable to thecan be as simple as a simple debit-credit workload in the TPC-A/B benchmarks, or can have medium complexitywith more that two to fifty times the number of calls of the simple transactions [LD93a].

A mix of five concurrent transactions of varying type and complexity are executed on the database. Table 4.1 on the following page shows all the transaction types and their corresponding frequencies.

- The New Order transaction places an order for 5 to 15 items from a warehouse. [LD93b].
- The Payment transaction performs a payment for a customer. It updates various data in Warehouse, District and Customer relations. A customer is selected by a unique customer-id or by a name [LD93b].
- The Order Status transaction returns the status of the last order placed by the customer. As in the Payment transaction, the customer may be specified by the customer-id or by name. [LD93b].

Transaction	Frequency (Percent)
New Order	45
Payment	43
Order Status	4
Stock Level	4
Delivery	4

Table 4.1: TPC-C Workload

- The Delivery transaction processes orders corresponding pending orders, one for each district, with 5 to 15 items per order. The corresponding entry in the New-Order relation is deleted [LD93b].
- Finally, the Stock Level transaction examines the quantity of stock for the items that have been recently ordered from a district [LD93b].

5. Approach

For our work we have done separate implementation of TPC-C schema in column store and row store storage structure. The query execution mechanism used for both of this approach is same, where we have implemented various operators as OpenCL kernels. These kernels will be used to execute the TPC-C workload. The operators will be reused throughout the execution of workload wherever necessary. The host program will create non-conflicting and ACID properties complying query plans though which it will send data to the executing device and invoke the kernels. The query plan will also make the decision of scheduling an operator on a device and will try to put the operator in the respective device that best suits it in terms of performance. Host will also make sure of not underutilizing or over utilizing any particular device so that proper load balancing is done between devices e.g. if operator 'A' performs better on GPU, but for execution on GPU it needs to be in queue, whereas CPU is free. In such scenarios to avoid the overhead of waiting in queue, operator A can be scheduled on CPU as it has no task at hand.

The approach to implement our work can be broadly classified into the following sections:

1. Row Store and Column Store Implementation.
2. Test Data Generation to be sent as input to transactions.
3. Operators Implementation
4. Operator placement decision
5. Implementation of TPC-C transactions

5.1 Row Store and Column Store Implementation

Given the characteristics of both storage mechanisms, we need a completely different set of data structures to efficiently represent row store and column store storage mechanism.

5.1.1 Row Store Implementation

The data in a row store implementation requires the storage of rows of a table in contiguous blocks of memory. Figure 5.1 shows row store storage mechanism for a table with multiple columns represented with different colors. As we can see from the figure row-wise storage or n-ary storage model is used to represent row store, i.e. for a row all the column values will be stored in contiguous blocks of memory which will be followed by the column values of next row.

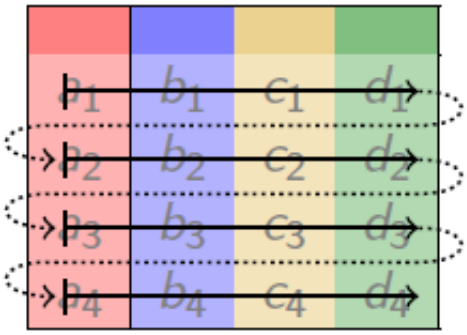

Figure 5.1: Row Store Storage mechanism[SB]

To represent row store storage model as a data structure we need a C++ structure.Each member variable of the structure represents a column of a table, hence, an instance of this structure will represent a row of a table. All the members in an instance of a structure are stored in contiguous blocks of memory e.g. for the table given in Figure 5.1 an instance of the structure will contain a1,b1,c1 and d1 as the values of member variables, this instance will represent the first row of this table.

To represent multiple rows of a table we can use C++ Vectors. C++ Vector is a sequence container that can represent an array with dynamic size. C++ Vectors use contiguous memory blocks for storing their elements, each members of the vector can be accessed using index or offsets. A C++ Vector of structure can be used to represent the complete table in a row store storage mechanism, where each index of the vector represents a row of the table. Given the contiguous storage mechanism of the vectors all the rows will be stored in contiguous memory locations.

Listing 5.1 on the next page shows the structure used to represent district table with all the members representing the columns in District table. All the columns in the table that represent string value are made a fixed length character array so that a single instance of this structure will have a constant size. A constant size of the structure is needed so that during the query execution the right amount of space can be allocated on the devices.

Listing 5.2 on the facing page represents the vector array that stores multiple rows of the tables. A vector of `District` is created to save the complete table data. Using the

```
struct District
{
    short int d_id;
    short int d_w_id;
    float d_ytd;
    int d_next_o_id;
    float d_tax;
    char d_name[10];
    char d_street_1[20];
    char d_street_2[20];
    char d_city[20];
    char d_state[2];
    char d_zip[9];
};
```

Listing 5.1: Row Store Implementation of District table

inbuilt vector functions like `push_back` and `erase` we can insert or remove a row from the table.

```
vector<District> dist_Data;
```

Listing 5.2: Vector storing row store table data

5.1.2 Column Store Implementation

In column store storage mechanism all the data inside a column of a table is stored in contiguous blocks of memory. Figure 5.2 shows the storage mechanism for column store data. The direction of the arrow in the figure shows the data storage sequence. At first all the data of first column is stored in contiguous blocks of memory followed by the data of second column and so on.

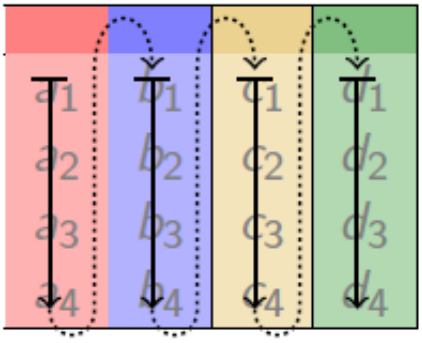

Figure 5.2: Column Store Storage mechanism[SB]

To represent column store data each column can be considered as a vector, so that each column will have homogeneous and contiguous blocks of memory. The complete table

can be represented by a structure which contains all the columns as vectors. Hence all the columns will be saved one after another in the memory. An instance of this structure will represent the complete table.

Listing 5.3 shows the representation of the column store data in C++ data structures. This code listing shows the representation of District table from the TPC-C framework. To represent the string columns a structure with the name **structArray** is created, this structure will only contains a character array of size 20. This structure has been so that all the columns with string values can be pushed to a vector.

```
1  std::vector<int> D_ID;
2  std::vector<int> D_W_ID;
3  std::vector<float> D_YTD;
4  std::vector<int> D_NEXT_O_ID;
5  std::vector<float> D_TAX;
6  std::vector<structArray> D_NAME;
7  std::vector<structArray> D_STREET_1;
8  std::vector<structArray> D_STREET_2;
9  std::vector<structArray> D_CITY;
10 std::vector<structArray> D_STATE;
11 std::vector<structArray> D_ZIP;
```

Listing 5.3: Column Store Data Structure

5.2 TPC-C Transaction

This section will provide the implementation overview of each transaction in a TPC-C benchmark that we have implemented in our work. The aim of this section is to make the readers aware about all the operators that have been used in TPC-C transactions and the Query plan of each transaction. Looking at these implementation overviews the reader can get an elaborate idea about the implementation details of the host programs. This section will also describe the input data that is needed to execute these queries in a bulk execution model.

5.2.1 New Order Transaction

This transaction simulates the process of creation of a new order by a customer for a specific district and warehouse. Each order in the transaction consists of 5 to 15 items. For each New Order transaction, an entry into three tables viz. New Order, Order Line and Order is made. For each item in an order, an entry is made in Order Line table. The order details are stored in the New Order and Order table whereas the details of each items associated with an order is stored in the Order Line table. When an order is delivered, then the entry of this order from New Order table is deleted and this data is archived using History table.

Listing 5.4 on the next page shows the implementation of the New Order transaction in a simplified way using a pseudo code. As we can see from the code listing a unique

customer is selected for triggering a new order transaction. Each customer is associated with a district and a warehouse. After selecting the customer an entry is made into the New Order and Order table. Each order consists of 5 to 15 items; these items are randomly selected from the Item table. For each item in a new order an entry is made into the Order Line table.

Listing 5.4: New Order Transaction: Pseudo Code

```
1   Get random (w_id) from Warehouse
2   Get random (d_id, w_id) from District
3   Get Random (c_id, d_id, w_id) from Customer
4   Insert into Order
5   Insert into New-Order
6   For each item in a order:
7        Get Random (item-id) from Item
8        Insert into Order-Line
9  Commit
```

5.2.1.1 Input

To perform New Order Transaction in a bulk scenario in an OpenCL context we need three input relations viz. New Order, Order and Order Line. These three input can be sent to the OpenCL devices along with the table or column data for the execution of the New Order Query. Once all the input relations are generated all the kernels associated with this transaction can be executed out of order. Listing 5.5 shows the data structure used to store the input data for New Order transaction in a Row Store implementation. We can see from the listing that each member of the structure represents the input relation. For Column Store implementation the input data will contain all the columns from respective table as vectors.

Listing 5.5: Input Data New Order Transactrion

```
1 struct InputDataNewOrderRowStore
2 {
3     vector<New_Order> no_input;
4     vector<Orders> o_input;
5     vector<Order_Line> ol_input;
6 };
```

5.2.2 Payment Transaction

Payment transaction processes a payment for an order by a random customer. There are two cases for selecting the customer:

1. For 40% of cases the customer is selected by customer id.

2. For 60% of cases the customer is selected by last name.

In case of multiple matches of the last name a customer is selected by a random choice. For each selected customer an update is made into the District, Warehouse and Customer Table. An Insert is made into the History table to record all the archival payments.

Listing 5.6 on the next page shows the pseudo code of the implementation of Payment Transaction.

Listing 5.6: Payment Transaction: Pseudo Code

```
1  Get random (w_id) from Warehouse
2  Get random (d_id, w_id) from District
3              Case 1: Select(c_id, d_id, w_id) from Customer
4              Case 2: Non-Unique Select (c_name d_id, w_id) from Customer
5  Update (w_id) in Warehouse
6  Update (d_id) in District
7  Update (c_id, d_id, w_id) in Customer
8  Insert into History
9  Commit
```

5.2.2.1 Input

The payment transaction needs the following inputs to execute the transactions for a bulk load in OpenCL context for Row Store Implementation:

1. Customer ID / Customer Name
2. District ID
3. Warehouse ID
4. Input table for History.

The Customer ID, Warehouse ID and District IDs are used to find the update positions for Customers District and Warehouse table. The input table for History table is used to insert new records into the History table.

For Column Store Implementation the input will be the same like above, only the Input table for History table will be replaced by many input columns that form the History table.

5.2.3 Delivery Transaction

The Delivery transaction simulates the database operations to be done for a successful delivery of an order. All the recent orders are present in the New Order table. Hence, the delivery transaction picks the latest order IDs from the New Order table. Listing 5.7 on the facing page shows the pseudo code for the execution of the Delivery Transaction.

We can see from the listing that random order ids are selected from the New Order table. These randomly selected order ids will be used as input order ids to Delivery transaction. A Delete operation is performed on the New Order table for all the randomly selected order ids. This is followed by an update operation on the Orders table which will set the order as delivered. All the entries in the Order Line table for the input order ids are also selected using a join operation and corresponding rows are updated to set the delivery date. Towards the end a select operation is performed on the Customer table to select the effected customers that have placed the input orders. Finally the selected Customer‘s data is also updated to update customer balance and credit limit.

Listing 5.7: Delivery Transaction: Pseudo Code

```
Get random (o_id) from New Order table
Get (o_id , w_id, d_id) from New Order table
Delete (o_id) from New—Order
Select (o_id) from Order
Update (o_id) Order
For each joining item in Order Line corresponding to o_id (i.e. 5 to 15):
            Select (o_id) from Order Line table
            Update columns in Order Line table
Select (c_id) from Customer
Update (c_id) Customer
```

5.2.4 Order Status

The Order status simulates the process of providing the status of an order that has been queried by the end user. In a nutshell it returns the information of the customer and details about the order. This transaction takes customer id as input and returns all the information of the latest order placed by this customer. The customer id is determined exactly like in Payment transaction where 40% of customer ids are retrieved by customer's last name and remaining 60% by customer's id. The Order Status transaction only requires a unique set of customer id and its associated district id and warehouse id. Using various selection and join operations all the information of Order and Order Line table is retrieved.

Listing 5.8 shows the pseudo code for the execution of the Order status query. We can see from the listing that the customer information is selected in the same manner like payment transaction. For each selected customer a unique order id is retrieved from the Order table which represents the latest order by this customer. To find the latest order for a customer we have to find the maximum order id for the searched customer. After selecting the latest order ids a join operation is performed with the Order Line table to select all the rows in the Order Line table that refers to the selected order ids.

Listing 5.8: Order Status Transaction: Pseudo Code

```
Select Customer
       Case 1: Select (Random (c_id), w_id, d_id) from Customer
       Case 2: Non—Unique Select (customer—name, d_id, w_id) from Customer
Select (Max (o_id), c_id, d_id, w_id) from Order
For each item in the order:
       Select (o_id) from Order Line
Commit
```

5.3 Basic Approach

The basic methodology for implementing an operator in an OpenCL context is same for all operators with minor changes in input and output data. This section will introduce this basic methodology in detail. The aim of this section is to let the reader understand the execution mechanism of OpenCL kernels; it will let the reader understand the approach used for implementing operators used in this work with more clarity.

To implement an operator in an external device like GPU firstly the host program sends the input data to the device as an input buffer. The same instruction set is then applied on the input data through a kernel program. After the processing of the input data, a result set is generated that is sent back to the host program in output buffer. The host program predefines the memory requirement of the input buffer and the output buffer. Hence, the host program needs to pre-calculate the expected amount of memory that needs to be used by OpenCL device output and the input buffers.

The approach to implement an operator in the bulk scenario can be broadly classified into following stages:

1. Position Calculation
2. Kernel Creation
3. Input/ Output Buffer Creation
4. Copy Buffers to OpenCL Device
4. Scheduling Operators
5. Reading Results

5.3.1 Position Calculation

Most operators are applied based on a predicate. To apply any operator on a table we need to firstly identify all the rows (in Row Store) and columns indexes(in Column Store) that satisfy the predicate. In position calculation phase all the indexes of the tables that are to be manipulated by the operator is calculated by the host program. A positions array is created based on the indexes of the selected rows of the table. In a bulk OLTP execution model, multiple predicates are matched to create a position array. Figure 5.3 on the facing page shows the algorithm for the creation of positions array from an input array based on a predicate. We can see from the figure that a flag array is of the size of the input array is created. The input array is scanned and for each index that satisfies a predicate a value of 1 is inserted in the same index of flag array. The flag array is used to find the size of the result set. This result set is then filled with the index positions of the selected rows.

5.3.2 Kernel Creation

The instruction set that is executed on the OpenCL device is referred as kernel. A kernel is a function written in C programming language which can only be executed in an OpenCL device. The host program is responsible for creation, scheduling and providing input parameters to the kernel. Listing 5.9 on the next page shows a sample kernel program that takes three input parameters. The value array and positions array are sent to the kernel to find the values of the indexes that satisfies the predicate. The processed data is then stored in the output buffer which is passed as the third parameter. OpenCL provide a mechanism to access the unique id of each work item that is executed in an OpenCL device for a kernel execution. Each execution of the work item can access its unique Id using get_global_id (0) function.

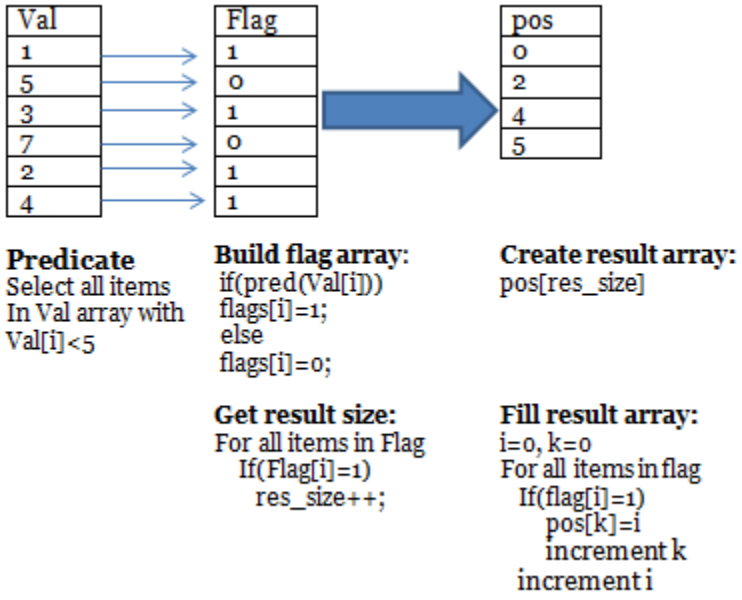

Figure 5.3: Position Calculation Based On Predicate

Listing 5.9: Example: Kernel Program

```
1 __kernel void op_kernel(__global int* val, __global int* pos ,__global int* output_Data)
2 {
3     const int g_id = get_global_id (0);
4     int index= pos[g_id];
5     output_Data[g_id]=val[index]+10;
6 }
```

5.3.3 Input/Output Buffer Creation

The data is transferred to the OpenCL devices like GPU through buffer objects. All the data that needs to be processed as an input of an operator is sent to the OpenCL device as an input buffer. The position array created in Figure 5.3 is also sent along with the input relation or column to identify the indexes of the effected rows. The result or the output of an operator is stored in output buffer. In operators like insert and select an empty output buffer is sent to the OpenCL device which is then filled with processed results using input buffer. In operators like update and delete the input array itself needs to be edited and returned as result hence, the input array is return back as output buffer after processing.

Listing 5.10 on the next page shows the code to create buffers in an OpenCL context. We can see from the listing that three buffers have been created for performing an operation on data created in Figure 5.3. The parameters for creating buffers in clCreateBuffer functions are:

1. **context**: an OpenCL context associated with a device.

2. **cl_memory_flag**: a bit field used to specify the usage information.

3. **size_t**: size of the memory allocated.

4. **errorcode_ret**: it returns an error code. Setting it to null returns no error codes.

Two input buffers have been created which contains the data of the position array (pos) and Value array (val). Since the value of the position array is never changed in the course of the operation, it is kept as a read only memory. The third buffer is created to get the output result from the device, the size of the output array is the same as the size of the position array, and hence the size of the position array is used to get the size of the output buffer.

Listing 5.10: Code sample to create buffer objects

```
1
2 cl_mem ib_pos_index = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, sizeof
      (int)*res_size, pos.data(), &error);
3 cl_mem ib_val = clCreateBuffer(context, CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR, sizeof(int)
      *6, Val.data(), &error);
4 cl_mem ob_output = clCreateBuffer(context, CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR, sizeof(
      int)* res_size, output.data(), &error);
```

5.3.4 Copy Buffers to OpenCL Device

In this work we have not considered the overheads due to transfer of data from the host memory to an external device like GPU. To avoid this overhead we need to store the data needed for kernel execution in the OpenCL device beforehand. Listing 5.11 shows the code block which transfers the data from the host memory to the OpenCL device. As we can see from the code **clEnqueueWriteBuffer** method is used to do the data transfer. This method takes the command queue associated with a OpenCL device as parameter along with the information of the buffer object that needs to be transferred. The code sample given in Listing 5.11 copies the buffer objects created in Listing 5.10 and puts them on the GPU as the command queue associated with the GPU is used as the parameter.

Listing 5.11: Code sample to create buffer objects

```
1
2 clEnqueueWriteBuffer(queueGPU, ib_pos_index,CL_TRUE, 0, sizeof(int)*res_size, pos.data(), &
      error);
3 clEnqueueWriteBuffer(queueGPU, ib_val ,CL_TRUE, 0, sizeof(int)*6, Val.data(), &error);
4 clEnqueueWriteBuffer(queueGPU, ob_output ,CL_TRUE, 0, sizeof(int)*res_size, output.data(), &
      error);
```

5.3.5 Operator Scheduling

In our scenario operators are the OpenCL kernel programs. The decision to perform a kernel execution on a particular device is made by the host application. The decision of

the host application to put a particular kernel on a particular device is referred as operator scheduling. Operator scheduling in an important aspect in OpenCL programming in a hybrid scenario as it lets the host application to effectively utilize a particular device for a better performing operation, it can also let the host application to do effective load balancing between devices.

Listing 5.12 shows the creation of an OpenCL kernel using a kernel program. The host application reads the **op_int_insert.cl** file from the file system. This file contains an OpenCL kernel program. The **cl_program** object is created using the loaded program. The created program is then built to check for errors. After successful build a kernel object is created. The buffers created in Listing 5.10 on the preceding page sections can now be sent as an input parameter to the created kernel. After setting the kernel argument the kernel can be queued for execution on a device using the command queue of a particular device. In the listing **globalWorkSize_CPU** represents the total number of work items that will be processed in the kernel execution. In this scenario the total number of work items processed will be the size of the position array as corresponding to each index of the position array a work item will be executed.

Listing 5.12: Creation of OpenCL kernel and Scheduling

```
1 cl_program program_int_insert = CreateProgram(LoadKernel("operator.cl"), context);
2 CheckError(clBuildProgram(program_int_insert, deviceIdCount, deviceIds.data(), nullptr,
      nullptr, nullptr));
3 cl_kernel kernel = clCreateKernel(program_int_insert, "operator_name", &error);
4 clSetKernelArg(kernel, 0, sizeof(cl_mem), &ib_res_index);
5 clSetKernelArg(kernel, 1, sizeof(cl_mem), &ib_val);
6 clSetKernelArg(kernel, 2, sizeof(cl_mem), &ob_output);
7 CheckError(clEnqueueNDRangeKernel(queueCPU,kernel,1,nullptr,globalWorkSize_CPU, nullptr,0,
      nullptr,nullptr));
```

5.3.6 Reading Results

After the successful execution of the kernels the results are sent back to the host application from OpenCL device. The results are read from the output buffer to the a data structure that stores result in host application. Listing 5.13 shows the code to read the data from the output buffer. We can see from the code listing that the data from the output buffer (**ob_output**) is copied from the CPU queue to a data structure named **output** in host application. The size of the output object and output buffer must exactly match to avoid errors due to over flow and under flow of memory.

Listing 5.13: Code sample for reading buffer

```
1 CheckError(clEnqueueReadBuffer(queueCPU,ob_output,CL_TRUE,0,sizeof(int)*res_size, output.data
      (), 0, nullptr, nullptr));
```

5.4 Row Store vs Column Store Kernel

In our work the kernel implementation for Row Store and Column Store methods are different due to the use of different data structures for both scenario. The Column

Store kernels are dependent on a data type of the column whereas the Row Store kernels are tightly bound to the schema of the operand table. This make our Row Store kernels non reusable for different table schemas. Listing 5.15 and Listing 5.14 shows the implementation of the Row Store and Column Store Implementations of a kernel to update a float column names **w_ytd** in Warehouse table. In the Column Store implementation we can see that a generic kernel is created which takes an input column and the update positions and performs the update on the input column. This same update logic can be reused in any other float column by using the same kernel.

Listing 5.14: Column Store Kernel

```
__kernel void OP_INT_ADD_CONSTANT(__global int* pos, __global float* val)
    {
        const int g_id = get_global_id(0);
        int id = pos[g_id];
        int id = pos[g_id];
        val[id]+ = 10;
    }
```

The implementation of the same operation in a Row Store kernel requires the redefinition of the structure representing the Warehouse table. An array of the structure representing Warehouse table along with the position is sent to get the row index for update. The **w_ytd** property of the row is then accessed and the update operation is performed. The Row store implementation requires the definition of the table structure inside the kernel and hence unlike column store implementation, this kernel cannot be reused for updating any float column from other table.

Listing 5.15: Row Store Kernel

```
typedef struct Warehouse
{
    short int W_ID;
    float w_ytd;
    float w_tax;
    char w_name[10];

}Warehouse;

__kernel void op_Update_Warehouse(__global Warehouse* input_data, __global int* pos)
{
       const int g_id = get_global_id (0);
    int index= pos[g_id];
    input_data[index].w_ytd+= 10;

}
```

5.5 Operators Implementation

You will see TPC-C transaction section that we have used Insert, Update, Delete, Select and Join operators for implementing TPC-C queries. This section only introduces the methodology that has been used to perform these operations using a data parallel

approach in OpenCL context. The technical implementation of each operator has been done using the same basic methodology defined in Basic Approach section.

5.5.1 Insert

The insert operator inputs the data into the table. In TPC-C workload insert operator is used in New Order transaction. In New Order transaction the input data is given to the operator along with the table or column where the data needs to be inserted. The insert operator will then insert the data into the table or column using the data parallel approach. Insert operation do not require the position array as the new records are only added at the end of each table or column and not at a specific index.

Figure 5.4 shows the concept used for insert approach using the data parallel method. As we can see from the figure that an output buffer is created with null values and sent to the kernel along with the input data that needs to inserted. The information about the amount of data to be inserted is also provided to the kernel which is termed as work size in OpenCL terminology. We do not need to calculate the output result size because during the execution of New Order transaction the amount of new orders to be generated are known beforehand. Using the result size the host program can allocate the right amount of space to the output buffer.

All the threads of the device can independently process the kernel with their own chunk of input and output data. Figure 5.4 represents threads functioning with blue lines where each line represents a thread. Each thread reads the input from a unique index in input array and replaces the same index in the output array with it. After the execution of the threads the output buffer will have all the values from the input buffer. After successful execution of the kernel the output array is read by the host program.

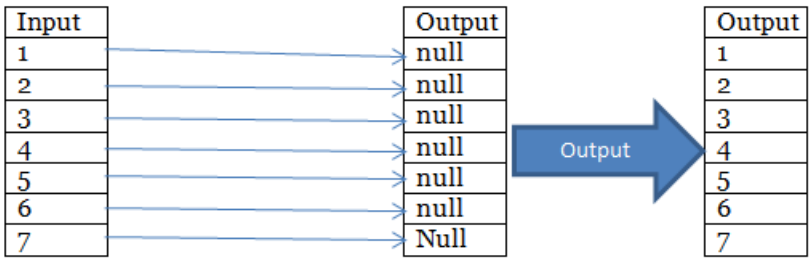

Figure 5.4: Insert Operator Approach

5.5.2 Update

Update operator updates the values of a column or a row. Unlike Insert operator, the Update operator requires index position of a column or row where update operation needs to be performed. These index positions are calculated based on a predicate. Update positions and input data are provided to the kernel as buffers. Since in an

update operation the row or columns are updated itself hence the input table or column are treated as output buffer.

Figure 5.5 shows the update operation for an array named Val that satisfies a predicate. All the items in the Val array that satisfies the predicate Val[i]<5 should be set to 100. To do this operation firstly the update positions are calculated using the approach discussed in earlier section. The position array and the value array are sent to the OpenCL kernel as input buffer and output buffer. For all the values in position array corresponding index are updated to 0 in the value buffer. Finally the output buffer is returned back to the host application where the updated value array is retrieved.

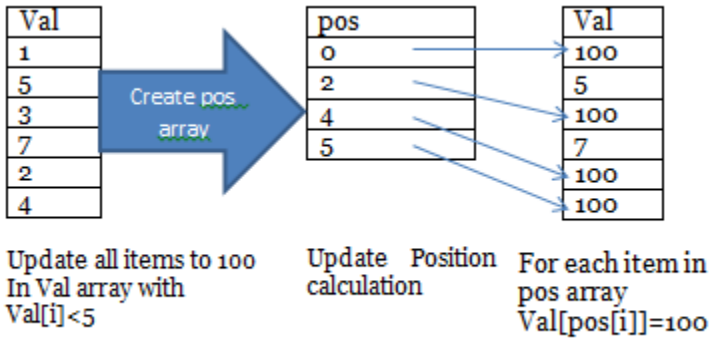

Figure 5.5: Update Operator Approach

5.5.3 Delete

It is not possible to allocate or de-allocate memory inside an OpenCL kernel as it only works on the memory allocated by host program. To perform delete operation we need to set an invalid flag on the row or column index. The value of the Primary key is set to 0 inside the kernel program. The Row or Column with invalid flag is read back to the host application where it can be de-allocated in regular intervals.

The implementation of the Delete kernel is very similar to the Update kernel as we only need to update the value of the primary key to 0. Figure 5.6 on the facing page depicts the implementation details for Delete operation in an OpenCL context. Firstly the position value is calculated based on the predicate. Using position array as input buffer and Value array as output buffer the OpenCL kernel sets the values inside the input array as 0 and sets them as invalid. The updated column will always be the primary key of the table.

5.5.4 Selection

The selection operator is a unary operation that collects the result which satisfies a predicate and sends it back as an output. The host needs to find out the size of the

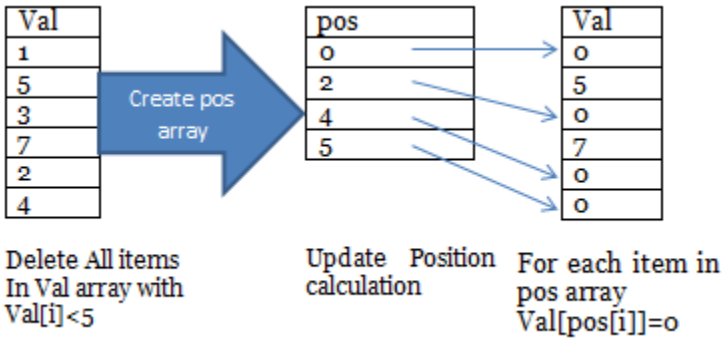

Figure 5.6: Delete Operator Approach

selected result beforehand so that it can allocate the space for output and send it to the kernel. The kernel can then fill the result in the allocated space and send it in an output buffer. Different implementations for a selection operator have already been investigated by Broneske et al. [BBS14].

Figure 5.7 on the next page shows the algorithm used for selection operator. This algorithm selects all the items in the Val array whose values are less than 5. To perform this operation in an external device like GPU we need to first identify the result size. An array containing the positions of the selected rows or column is created using the result set. An empty array to contain the result set is constituted using the result set. The position array and the value array are passed in input buffers and the result array is passed in an output buffer to OpenCL device. The value of the result array is filled using the position array and the value array. The result array is finally sent back to host application.

5.5.5 Join

Join operators are used to get the joining results from two tables based on a join condition. Join operators are way more heavy weight as compared to other operators in general. In selection operator the maximum size of the result can be equal to the input relation, whereas in a join operation this can be very large e.g. if we want to join two relations R and S then their maximal result size can be as big as (|R|.|S|) or even larger. The problem becomes more difficult to handle in case of multiple join partners.

In TPC-C workload the Queries like Delivery and Payment needs the join operation. Figure 5.6 shows two tables i.e. Order and New Order that needs to be joined on the given condition. In our implementation the the join operator is not performed on the OpenCL devices, instead the row or column indexes of the joining result is calculated on the host application and sent to the OpenCL devices. Hence, to perform the join we just need to perform the join between the columns used in joining condition. Hence, the joining positions are retrieved by performing a nested loop where each index of the joining

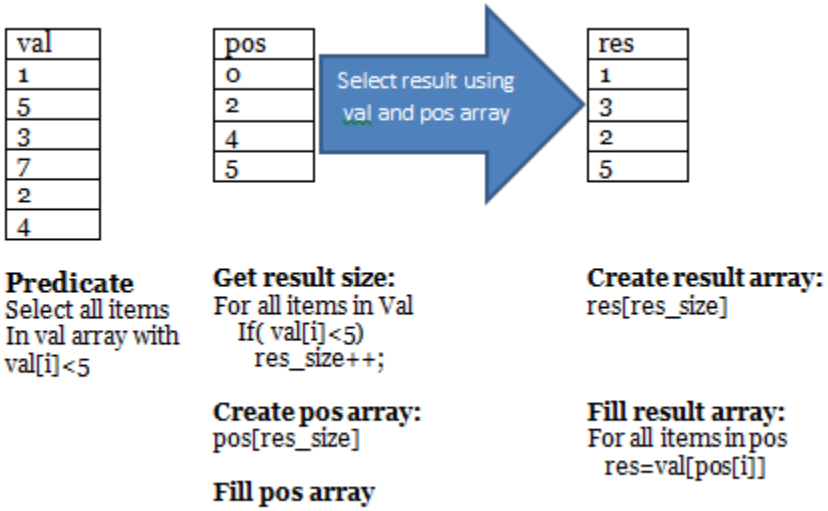

Figure 5.7: Select Operator Approach

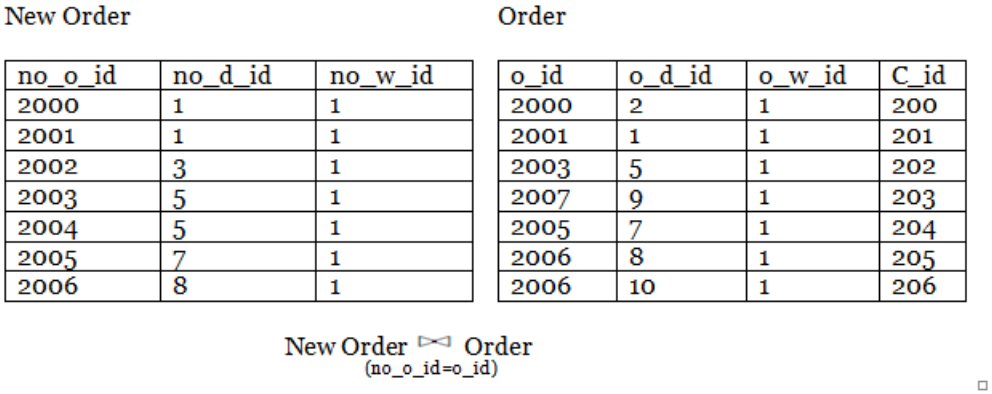

Figure 5.8: Join Operation

column of the left operand is compared with all the indexes of joining column in right operand.

Listing 5.16 shows the join position calculation where both the joining columns are iterated in the nested loop to get the joining indexed of the right operand.

Listing 5.16: Join Position Calculation

```
int k=0;
for (int j = 0; no_o_id.size(); j++)
{
  for (int i = 0; o_id.size(); i++)
    {
     if (no_o_id[j] == o_id[i])
      {
       Pos[k]=j;
       K++;
      }
    }
 }
```

5.6 Workload Distribution

In a hybrid system the workload needs to be divided between the CPU and GPU. In our work we have manually distributed the workload using the host program. During Workload distribution special emphasis is given on utilizing both CPU and GPU to its full potential. Previous researches in the field of GPGPU have clearly suggested that GPU is a faster device to perform the database operations [GLW+04]. Keeping this in mind we have tried to distribute the workload such that GPU gets to do the major part of the work.

In New Order transaction there are entries made to New Order, Order and Order Line table. Corresponding to each new order placed there is a single entry in New Order and Order table but 5 to 15 entries in Order Line table one for each item in a order. For new order transaction we have performed the input in New Order and Order table on the CPU whereas the insert of Order Line in GPU. To summarize, if we place 100 new orders with 5 items each then 100 entries will be made into New Order and Order table and 500 entries will be made into Order Line table. Hence in total 200 inserts will be performed on CPU and 500 inserts on GPU.

6. Evaluation

This chapter will give a detailed description of the evaluation setup, evaluation mechanism and result of our thesis. This chapter will start by introducing the evaluation setup that has been used for our research. Further, in this chapter we will show the evaluation mechanism for both of our research questions. There will be two evaluation scenarios which will be discussed in this chapter:

1. Comparison of the execution time of TPC-C workload for Row Store and Column store storage mechanism on a hybrid CPU/GPU based system.

2. Comparison of the execution time of TPC-C workloads on hybrid CPU/GPU based system and CPU only system.

Towards the end, this chapter will give a brief description of the evaluation results and will answer our research questions that have been set forward. Finally, we will discuss some of the factors than can be a threat to the validity of our evaluation followed by a collective conclusion.

6.1 Evaluation Setup

The machine used for evaluation has the following configuration:

- CPU: Intel(R) Core(TM) i5-2500 @3.30 GHz
- GPU: NVIDIA GeForce GT 640
- OS: Linux
- API: OpenCL

Workload ID	New Order	Payment	Delivery	Order Status
Workload 1	1000	1000	100	100
Workload 2	10000	10000	1000	1000
Workload 3	100000	100000	10000	10000
Workload 4	200000	200000	20000	20000
Workload 5	300000	300000	30000	30000
Workload 6	400000	400000	40000	40000
Workload 7	600000	600000	60000	60000
Workload 8	800000	800000	80000	80000

Table 6.1: TPC-C Workloads for evaluation

To answer our research question we need to perform a comparative study of the execution time of the TPC-C workload for various scenarios. Hence, the configuration of the system will have little impact on our end results as all the scenarios will be executed on the same machine. Further, we have kept the workload size to a limit where it does not overrun our GPU memory or RAM size.

Table 6.1 shows different workloads that have been used for performing the evaluation each workload has a different number of transactions. The ratio of the number of each transaction is decided based on Table 4.1 on page 24.

6.2 Row Store vs Column Store

This section of evaluation chapter is focused towards answering our first research question which has been restated below:

RQ1: Which is the most efficient storage mechanism for OLTP query processing on a hybrid CPU/GPU System?

6.2.1 Evaluation: Workload level

To answer our first research question we have executed the TPC-C workloads define in Table 6.1 in a hybrid CPU/GPU based system for Row Store and Column Store storage mechanism. We will show our evaluation results firstly by comparing the kernel execution time of each workload in Row Store and Column Store storage mechanism. Further we will drill down into each transaction to see the impact of these storage mechanisms at a operator level.

Figure 6.1 on the next page shows the comparison of execution time of TPC-C workloads for Row Store and Column Store storage mechanism. We can see from the graph that Column Store implementation performs better for all the workloads except the first two smaller workloads. For heavier workloads Column Store implementation can outperform the Row Store implementation by a factor of 4x.

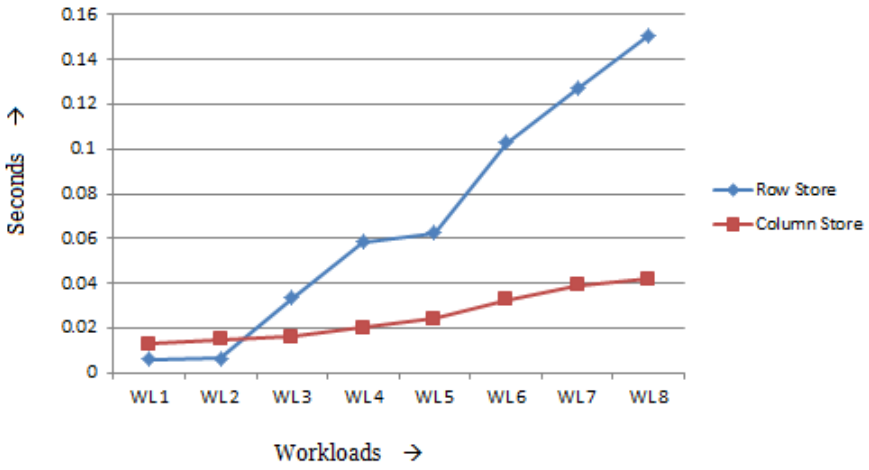

Figure 6.1: Row Store vs Column Store Execution time

6.2.2 Result Discussion

In this section we have discussed the evaluation results obtained by comparing Row Store and Column Store implementation. We will start by discussing the reason for the fast performance of Column Store implementation for heavier workloads. The next part of this section will discuss the reasons for fast performing Row Store implementation for smaller workloads. Finally,we will draw a conclusion from these results and will answer our research question.

The fast performance of Column Store implementation for heavier workloads can be attributed to following factors:

1. Better Coalesced Memory Access: In a Column Store implementation memory access is done in a more coalesced way as compared to Row Store Implementation. In operations like select, delete and update local memory of the work group is better utilized as multiple column indexes that are required by a work group can be loaded together. This will lead to faster kernel execution as multiple access to slow performing global memory is reduced.

2. Minimum Projection: In Column Store kernel we only need to load the columns which are needed to perform a particular operation, this is not possible in Row Store kernel as complete table is sent as a single unit to the OpenCL device. Figure 6.2 on the following page shows this phenomenon where only two columns that are needed for performing an operation are projected. There are many operations in OLTP workload like update and delete which needs to access only a single column. These operators are performed better in a Column Store implementations.

3. Fewer instructions in a kernel: The Column Store kernels are very light weight as compared to Row Store kernels in terms of instructions. To update two column in a table we need to create two kernels in a Column Store system whereas a single kernel in

Figure 6.2: Minimal Projection

Row Store kernel with more instructions. In a hybrid scenario a Row Store kernel cannot take full advantage of external devices like GPU as performing heavy instructions in them seriously effects the performance.

The slow performance of Column Store implementation in case of smaller workloads can be attributed to the following factor:

OpenCL function call overhead: Each call to an OpenCL function has an overhead as every function call to an OpenCL device is first scheduled to the appropriate driver which then schedules the activity for processing. Hence, every kernel call from the host program has a function call overhead. This functional overheads are less in Row Store implementation compared to a Column Store implementation e.g. to perform an insert operation for a table with 10 columns there will be 10 kernel(one for each column) scheduled in a Column Store implementation whereas only one kernel would be scheduled in Row Store implementation. The extra time taken by functional overheads in a Column Store implementation is hidden by benefits discussed above for bigger workloads. In smaller workloads the overheads of functional overheads in Column Store implementation takes a considerable time as compared to the actual execution time. This leads to better performance of Row Store kernels in smaller workloads.

6.2.3 Evaluation: Transaction level

To further study the effect of Row Store and Column Store storage mechanism on an OLTP workload we have studied the execution time of each transaction for workload 8. Each transaction uses a unique set of operations; this will give us a clear idea about the performance of each operator in both storage mechanisms. Figure 6.3 on the next page shows the comparison of the execution time of each transaction of TPC-C benchmark in Column Store and Row Store mechanism. We can see from the figure that only the New Order transaction performs better in Row Store Implementation whereas all other transactions perform better on the Column Store implementation.

6.2.4 Result Discussion

Looking at the nature of New Order transaction we can say that only the insert operations perform better on the Row Store implementation.There are various factors that lead to the better performance of insert operation in Column Store implementation:

1. Less Functional Overheads: In an insert operation all the columns of the table are accessed which is not the case of operations like delete and update. Due to this behaviour a single Row store kernel can perform the task of insert in a Row Store kernel whereas in Column store multiple kernels will be needed. Hence the Row Store kernels will have less functional overheads.

2. Better Coalescing: One of the biggest benefit of Column Store Data is minimal projection which allows only the transfer of required columns to the OpenCL device.This behaviour makes OpenCL kernels more efficient as better coalescing is achieved in a single column. As we require all the columns of a table in an insert operation, the Row Store implementation can also get better coalescing as multiple work items access multiple rows which are stored sequentially.

All other operators like update, delete and select do not use all the columns of the table, hence the column store implementation performs better then Row Store counterpart for these operators.

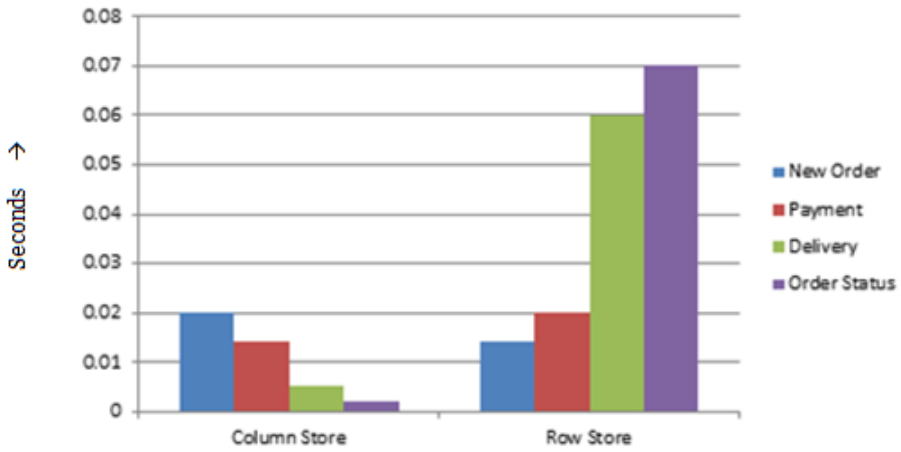

Figure 6.3: Column Store vs Row Store: TPC-C transaction execution time

6.2.5 Conclusion

In our research we have found that leaving Insert operation all other operators perform better on a Column Store storage mechanism. Even though the workload of TPC-C contains about 50% Insert operation through New Order Query, but still the performance gain achieved in other operators overpowers the slow performing insert operations for a Column Store scenario. Hence, even after having better performance of many Insert operators in TPC-C workload Column Store implementation outperforms the Row Store implementation. Apart from it their are various proven factors like Vectorizations and Data Compression that can be effectively utilized to further boost the performance of a Column Store implementation. In the end we can safely say from our evaluation that for a Hybrid CPU/GPU based system Column Store is a better storage method

as it effectively uses the memory hierarchy of the CPU and leads to better cohesion in memory access.

6.3 Hybrid CPU/GPU based system vs CPU Only System

In this section the evaluation will be centred towards finding the answer to our second research question which is restated below:

RQ2: Is OLTP query processing on CPU/GPU based query processing engine faster than traditional CPU only system?

6.3.1 Evaluation: Workload level

To answer our second research question we have performed a detailed comparison of TPC-C workload execution time for a hybrid CPU/GPU based system versus a CPU only system. In an OpenCL context, all the kernels in a CPU only system will be queued to the CPU device whereas, in the hybrid scenarios the kernels will be distributed amongst CPU and GPU. The Data that needs to be worked on is separated manually on the host program and copied to the respective device before the kernels are executed.

To compare the execution time of CPU only system with hybrid CPU/GPU based system we have performed the execution of all the TPC-C workloads given in Table 6.1 on page 44 for Column Store implementation of TPC-C schema. We will be using Column Store implementation for this evaluation as we have seen from the previous section that Column Store is a better storage mechanism for implementing the CPU/GPU based hybrid system for an OLTP workload.

Figure 6.4 on the next page shows the comparison of the execution time of each workload in both scenarios. We can see from the trend of the graph that for small workload the execution time of CPU only machine is comparable to hybrid CPU/GPU based machine. With increasing size of the workload the hybrid system starts over performing the CPU only counterpart. For heavier workloads like Workload 7 and Workload 8 we can see that the hybrid system outperforms the CPU only system by a factor of 2x.

Although the above analysis gives a clear edge to the hybrid systems but, to study the exact behaviour of the hybrid system we need to do a comparative study of each OpenCL device used and their performance. To further drill down we did a comparative study amongst CPU only, GPU only and Hybrid systems to find the better performing device. Figure 6.5 on the facing page shows the execution time of different workloads for each of the mentioned scenarios. We can see from the graph that the GPU only system performs the best as compared to any other systems followed by hybrid CPU/GPU based system. This behaviour clearly indicates that GPU is a faster device to execute OLTP workload than CPU.

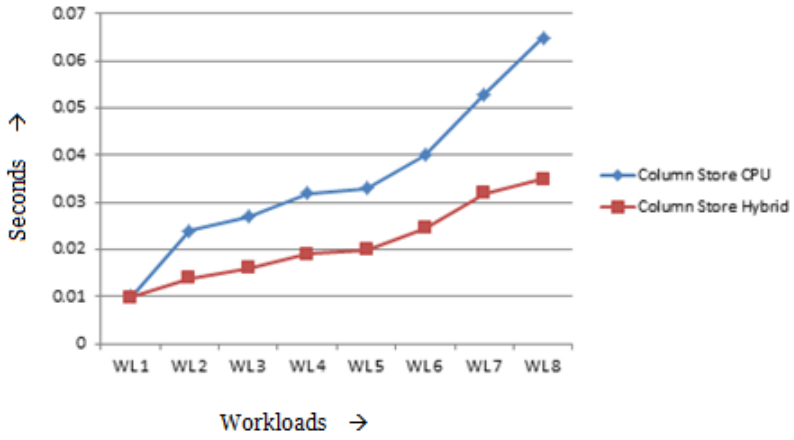

Figure 6.4: Hybrid vs CPU only execution time comparison for Column Store implementation on TPC-C benchmark

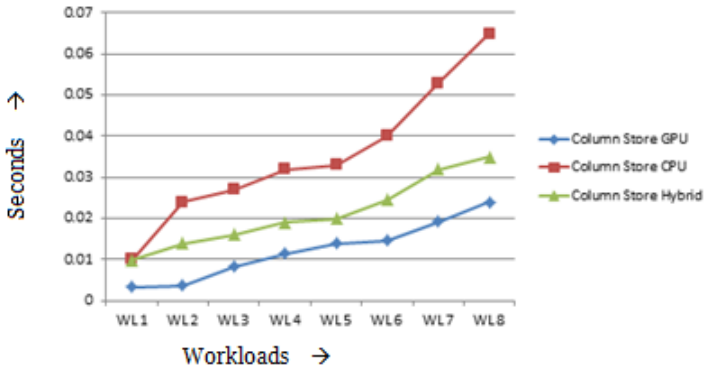

Figure 6.5: Comparison of Execution time of CPU, GPU and Hybrid System

6.3.2 Result Discussion

Even through the hybrid implementation outperforms the CPU counterpart by a factor of 2x for heavier workloads, still it under performs when compared to a GPU only system by a factor of 2x for heavier workloads. The reason for this behaviour can be attributed to following factors:

Under utilization of GPUs: In our implementation we were not able to properly distribute the load between the devices which led to under utilization of GPUs. For better performance of a Hybrid system the fast performing device should be fed with more tasks which will lead to a significant improvement of the overall system.

Lack of Inter Transaction Parallelism: Our implementation performs all the TPC-C transactions sequentially. Hence, at times there are not enough tasks that can be allocated to a fast performing device like GPU. For better utilization of the GPU there should be enough tasks available that can be scheduled as it will greatly reduce GPU's idle time. Performing multiple transactions in parallel will make more tasks available which can be scheduled to each OpenCL device.

6.3.3 Transaction level performance overview

To study the behaviour of each TPC-C transaction in CPU only, GPU only and Hybrid processor we evaluated the execution time of each transaction for each scenario. Figure 6.6 on the next page shows the execution time of each implemented TPC-C transaction for GPU based system, CPU based system and hybrid CPU/GPU based system for workload 8 respectively. We can see from the figure that all the transactions have shown improvement in a hybrid system. Even though the performance of each transaction in a hybrid system depends heavily on the workload distribution but still we can get a rough idea about the performance of different operator on CPU and GPU device. New Order and Payment transaction contains Insert and Update operation respectively. By comparing the execution time of these two transaction with CPU only and GPU only system we can say that a speed up of upto 2x is achieved on insert operation and 3x for update operation in a GPU only system. Order Status transaction has multiple select operations, it performs better on a GPU only device by a factor of 6x. Delete operations performs exactly like a update operation as we only update the value of the primary key to 0. Hence, our evaluation results suggest that delete operations can also be performed faster on GPU by a factor of 3x.

6.3.4 Conclusion

A typical OLTP workload performs multiple Select, Update, Delete and Insert operations. We can see from our evaluation results that all OLTP operators are performed faster on a GPU only device. Our results also suggests that a proper load balancing can lead to a faster hybrid systems for OLTP workload. The performance of the complete TPC-C workload (only for implemented transaction) suggests that a hybrid CPU/GPU based systems can easily outperform a CPU based system by a factor of 2x which can be further improved by effective load balancing.

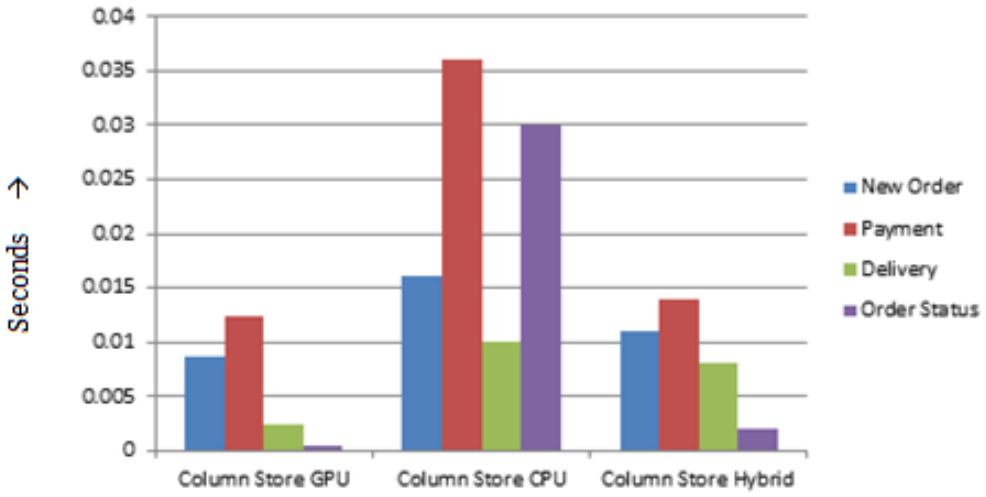

Figure 6.6: Comparison of Execution time of each transaction in CPU, GPU and Hybrid System

6.4 Threats to validity

In this section we are going to discuss the internal and external threats to validity

6.4.1 Threats to internal validity

For evaluation we performed thirty iterations for executing each workload in different scenarios like Column Store in a hybrid system, Column Store in a CPU only system etc. We did multiple iterations to get a mean value from a large set of result so that we can get more reliable results. This helped us to get more precise result for the reliability of our work.

For heavier workloads the GPU almost reaches its memory limits hence we can observe more variation in results for multiple iteration. Even though we have taken a mean of our results but still, we believe a GPU with more memory can provide more stable results in heavier workloads.

6.4.2 Threats to external validity

We are aware of the fact that using a standard benchmark like TPC-C do not automatically simulate a real time OLTP database. However, they are helpful to get a detailed evaluation of the behaviour of various OLTP workloads on systems of varied types. A typical TPC-C workload consists many insert, update, delete and select operation and few join operations, a change in the nature of the workload can effect the performance of the system.

The Join operation that we have implemented in our work does not perform the actual join operation on the OpenCL device as we calculate the joining indexes on the host program. Even though there are less Join operations in the TPC-C workload but still a properly implemented join operation can vary the performance of the system.

In this work we have not implemented Stock level transaction in the TPC-C benchmark which consists of join and select operation. Even though we can easily see from our evaluation that all operators gain highly from a hybrid system but still, the inclusion of stock level transaction will help to simulate a more life like OLTP system and evaluate better.

6.5 Conclusion

Through our research we were able to find the answers to our research question which will pave the way for finding the architectural properties and potentials of a hybrid CPU /GPU based system. It was evident from evaluation that a Column Store hybrid system can greatly benefit from the co-processing power of CPU and GPU. Our results showed that Column Store is a preferred storage mechanism for a hybrid CPU/GPU based system. A Column Store implementation benefits greatly from the architecture of GPU and is more space efficient. A hybrid system using a Column Store mechanism can outperform the CPU only Column Store OLTP system for all operators. A OLTP workload can benefit highly from a hybrid system if proper load balancing is done and emphasis is given on keeping a faster device like GPU always occupied.

7. Conclusion

Over the past years GPUs have shown tremendous speed up in the performance of OLAP workload. This can be highly attributed to the possibility of executing OLAP queries using a data parallel approach. In this work we have investigated the capabilities of hybrid CPU/GPU based systems for the execution of bulk OLTP workload. Column Store storage mechanism is a preferred storage mechanism in state of the art GPU based databases for OLAP scenario as it provides coalesced memory access and supports compression. In traditional CPU based systems Row Store becomes a preferred choice for OLTP query execution as OLTP queries have multiple attribute access. In our work we have done research to find out the better performing storage mechanism in a hybrid CPU/GPU based OLTP system. We first started our research by finding the best storage mechanism for an OLTP workload in a hybrid CPU/GPU based systems by comparing the performance between Row Store and Column Store implementation of our work. After finding this answer we tried to investigate the capabilities of a hybrid CPU/GPU based system for executing a bulk OLTP workload.

There have been previous researches like GPUTx which have shown that a GPU only system can speed up the query execution time of an OLTP workload. In our work we have further investigated the possibilities of using hybrid system for OLTP workload execution. One of the major contribution of this research is the implementation of OLTP transactions using operators unlike GPUTx where a OLTP transaction is executed as a single task. Operator based approach made it possible for us to investigate multiple kernel handling overheads. It also made our implementation independent to the transaction nature and schema as our operators can be reused irrespective of the nature of query and schema in use. This work will take us one step closer to finding the right architecture and capabilities for a hybrid OLTP system.

For this work we have done two separate implementation for Row Store and Column store storage mechanism. We have used the standard TPC-C database and its transaction to do the evaluation. To find the better storage mechanism we have executed workloads of

varying sizes on both the implementation. The better performing storage mechanism was chosen based on rigorous evaluation of the execution time of different workloads for both the scenario. The capabilities of a hybrid CPU/GPU based system is calculated by comparing it against CPU only and GPU only systems. The comparison is made by executing TPC-C workloads of varying sizes on each system and comparing the execution time.

Our results clearly showed that Column Store is a better storage mechanism as compared to Row Store for executing bulk OLTP workloads in a hybrid CPU/GPU based system. For heavier workloads the Column Store implementation outperformed the Row Store implementation by a factor of 4x. This can be largely attributed to the efficient use of the GPU memory model by better coalesced memory access. Factors like projection of only used columns and lightweight kernels also make Column Store a preferred choice. Apart from it there are some proven factors like vectorization and data compression which makes Column Store a preferred choice.

The Hybrid CPU/GPU based systems were able to outperform the CPU only system by a factor of 2x for heavier workloads. The GPU only system clearly outperformed the hybrid systems by a factor of 1.5x for heavier workloads. We attribute the under performance of our hybrid system against GPU only system to the improper utilization of GPU device. The fast performance of GPU device clearly indicates that our workload distribution between CPU and GPU was not sufficient to fully utilize the potentials of a hybrid system. Apart from it there are various characteristics of OLTP workloads like multiple access to critical data and uneven memory access which stops the OLTP workload to benefit heavily from GPU style of processing.

Through this research we have identified the storage structure needed and the capabilities of a hybrid CPU/GPU base system for OLTP workload. This work is a prototypical implementation which can be used as a starting point for creating a hybrid OLTP system. There are various aspects in GPU programming which have not been considered in our work like data transfer overheads and index calculation overheads which needs future research. Also we have identified the gaping holes in our implementation in workload distribution between CPU and GPU. We need to identify a better workload distribution mechanism to effectively utilize the hybrid systems. Some of the optimization techniques like better coalesced memory access and vectorization needs to be investigated as they can greatly boost the performance of a hybrid CPU/GPU based system.

8. Future Work

In our work we have tried to demonstrate the potential and limitations of using a hybrid CPU/GPU based system for an OLTP workload. Due the scope of this thesis and time limits there are many perspectives which have not been considered in our work. We have also identified various aspects that need to be improved in our work. In this chapter we have described all the limitations in our work that can lead to an improved system and can lead towards further evidences to support our work.

1. Coalesced Memory Access: We have identified the problem of coalesced memory as the biggest bottleneck for slow performing hybrid CPU/GPU System. To overcome this limitation we need to have sorted indexes which can lead to a coalesced memory access. Although we are assuming that sorting can be a potential bottleneck but still this approach can be further investigated.

2. Inter transaction Parallelism for better operator placement: In this work all the implemented transactions in TPC-C workload have been executed sequentially in a data parallel approach. During the execution of a single transaction mostly a homogeneous set of operations are performed. Due to homogeneous nature of operations we do not have the choice to do effective operator placement as the best device cannot always be chosen for the placing the operator as it can lead to ineffective load balancing. Performing multiple transactions in parallel will provide heterogeneous operators that can be effectively scheduled to better performing devise. We assume than intra transaction parallelism needs to be investigated as it can further speed up the performance of heterogeneous CPU/GPU based systems.

3. Better Load Distribution: In our work the biggest limitation that led to the slower performance of hybrid CPU/GPU based system as compared to GPU only was due to improper load balancing. We tried to distribute the load based on the performance of the devices but still, the GPU device was left task hungry which led to a slower hybrid system. We strongly recommend that a proper study about the load distribution is needed as it is of paramount importance for the performance of hybrid system.

4. Memory Overrun: In our work we have limited the number of transactions based on the available memory in OpenCL devise s. Our Implementation do not deal with the scenario of memory overrun. In real time OLTP systems the size of the workload should not be a limiting factor as a large volume of OLTP transactions are performed at a single time. In state of the art GPU based systems; the issue of memory overflow is handled by aborting the unfinished transactions on GPU and performing then on CPU. We assume that a successful hybrid CPU/GPU based system should handle the issue of memory overrun efficiently. Hence, a detailed investigation of this topic is needed for future works.

5. Data Compression: Data Compression is an efficient methodology that can use the limited memory of external devices like GPU to execute heavy workload. It can greatly reduce the problem of memory overrun. There are lots of light weight compression techniques like Dictionary Encoding; Run Length Encoding etc. which have shown good results of a GPU based system. These compression techniques need to be investigated on hybrid OLTP system for improved hybrid database in future.

6. Dynamic Query Plan Generation: In our work we have implemented static query plans that can only perform TPC-C Workload. These query plans cannot be re-used for other transactions. A real time OLTP system should be able to generate efficient query plans dynamically so that it can serve any transaction on the fly.

7. Dynamic Load Balancing In our work we have done static load balancing to provide both CPU and GPU enough tasks for execution. We used the existing knowledge about the data access patterns in TPC-C transactions to perform load balancing. A real time OLTP system should be able to perform dynamic load balancing and operator placement to effectively utilize both devices.

8. Vectorization: Vectorization is the capabilities of certain hardware to allows the processing of multiple instructions of the same nature in a single cycle. The task of a single Work Item in OpenCL devices can be vectorized so that multiple operations can be performed in a single Work Item. Although in the CPU it can lead to performance gain but in devices like GPU its performance needs to be investigated due to different memory hierarchy. Multiple instructions will lead to multiple the accesses to the Global Memory and this can be a potential bottleneck. The Vectorized approach of kernel implementation needs to be investigated as it can improve the performance of OLTP systems. The Vectorization approach can only be used with the Column Store storage mechanism as the OpenCL kernels of Column Store implementation perform instructions on a single data type.

9. Data transfer overhead consideration: In our work we have only calculated the kernel execution times for execution of TPC-C transaction. We had an assumption that data is already placed in OpenCL device memories, hence data transfer overheads in external device like GPU has not been considered. In a real life OLTP scenario, it is not possible to pre-load the data into the devices as the consistency of the data would be hard to maintain. Hence, for realizing a real life OLTP system we assume that the transfer overheads should be taken into consideration and evaluated.

10. Critical Data Access: In our implementation of TPC-C database we have used only one warehouse along with 10 district. During a bulk execution lots of transactions can try to update the information of the Warehouse or District which makes these records as critical data. In our work we have scheduled performed operations on critical data on the host program as they can not be performed in a data parallel way. We believe that in future works the overheads due to the access of critical data needs to be investigated as it is an important aspect of OLTP workload.

Bibliography

[BBHS14] David Broneske, Sebastian Breß, Max Heimel, and Gunter Saake. Toward hardware-sensitive database operations. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 229–234, 2014. (cited on Page 1)

[BBS14] David Broneske, Sebastian Breß, and Gunter Saake. Database scan variants on modern CPUs: A performance study. In *VLDB Workshop on In Memory Data Management (IMDM)*, volume 8921 of *LNCS*, pages 97–111. Springer, 2014. (cited on Page 39)

[BFT16] Sebastian Breß, Henning Funke, and Jens Teubner. Robust query processing in co-processor-accelerated databases. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1891–1906. ACM, 2016. (cited on Page 13 and 14)

[BGW+08] Edward W Bethel, Luke J Gosink, Kesheng Wu, Edward Wes Bethel, John D Owens, and Kenneth I Joy. Bin-hash indexing: A parallel method for fast query processing. Technical report, Ernest Orlando Lawrence Berkeley National Laboratory, Berkeley, CA (US), 2008. (cited on Page 7 and 8)

[BKH+14] Sebastian Breß, Bastian Köcher, Max Heimel, Volker Markl, Michael Saecker, and Gunter Saake. Ocelot/hype: optimized data processing on heterogeneous hardware. *Proceedings of the VLDB Endowment*, 7(13):1609–1612, 2014. (cited on Page 18)

[Bre15] Sebastian Breß. *Efficient Query Processing in Co-Processor-accelerated Databases*. PhD thesis, University of Magdeburg, Germany, 2015. (cited on Page 6 and 7)

[Bro15] David Broneske. Adaptive reprogramming for databases on heterogeneous processors. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15 PhD Symposium, pages 51–55, New York, NY, USA, 2015. ACM. (cited on Page 14)

[BS13] Sebastian Breß and Gunter Saake. Why it is time for a hype: A hybrid query processing engine for efficient GPU coprocessing in DBMS. *Proceedings of the VLDB Endowment*, 6(12):1398–1403, 2013. (cited on Page 13 and 17)

[CD97] Surajit Chaudhuri and Umeshwar Dayal. An overview of data warehousing and olap technology. *SIGMOD Rec.*, 26(1):65–74, March 1997. (cited on Page 1)

[Cen] Texas Advanced Computing Center. 8 things you should know about gpgpu technology. https://www.tacc.utexas.edu/documents/13601/88790/8things.pdf. Accessed July, 2016. (cited on Page 6)

[fc] fujitsu co. White paper: Benchmark overview tpc-c. https://sp.ts.fujitsu.com/dmsp/publications/public/benchmark_overview_tpc-c.pdf. Accessed Oct 2, 2003. (cited on Page 9)

[Gho12] Pedram Ghodsnia. An in-gpu-memory column-oriented database for processing analytical workloads. *VLDB 2012 PhD Workshop*, 2012. (cited on Page 2)

[GLW+04] Naga K. Govindaraju, Brandon Lloyd, Wei Wang, Ming Lin, and Dinesh Manocha. Fast computation of database operations using graphics processors. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, pages 215–226, New York, NY, USA, 2004. ACM. (cited on Page 41)

[HLY+09] Bingsheng He, Mian Lu, Ke Yang, Rui Fang, Naga K Govindaraju, Qiong Luo, and Pedro V Sander. Relational query coprocessing on graphics processors. *ACM Transactions on Database Systems (TODS)*, 34(4):21, 2009. (cited on Page 17)

[HSP+13] Max Heimel, Michael Saecker, Holger Pirk, Stefan Manegold, and Volker Markl. Hardware-oblivious parallelism for in-memory column-stores. *Proceedings of the VLDB Endowment*, 6(9):709–720, 2013. (cited on Page 14)

[HY11] Bingsheng He and Jeffrey Xu Yu. High-throughput transaction executions on graphics processors. *Proceedings of the VLDB Endowment*, 4(5):314–325, 2011. (cited on Page 1, 2, and 18)

[HYF+08] Bingsheng He, Ke Yang, Rui Fang, Mian Lu, Naga Govindaraju, Qiong Luo, and Pedro Sander. Relational joins on graphics processors. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 511–524. ACM, 2008. (cited on Page 5 and 6)

[KHL15] Tomas Karnagel, Dirk Habich, and Wolfgang Lehner. Local vs. global optimization: Operator placement strategies in heterogeneous environments. *Computing*, 1:O2, 2015. (cited on Page 1 and 2)

[KKN+08] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan PC Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, et al. H-store: a high-performance, distributed main

memory transaction processing system. *Proceedings of the VLDB Endowment*, 1(2):1496–1499, 2008. (cited on Page 2)

[LD93a] Scott T Leutenegger and Daniel Dias. *A modeling study of the TPC-C benchmark*, volume 22. ACM, 1993. (cited on Page 21 and 23)

[LD93b] Scott T. Leutenegger and Daniel Dias. A modeling study of the tpc-c benchmark. *SIGMOD Rec.*, 22(2):22–31, June 1993. (cited on Page 23 and 24)

[MBS15] Andreas Meister, Sebastian Breß, and Gunter Saake. Toward gpu-accelerated database optimization. *Datenbank-Spektrum*, 15(2):131–140, 2015. (cited on Page 7)

[OAD14] Bogdan Oancea, Tudorel Andrei, and Raluca Mariana Dragoescu. Gpgpu computing. *arXiv preprint arXiv:1408.6923*, 2014. (cited on Page 16)

[OvO] Oltp vs olap. http://datawarehouse4u.info/OLTP-vs-OLAP.html. (cited on Page 11 and 12)

[SB] TU Dortmund Sebastian Breß. Co-processor accelerated data management(lecture slides). Summer Term, 2015. (cited on Page xi, 10, 15, 26, and 27)

[Sca] Matthew Scarpino. A gentle introduction to opencl. http://www.drdobbs.com/parallel/a-gentle-introduction-to-opencl/231002854. Accessed August, 2011. (cited on Page 16)

[SMA+07] Michael Stonebraker, Samuel Madden, Daniel J Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The end of an architectural era:(it's time for a complete rewrite). In *Proceedings of the 33rd international conference on Very large data bases*, pages 1150–1160. VLDB Endowment, 2007. (cited on Page 2 and 17)

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Magdeburg, den