Database Query Execution on Heterogeneous Architecture

A PROJECT REPORT

SUBMITTED IN PARTIAL FULFILLMENT OF THE

REQUIREMENTS FOR THE DEGREE OF

Master of Engineering

IN

Computer Science and Engineering

BY

Vinay Kumar Rijhwani

Computer Science and Automation
Indian Institute of Science
Bangalore – 560 012 (INDIA)

June, 2017

Declaration of Originality

I, **Vinay Kumar Rijhwani**, with SR No. **04-04-00-10-41-15-1-12203** hereby declare that the material presented in the thesis titled

Database Query Execution on Heterogeneous Architecture

represents original work carried out by me in the **Department of Computer Science and Automation** at **Indian Institute of Science** during the years **2015-17**.
With my signature, I certify that:

- I have not manipulated any of the data or results.
- I have not committed any plagiarism of intellectual property. I have clearly indicated and referenced the contributions of others.
- I have explicitly acknowledged all collaborative research and discussions.
- I have understood that any false claim will result in severe disciplinary action.
- I have understood that the work may be screened for any form of academic misconduct.

Date: Student Signature

In my capacity as supervisor of the above-mentioned work, I certify that the above statements are true to the best of my knowledge, and I have carried out due diligence to ensure the originality of the report.

Advisor Name: Jayant R. Haritsa Advisor Signature

DEDICATED TO

My Family and Friends

Acknowledgements

I would like to express my sincere gratitude to Prof. Jayant R. Haritsa for his unmatched guidance and supervision. I have been extremely lucky to have work with him.
I am thankful to Srinivas Karthik for its assistance and guidance. It had been a great experience to work with them. I would like to thank the Department of CSA for providing excellent study environment and infrastructure. My sincere thanks goes to my fellow lab mates for all the help and suggestions. Also I would like to thank my CSA friends for their warm friendship and for adding an amazing experience in my life.
Finally, I am indebted with gratitude to my parents and sister for their love and inspiration. This project would not have been possible without their constant support and motivation.

Abstract

Graphics processors (GPUs) have emerged as a powerful co-processor for general-purpose computation. GPUs have been very widely used for various applications including database query execution. GPUs have an order of magnitude higher computation power as well as the memory bandwidth but they have a bottleneck of limited GPU device memory and transferring data between CPU main memory and GPU device memory. The database optimizer generates a query plan i.e. a tree structure which contains operators as nodes and edges represent the dependency between the nodes. The problem is how to divide the nodes of this tree-structured query plan, given by MonetDB[1], into the heterogeneous environment (CPU and GPU). We have proposed greedy approach and dynamic programming approach for the chain trees i.e., trees in which nodes have exactly one child. We have also verified that HEFT (Heterogeneous Earliest Finish Time) algorithm[14] works very well for non-chain trees i.e., trees in which nodes have many child nodes. We have also implemented operators for hybrid environment (CPU-GPU) i.e., running a single operator in the hybrid environment by using data parallelism. The thesis also includes the resource configurable implementation of filter and hash join operators.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

GPUs have evolved into general purpose computing with the emergence of efficient parallel programming models, such as CUDA[2]. The use of such heterogeneous computing devices is highly recognized as the only promising way to achieve application speedups. GPUs high computational power gives us the possibility of accelerating execution of database queries. However, there is still a significant challenge for using GPUs in the most effective manner to speed up the database query execution. The database query (Query 1) is converted into execution plan tree, as shown in Figure 1.1, which is given by the database query optimizer. In this plan tree, relations are specified in ellipses and rectangles show the operators at the different level. The optimizer finds this optimal plan tree such that the execution time of the query is minimal. An optimizer uses its cost model to estimate the cost of different query plans generated for a query. This cost model give its estimated cost based on the input selectivity (number of tuples in the input), organization of data on the disk, indexes available, the operator algorithm and other metadata. The database query optimizer used by us is the one given by MonetDB[1] as it uses columnar storage. The plan given by MonetDB is a CPU-optimized plan and we are using this plan to run it in a heterogeneous environment. The plan given by MonetDB is in the form of MAL (MonetDB Assembly Language)[1] instruction and we are converting these instructions into the tree-structured plan. The problem we are trying to solve is the optimal assignment and scheduling of plan nodes into the CPU and GPU so as to get the minimum execution time.

```
Select sum(l_extendedprice) as sum , l_orderkey
From lineitem, orders, customer
Where l_orderkey = o_orderkey and o_custkey = c_custkey
and c_acctbal ≥ 5000 and o_totalprice ≥ 10000
group by l_orderkey;
```

Query 1

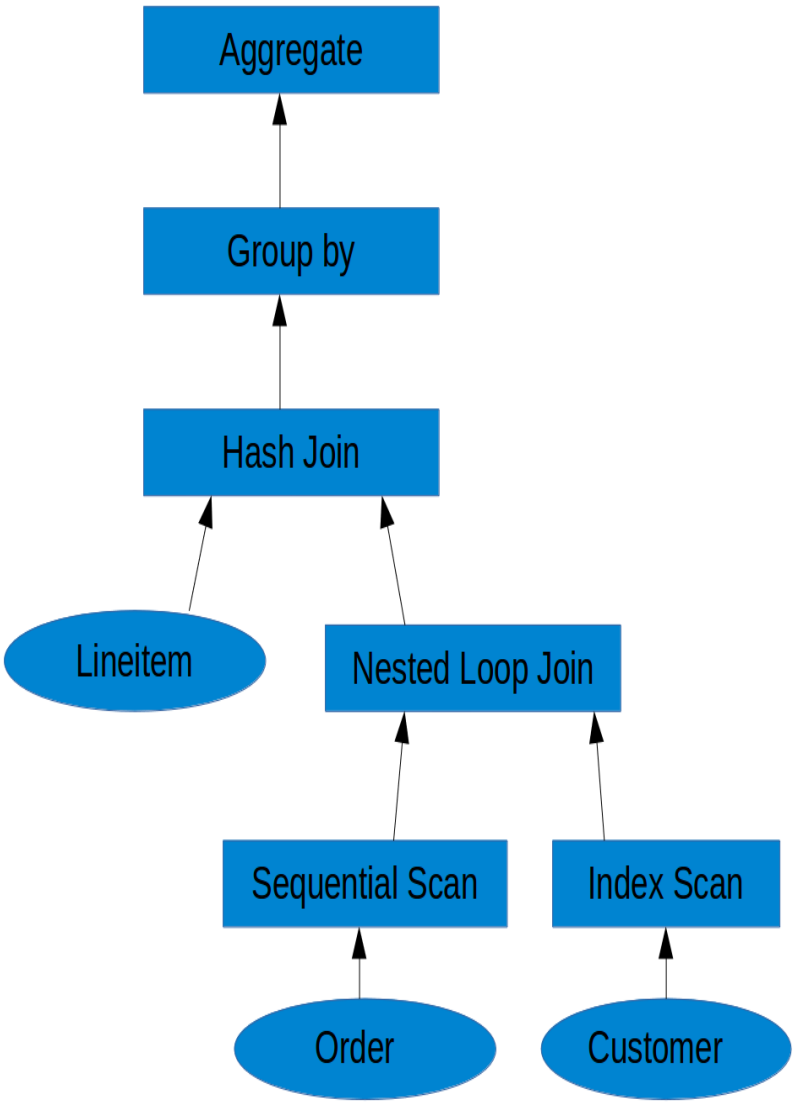

Figure 1.1: Plan Tree

In this report we have given two approaches for solving this problem for chain trees- greedy scheduling and dynamic programming scheduling. The dynamic programming scheduling gives

the optimal assignment of nodes into CPU or GPU which works slightly better than the greedy approach but the amount of time taken to output the scheduling is much more in case of dynamic programming approach. For non-chain trees, we have verified that the execution of the plan in CPU-GPU environment such that the assignment and scheduling of the plan nodes is done by using HEFT (Heterogeneous Earliest Finish Time) algorithm[14] works very well as compared to CPU only execution. Until now the operator was either running in CPU or GPU but we have also explored the prospect of running intra-parallelism operators i.e. running a operator in CPU-GPU environment. The operators are implemented to use both CPU and GPU in single run by dividing the amount of data processed by that operator. This gives better performance for queries where it's not possible to use both CPU and GPU simultaneously to execute the query as in case of chain tree structured plans. It can also be used for certain nodes in non-chain plan trees if one of CPU or GPU is sitting idle.

Chapter 2

Background and Preliminaries

The database storage in our system is similar to that of MonetDB[1]. There is a significant deviation in the storage model of MonetDB from traditional database systems. The relational tables are represented using vertical fragmentations, by storing each column in a separate (id,value) table, also called a BAT (Binary Association Table). These BATs are stored in the separate files. All the intermediate results are also stored in the form of BATs. The reasons for choosing MonetDB for comparison as well as the storage model are following -

- Before executing the algorithm on the GPU, we have to transfer input data from the CPU host memory to the GPU device memory. The transfer operation from CPU main memory to GPU device memory and vice versa is costly as it happens on limited bandwidth PCIe bus. The columnar storage aids us to only send the required part of relation to the device memory hence reducing the transfer cost.
- It is an in-memory database. For executing any operator on GPU, the data should be in GPU device memory as GPU cannot directly access directly main memory to do disk I/O. Therefore it is necessary to consider that the database is present in the main-memory.
- Since less data is required for any operator, better cache and shared memory optimizations are possible.

2.1 Background on GPU

GPUs are originally designed as co-processors for CPUs to process graphics tasks. In recent years, they have evolved into a powerful accelerator for many applications such as High-Performance Computing (HPC) and deep learning. Though the detailed components of GPUs

from different vendors vary, their general architectural designs can be abstracted in the same model as shown in Figure 2.1. For illustration purposes, we use the terminology of Nvidia GPU. GPU consists of many streaming multiprocessors(SMs) and each SM consists of many SIMD (Single Instruction Multiple Data) Lanes. At any given clock cycle, all SIMD lanes of a particular SM execute the same instruction but operates on different data. The GPU supports thousands of concurrent threads. The threads on each SM are organized into groups called as warps. Warps are dynamically scheduled on SMs. Each thread in a particular warp (typically group of 32 threads) execute same instruction at a clock cycle. When a warp is scheduled on the SM, then each thread in a warp gets one SIMD lane to get executed and as there are as many numbers of SIMD lanes as a number of threads in a warp and all of them can execute at the same clock cycle.

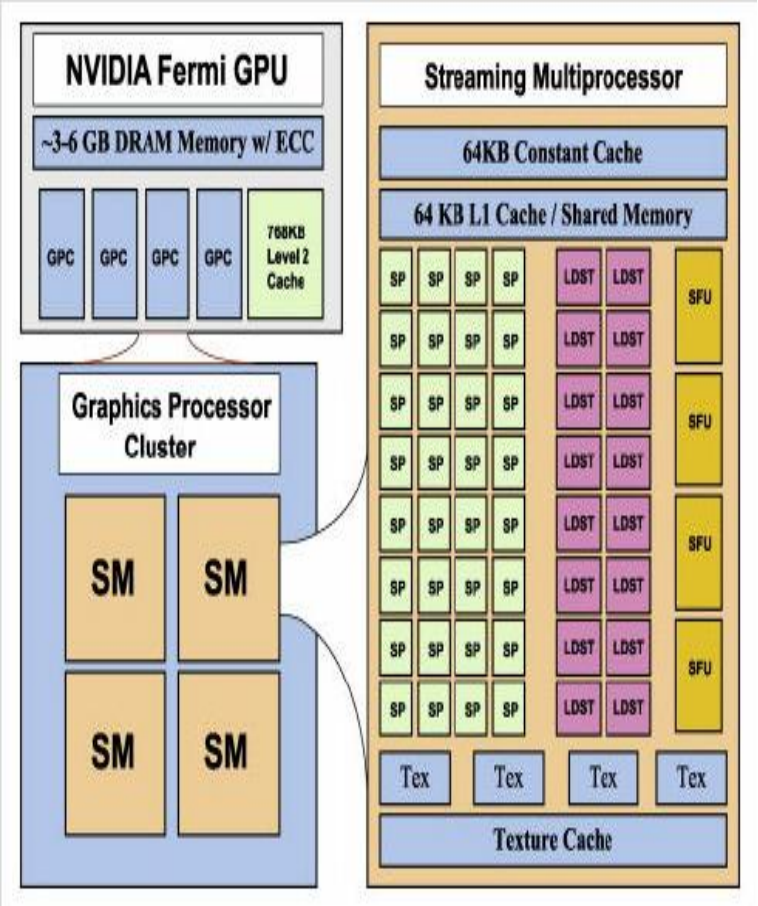

Figure 2.1: GPU Architecture

The kernel is a basic function which executes on GPU. It can be written in CUDA[2] or OpenCL[3]. Whenever the host thread launches (calls) the kernel, it is scheduled on GPU and gets executed on the GPU with control returning to the host thread i.e launching of the kernels by the host thread on the GPU is non-blocking. Therefore host thread could launch multiple kernels even though the first kernel has not even started execution. We can specify the number of threads that we want to launch for that kernel. The number of threads is specified in the form of grid size and the number of threads. The grid is the collection of thread blocks whereas each thread block is the collection of threads.Each thread runs the instructions in the kernel independent of each other, therefore each thread requires their own set of registers for the variables. The programmer could write the kernel in such a way that each thread can allocate memory for the variables on the on-chip cache (shared memory) so that the thread does not have to access the device memory repeatedly for the variable. Shared memory is allocated on a per thread block basis, therefore, all the threads belonging to same thread block can access shared memory allocated for that thread block. Each SM has fixed a number of registers, fixed amount of shared memory, which is a software controlled cache in contrast to CPU which has hardware control cache. Each SM can have at the most certain number of threads and thread block to execute. A thread block of a kernel is a schedulable unit on SMs i.e. either all the threads of a thread block are scheduled on SMs or none of them. Once the thread block is scheduled to SM, it will not be preempted from that SM until all of its threads have completed their execution.

Stream

The stream is a sequence of operations that execute in issue-order on the GPU. The operation in a stream start execution only when earlier operations in that stream have completed their execution. The operations in different streams may run concurrently or they may be interleaved. The operation could be a kernel execution or a memory transfer operation from host to device memory or from device to host memory. We are using streams extensively for asynchronous data transfers between these memories as well as for concurrent kernel executions in cases where it is possible.

In figure 2.2[8], kernel K_1, P_1 and Q_1 can be executed concurrently while kernel K_2 will only be scheduled when K_1 is executed i.e. when all the thread blocks of kernel K_1 are executed. A stream is said to be executed when all its kernels have completed their execution. A stream is said to be activated when it is ready to schedule its front kernel on the GPU. In Figure 2.2, stream 1 is ready to schedule kernel K_1 on the GPU. Once the GPU scheduler has scheduled all the thread blocks of kernel K_1 on the GPU then Stream 1 will be deactivated as it is not

ready to schedule its kernel K_2 on the GPU. This is because stream 1 is waiting for kernel K_1 to be executed. Streams can be created in the following two ways:

1. By calling CUDA function *cudaStreamCreate* explicitly which launches the kernel in different stream.

2. By creating POSIX threads, each thread will have its respective default stream. Kernels launched by different pthreads will execute in different streams.

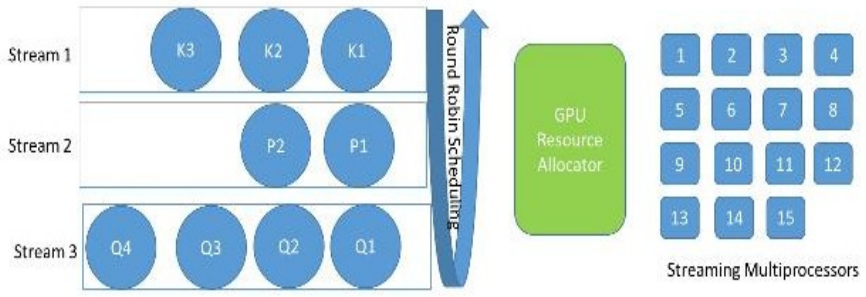

Figure 2.2: Stream Execution

Chapter 3

Related Work

GPUs have been very widely available for last decade which resulted in data co-processing on these GPUs. Govindaraju et al.[10][9], first presented their work on relational operators running on graphic cards. They investigated relational query processing by implementing complete set of database operators for the GPUs as well as for CPU so that their storage model i.e. columnar storage could be run on this system. We are also using these operator implementations in our system. Manegold et al. developed a hardware-oblivious system, Ocelot[11], which can execute a complete query on either CPU or GPU but there is no CPU-GPU heterogeneous computing involved. There has been some other work[13][6] which mostly talks about query executions on GPU.

Chapter 4

Database Operators on GPU

The implementation of various database operators on CPU and GPU has been taken from the work presented by Govindaraju et al.[10][9]. These operators are implemented for CPU using openmp[4] and for GPU using CUDA[2]. Following are the implementation details of various operators on GPU-

Selection

The selection operator outputs a subset of tuples from the relation based on a certain condition i.e. tuples which satisfy the condition remains in the output and others are discarded. Selection is implemented using map, prefix scan and scatter primitives. First, the map primitive processes the input tuples (array containing the column values) that results in the corresponding 0-1 array (0 represents the column value does not satisfy the condition and 1 represents the column value satisfy the condition). Then the prefix sum is computed on that array and stored into another array. Lastly the result is scattered using scatter primitive to the output array according to the previous result arrays.

Projection

The projection operator extracts a column of the relation based on the given record IDs. It is implemented using gather primitive. A gather primitive performs an indexed reads from a relation. The read location for each output tuple is defined by a location array. It uses sorting if the elimination of the duplicates is required.

Order-by

The order-by operator is implemented using sort primitive specifically quick sort. First, the algorithm divides the relation into multiple chunks using a given set of randomly chosen pivots. This splitting process goes recursively until the chunk size is smaller than the local memory size. Lastly these chunks are sorted in parallel using the bitonic sort method.

Group-by and Aggregation

The sort primitive is used for grouping and reduce primitive is used for aggregation. The sorting on GPU is explained in the previous paragraph. The reduce operation computes a value based on the given input relation. It can be used to compute the sum, average, maximum and minimum of all the key values in the relation. Again this reduce primitive is implemented as a multi-pass algorithm for better utilization of local memory. In each pass, the input data is divided into multiple chunks and these chunks are evaluated in parallel. The size of the chunk is equal to the local memory size which improves temporal locality.

Join

The join methods used in our system are hash join and nested loop join. In the nested loop join, each thread block performs the join on a smaller chunk of the relations R and S (relations involved in the join). The size of these chunks is again equal to the local memory size. Each thread of the thread block performs join of one tuple of relation R to all the tuples of relation S. The hash join on GPU is implemented as a parallel version of the radix hash join[16]. The hash join works in two phases- Firstly, both R and S are divided into the same number of chunks using radix bits partitioning. This partitioning is done in such a way so that most of the S partitions fits into the local memory. In the second phase, these partitions of R and S are joined in parallel using nested loop join method by making smaller relation as the inner relation.

4.1 Cost Estimations

The cost model used for estimating the cost of operators in the GPU is the one given by Govindaraju et al.[9]. It involves cost of transferring the data from CPU host memory to GPU device memory and the cost of executing the operator. The transfer time can be calculated as the sum of two parts, the cost of invoking the transfer process and the cost of transferring the

data. Now, to estimate the correct cost of executing an operator in GPU is very difficult since GPU computation is highly parallel. The cost model takes into consideration the access time of the local memory and the pure computation cost. The detail modelling of the cost model is given in the Govindaraju et al.[9].

Chapter 5

Problem Statement

A brief description of your chapter.

Given a query plan, which is produced by the query optimizer in the form of tree structure containing nodes as operators and edges represent the precedence relation, output the assignment and scheduling of these nodes for the heterogeneous environment consisting of CPU and GPU. The problem is divided into two parts- one is solving it for chain plan trees (Figure 3) and other is for non-chain plan trees (Figure 4). The query plan that we are using is the one given by MonetDB[1] optimizer. The MonetDB optimizer gives the plan in the form MAL (MonetDB Assembly Language) instructions. In MonetDB, queries are parsed into domain-specific representations and optimized. The generated logical execution plans are then translated into MonetDB Assembly Language (MAL) instructions, which are passed to the next layer. The MonetDB uses three layers of abstraction. The middle layer provides a number of cost-based optimizers for the MAL. The bottom layer is the database kernel, which provides access to the data stored in Binary Association Tables (BATs). Each BAT is a table consisting of an object-identifier and value columns, representing a single column in the database [12]. We are converting these plans present in the form of MAL instructions(Figure 5.1) into the tree structured plans(Figure 5.2).

```
function user.s6_1{autoCommit=true}(A0:int,A1:flt,A2:dbl,A3:dbl,A4:dbl):void;
    X_7 := sql.mvc();
    X_8:bat[:oid,:oid]  := sql.tid(X_7,"sys","lineitem");
    X_11 := sql.bind_idxbat(X_7,"sys","lineitem","lineitem_l_partkey_fkey",0);
    X_14 := algebra.leftfetchjoin(X_8,X_11);
    X_17 := sql.bind(X_7,"sys","part","p_retailprice",0);
    X_19 := X_17;
    X_20 := sql.bind(X_7,"sys","part","p_size",0);
    X_22 := X_20;
    X_15:bat[:oid,:oid]  := sql.tid(X_7,"sys","part");
    X_23 := algebra.subselect(X_22,X_15,A0,A0,true,true,false);
    X_25 := algebra.thetasubselect(X_19,X_23,A1,"<=");
    X_27 := X_25;
    (X_28,r1_30) := algebra.join(X_14,X_27);
    X_30 := sql.bind(X_7,"sys","lineitem","l_quantity",0);
    X_32 := algebra.leftfetchjoin(X_8,X_30);
    X_33 := algebra.leftfetchjoin(X_28,X_32);
    X_34 := batcalc.dbl(X_33);
    X_35:bat[:oid,:oid]  := X_15;
    (X_36,r1_41) := algebra.join(X_14,X_35);
    X_40 := algebra.leftfetchjoin(X_36,X_32);
    X_41 := batcalc.dbl(X_40);
    X_42 := sql.bind(X_7,"sys","part","p_partkey",0);
    X_45:bat[:oid,:int]  := algebra.leftfetchjoinPath(r1_41,X_35,X_42);
    (X_46,r1_55,r2_55) := group.subgroupdone(X_45);
    X_49:bat[:oid,:dbl]  := aggr.subavg(X_41,X_46,r1_55,true,true);
    X_51:bat[:oid,:dbl]  := batcalc./(X_49,A3);
    X_52:bat[:oid,:dbl]  := batmmath.floor(X_51);
    X_53 := algebra.thetasubselect(X_52,A4,">=");
    X_55 := algebra.leftfetchjoin(X_53,X_49);
    X_56:bat[:oid,:dbl]  := batcalc./(X_55,A2);
    X_57:bat[:oid,:dbl]  := batmmath.floor(X_56);
    (X_58,r1_74) := algebra.join(X_34,X_57);
    X_95 := algebra.leftfetchjoin(X_28,X_8);
    X_60 := sql.bind(X_7,"sys","lineitem","l_extendedprice",0);
    X_65:bat[:oid,:flt]  := algebra.leftfetchjoinPath(X_58,X_95,X_60);
    X_66 := algebra.selectNotNil(X_65);
    X_67:flt  := aggr.sum(X_66);
    sql.exportValue(1,"sys.L3","L3","real",24,0,9,X_67,"");
```

Figure 5.1: MonetDB MAL Plan

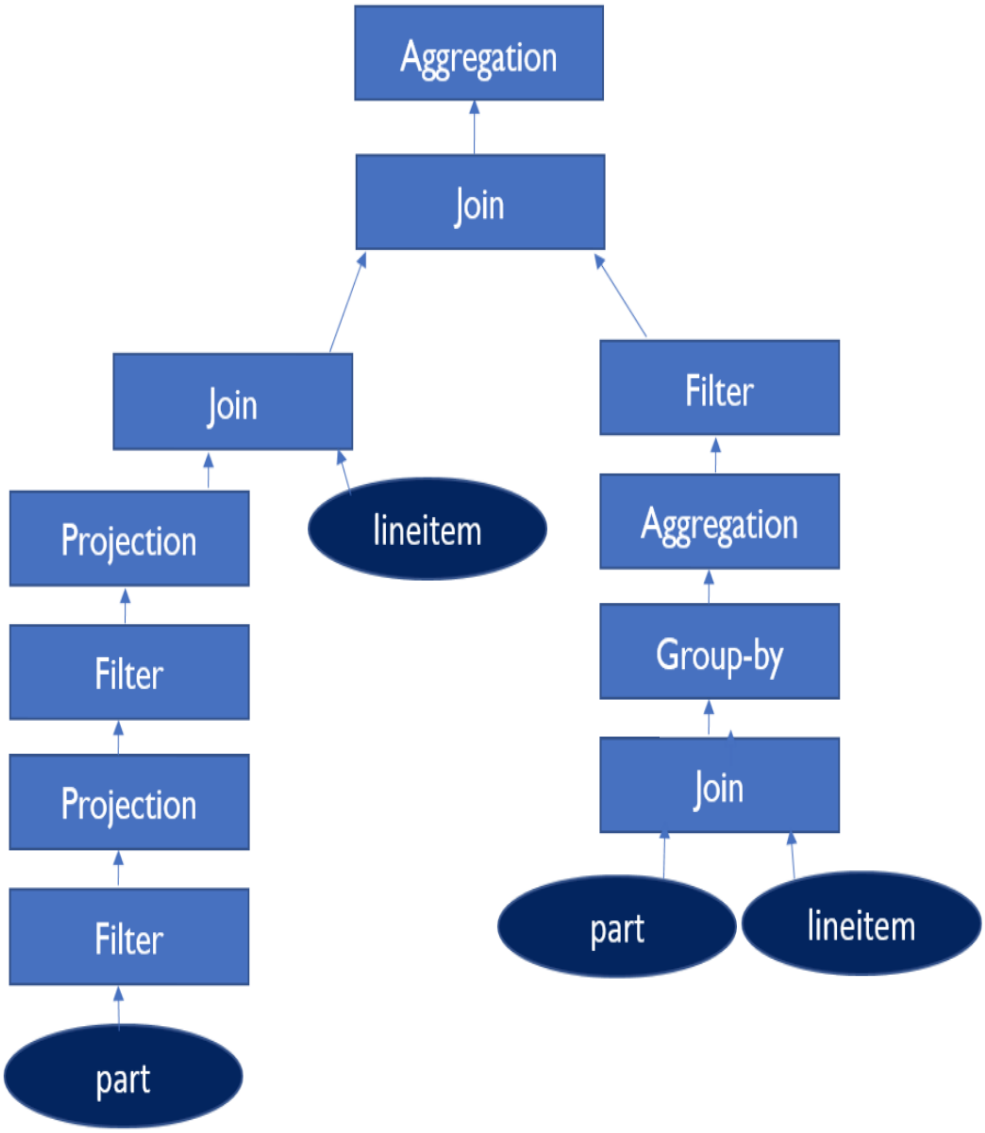

Figure 5.2: Tree Plan

Chapter 6

Chain Structure Plan Trees

The chain tree structured plan is shown in Figure 6.1. These are the trees in which nodes have exactly one child i.e. only one relation is involved in the database query, for example TPCH benchmark query 1. There are two possible ways of finding the schedule to this type of plan - dynamic programming approach and greedy approach. The dynamic programming approach gives the optimal assignment of the plan nodes whereas the greedy approach uses the greedy technique to find the assignment which may not be optimal but takes a lot lesser time than the dynamic programming approach to output the assignment and schedule.

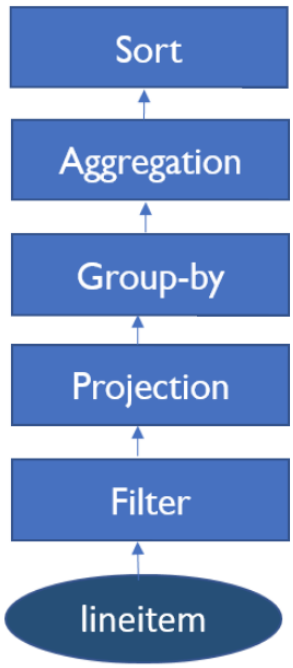

Figure 6.1: Chain Tree Structured Plan

6.1 Greedy Approach

The greedy approach takes as input a chain tree structured plan with the information of nodes and the precedence constraint between the nodes. It is also provided with the information of CPU execution time, GPU execution time of the nodes and the data transfer time between the CPU host memory and the GPU device memory, required if the node is executed in the other device. It is a bottom-up approach. It starts assigning the nodes to either CPU or GPU from the leaf node and goes in the upward direction. Since it is a greedy approach, it only looks for short term profits i.e. if the current node is executed in the CPU then the next node will be executed in the CPU if the CPU execution time of that node is lesser than the summation of GPU execution time and the data transfer time and vice versa.

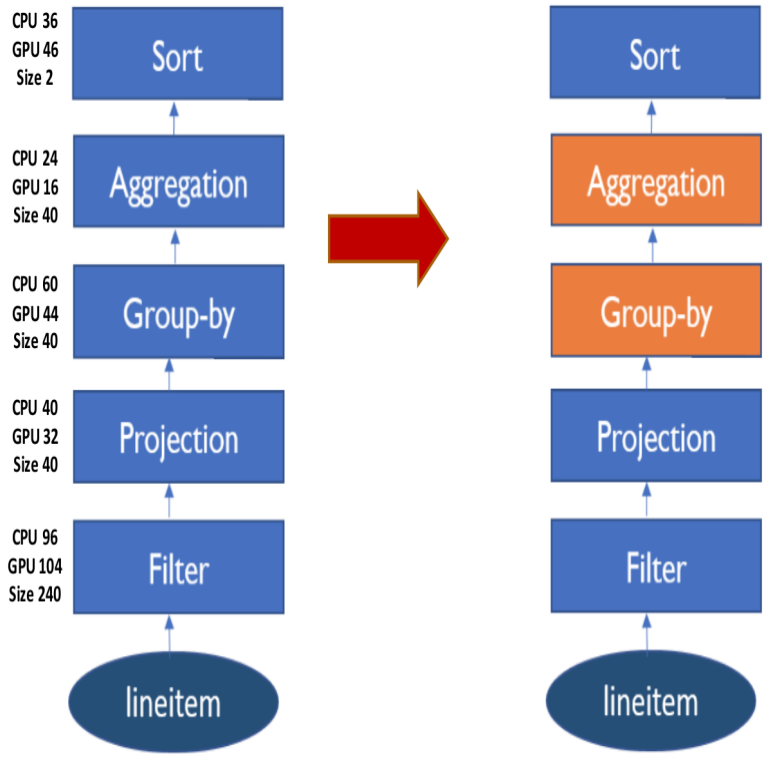

Figure 6.2: Greedy Approach

Given a hypothetical plan tree (Figure 6.2 left-hand side), showing various nodes representing operators and ellipse representing the relation. The CPU execution time (secs), GPU execution time (secs) and the amount of data processed by each node is given with the bandwidth of PCIe is 4 units/sec. The algorithm starts from the leaf nodes and the initial data is present in the main memory. The leaf node (Filter node) is assigned to the CPU since the CPU execution time (96 secs) is lesser than the summation of GPU execution time and data

transfer time (164 secs). Again the projection node is assigned to the CPU as the CPU execution time (40 secs) is lesser than the summation of GPU execution time and data transfer time (42 secs). Now the group-by node is assigned to the GPU as the summation of GPU execution time and the data transfer time is lesser than the CPU execution time. In the similar fashion, we traverse the tree in the upward direction and keep assigning the nodes to either CPU or GPU. The final output of the greedy algorithm is on the right-hand side of Figure 6.2 where blue nodes will be running on CPU and orange nodes will be running on GPU. The complexity of greedy algorithm is $O(n)$ where n is the number of nodes in the plan tree.

6.2 Dynamic Programming Approach

The dynamic programming approach is a technique where each possible solution is explored and the optimal assignment of the plan tree nodes is given as an output. The input to this technique is same as in the case of greedy approach. It basically forms a tree type structure to get the final output. It starts exploring nodes from the leaf nodes and at every node there exist two possibilities, either the node is executed by CPU or GPU. In this way, the dynamic programming approach iterates over all the nodes and finally outputs the optimal assignment which will take the least time to execute the given plan.

Given the same plan tree as in greedy approach, applying the dynamic programming approach will result in the assignment of the nodes into CPU or GPU, as given in Figure 6.3 right-hand side. The complexity of this approach is $O(2^n)$ which is very large as compared to greedy approach but the number of nodes in a plan tree having single relation is usually not very large so dynamic programming approach can be used. In this approach also, it starts from the filter node and recursively calls the function- one with considering it to be executed on CPU and other with GPU. It does that for all the nodes and measures the cost of each possible assignment and finally outputs the one with the least cost. The dynamic programming approach execution time is 234 secs whereas the greedy approach execution time is 242 secs so the difference is not significant in this example but this is the best possible way of executing the given plan in the CPU-GPU heterogeneous environment.

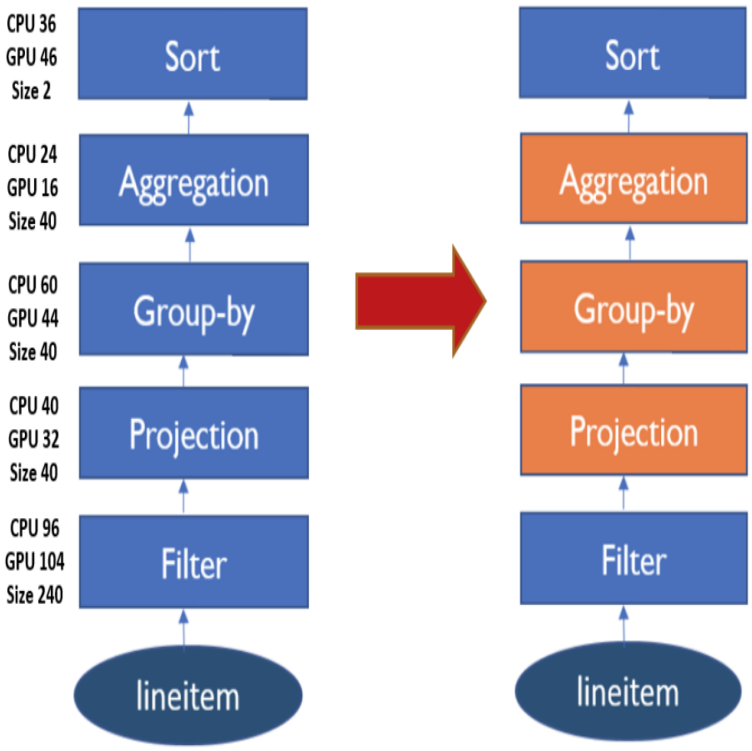

Figure 6.3: Dynamic Programming Approach

Chapter 7

Non-Chain Structure Plan Trees

The non-chain plan tree is shown in Figure 7.1. These are the trees in which the nodes can have more than one child i.e. more than one relation is present in the query. The problem of scheduling nodes of this type of tree structure into multiple devices can be mapped into the problem of scheduling task graphs[5]. The task graph represents the graph where nodes denote computational tasks, and edges model precedence constraints between tasks. The task graph scheduling is an activity that consists of mapping a task graph onto a target platform. For each of the task the algorithm decides the assignment (node on either CPU or GPU) and the scheduling (sequence of running these nodes on CPU and GPU). The aim is to get the most effective execution of the task graph.

The plan trees can be directly mapped into the task graphs in which operator nodes can be mapped into computational tasks and the edges represents precedence constraints. Each node label shows computation cost of the task for various resources (CPU and GPU) and each edge label shows the amount of data needed to sent from one node to other. The problem of scheduling task graph is a NP-Hard problem [15][7] so finding the optimal assignment and schedule for non-chain structured plan trees is also NP-Hard problem.

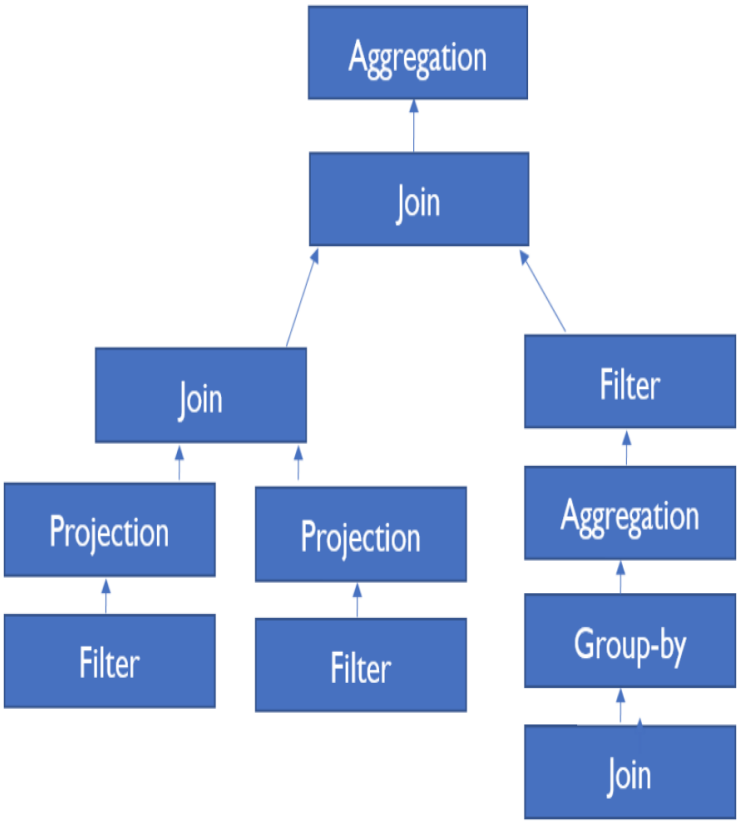

Figure 7.1: Non-chain Tree Structured Plan

7.1 HEFT (Heterogeneous Earliest Finish Time)[14]

HEFT[14] algorithm is a well-known technique used for scheduling task graphs for heterogeneous computing. We are using this HEFT technique for getting an assignment and scheduling of the nodes in the heterogeneous environment and have found it to be performing very close to optimal. The HEFT algorithm takes the following inputs-

- A graph $G = (V, E)$ where V is the set of n nodes and E is the set of e edges between the nodes. Each edge $(i, j) \in E$ represents the precedence constraint such that node n_i should finish its execution before node n_j.
- A $v \times v$ $Data$ matrix consisting of communication data, where $Data_{(a,b)}$ is the amount of data output by node n_a and sent to node n_b.
- A set R of r heterogeneous processors, a matrix B of size $r \times r$ consisting of data transfer rates and the communication startup cost s.

- A $v \times r$ matrix C where $C_{(i,j)}$ gives the estimated execution time of node n_i on processor j.

HEFT takes into consideration that the computation can also be overlapped with the communication and assumes that the intraprocessor communication cost is negligible as compared to interprocessor communication cost. The HEFT algorithm works in two phases- task prioritizing phase and processor selection phase. In task prioritizing phase, each node is given a priority which is based on the mean computation and mean communication costs and then the list of nodes is sorted on the basis of this priority. This ordering provides the topological order of nodes so preserving the precedence constraint among nodes. In the processor selection phase, the nodes are assigned to the processors by using insertion-based policy i.e. by inserting the node between two already scheduled nodes on the processor by looking at the idle time-slot between the scheduled nodes. Given the plan plan of Figure 7.1 and applying HEFT algorithm to get the assignment of nodes to either CPU or GPU and scheduling of the nodes. The computation cost of each node is shown in Figure 7.2 with the numbering of nodes as in Figure 7.3. The result is shown in Figure 7.3 where orange nodes will be running on CPU and blue nodes on GPU. The hand-tuned scheduling and assignment of nodes to CPU or GPU gives the same assignment as given by HEFT.

Node	CPU(ms)	GPU(ms)
1	165	150
2	130	144
3	110	95
4	80	104
5	2544	9857
6	1245	2147
7	347	114
8	67	61
9	987	3547
10	478	2475
11	144	75

Figure 7.2: Computation Costs

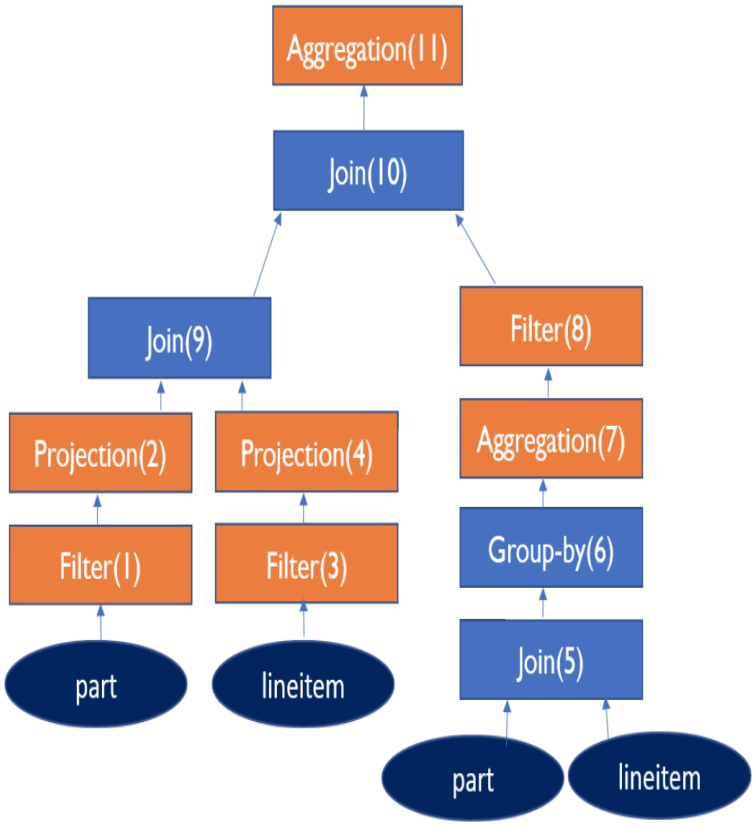

Figure 7.3: HEFT Algorithm

Chapter 8

Intra-Operator Parallelism

The problem with smaller queries and queries involving single relation is that one of CPU or GPU sits idle most of the time so we are exploring intra-operator parallelism in that aspect. The intra-operator parallelism means the same operator runs on CPU and GPU simultaneously. In this work, the data parallelism is used i.e. dividing the amount of data to be processed by that operator between CPU and GPU in such a way that the execution time in CPU is equal to the summation of GPU execution time and the data transfer time. The output given by GPU is transferred to CPU and merged with the output given by CPU and also if the next operator is required to be executed in GPU then the output of CPU is transferred to GPU device memory from host memory and the merging is done by the GPU.

Chapter 9

Resource Configurability

9.1 Resource Configurable Hash Join

The hash join implementation for the GPU was not resource configurable i.e. it is not possible to specify the amount of resources to give to various kernels involved in the hash join algorithm. The code base used for operators is the one given by Govindaraju [9]. Specifically, we want to fix the number of thread blocks assigned to these kernels so that multiple kernels can execute at the same time. The previous implementation (i.e. the implementation of hash join as given in [9]) resource allocation depends on the amount of data size (cardinality) of relations involved in the join. Hash join involves two phases- the first phase is called the build phase whereas the second phase is called probe phase. In the build phase, the hash table is build using one of the relations involved in the join, usually the smaller relation. The probe phase involves the scanning of the other larger relation and finding the matching tuples by looking at the hash tables of the smaller relation.

The hash join GPU implementation is the radix based hash join which partitions both the input relations. The partitioning is done using multiple passes until we get the smaller partitions such that these partitions are not bigger than the shared memory size of the GPU. In every pass, the previous implementation assumes to have that many thread blocks, for the kernel involved in partitioning, as the number of partitions that are needed to be processed. The partitioning is done using histograms and each thread build its own histogram on the data that it processes. Then those histograms are combined to create block level histograms and finally the global histograms pertaining to complete data. In previous implementation, the kernels were written assuming there is at least one thread block working on single partition and now the current implementation will have the same number of partitions but a single thread

block will be able to work on more than one partition. This has been done for all the kernels that were present in hash join implementation, the ones that are partitioning the data and the others that are doing the probing.

After the partition phase, the nested loop join is performed on the partition pair of the relations to get the desired result. The graph in Figure 9.1 shows the performance of resource configurable hash join after changing the number of thread blocks keeping the number of threads same as given by the previous implementation. The experiments are performed on Nvidia tesla K20X having specification as given in the following table.

Number of SMs	14
Cores per SM	192
Number of registers per SM	64K
Shared memory per SM	48K
Device memory	6GB
Max Number of threads per SM	2048
Max number of thread blocks per SM	16

Table 9.1: System Configurations Nvidia K20

The resource configurable hash join is performed on relations, both of which having 2 million tuples. The time taken by the algorithm is shown on the y-axis and the x-axis shows the number of thread blocks given to all the kernels involved in the algorithm. The number of threads per thread block is the same as chosen by the previous implementation as it better utilizes the shared memory.

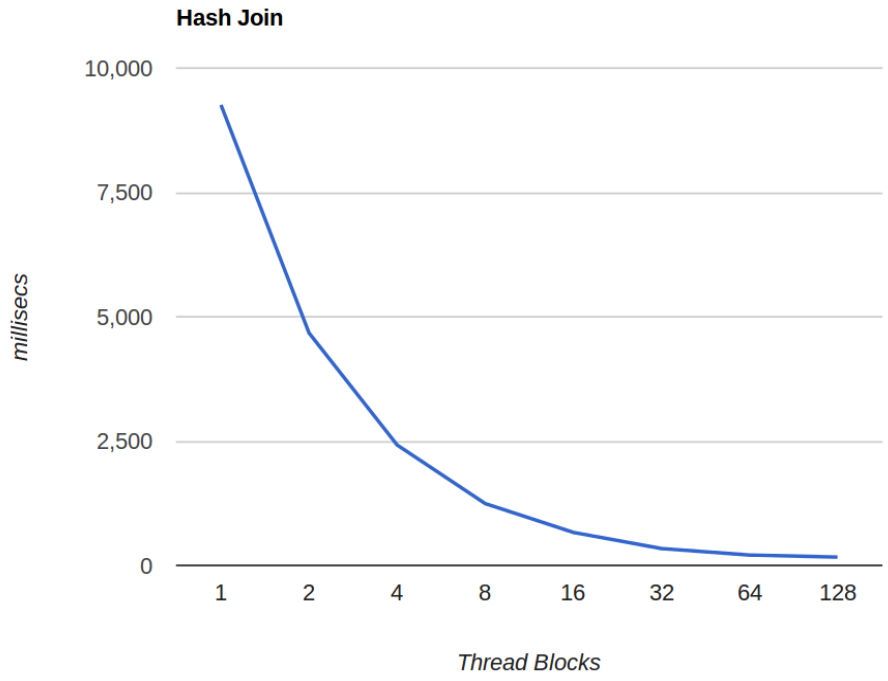

Figure 9.1: Resource Configurable Hash Join

9.2 Resource Configurable Prefix Scan

The prefix scan implementation (i.e. the implementation of prefix scan as given in [9]) was also not resource configurable. The prefix scan (also known as prefix sum) performed on an array of integers returns another array of same size in which each element is the sum of all the integers that are present before that element in the given original array. Given a binary associative operator $\oplus$ and an array of n elements $\big[a_0, a_1, a_2, a_3..., a_n - 1\big]$, the prefix scan returns an array $\big[(a_0), (a_0 \oplus a_1), (a_0 \oplus a_1 \oplus a_2), ..., (a_0 \oplus a_1 \oplus .... \oplus a_n - 1)\big]$.

The algorithm implemented is based on the scan algorithm given by Hillis and Steele [**?**]. The resource configurable implementation of it takes $O(\log n)$ steps to finish. Each step performs the kernel invocation on same amount of resources. The performance of prefix scan is shown in Figure 9.2 (prefix scan on 20 million tuples). The number of threads per thread block is taken as 1024 for this cases.

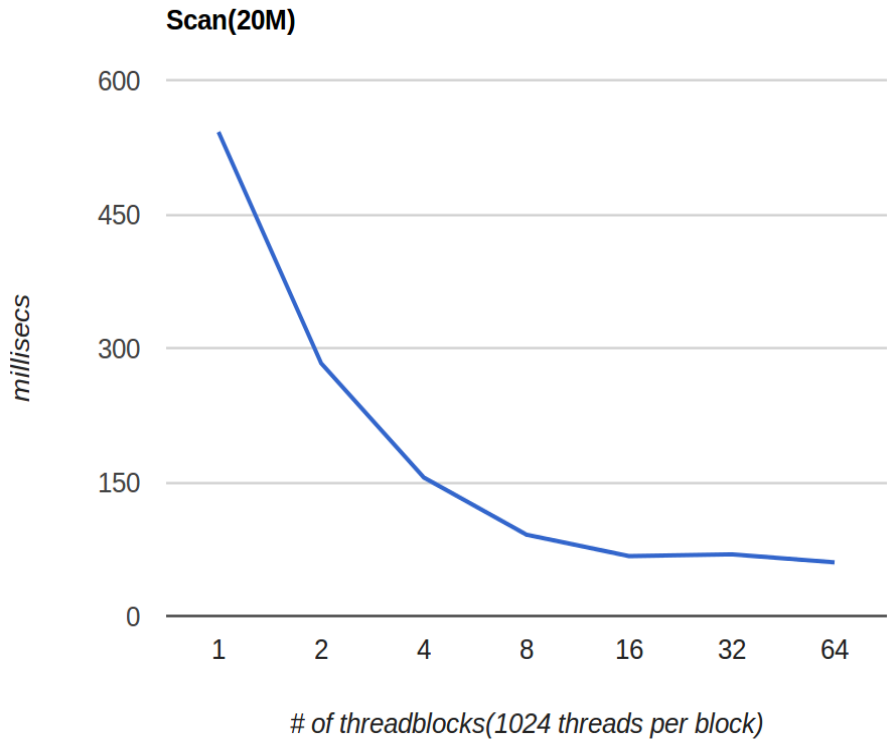

Figure 9.2: Resource Configurable Prefix Scan(20M)

Chapter 10

Performance Evaluation

10.1 Experimental Setup

For evaluating the performance of algorithms, we are using a machine with an Intel Xeon CPU E5-2620 v2 with six cores @2.10GHz, 24 GB main memory, and a NVIDIA Tesla K40m having 10 GB of device memory. Apart from these, on a software side cuda is used for GPU programming and openmp[4] for parallelism in CPU. We are doing the experiments on modified TPCH queries as our system is not capable of dealing with strings and the database size is 10 GB. Before starting the experiments, the database is pre-loaded into the main-memory.

10.2 Experiments

10.2.1 Chain Plan Trees

The dynamic programming algorithm gives us the optimal solution whereas greedy algorithm works for the short term benefits. The assignment of nodes produced by greedy algorithm is very close to the assignment produced by dynamic programming algorithm whereas the time taken to produce these assignment is way more in case of dynamic programming algorithm. In most of the cases, the pure CPU and the pure GPU time is more than the hybrid execution time. The queries QS1,..QS6 are shown in the appendix and the results are shown in Figure 10.1.

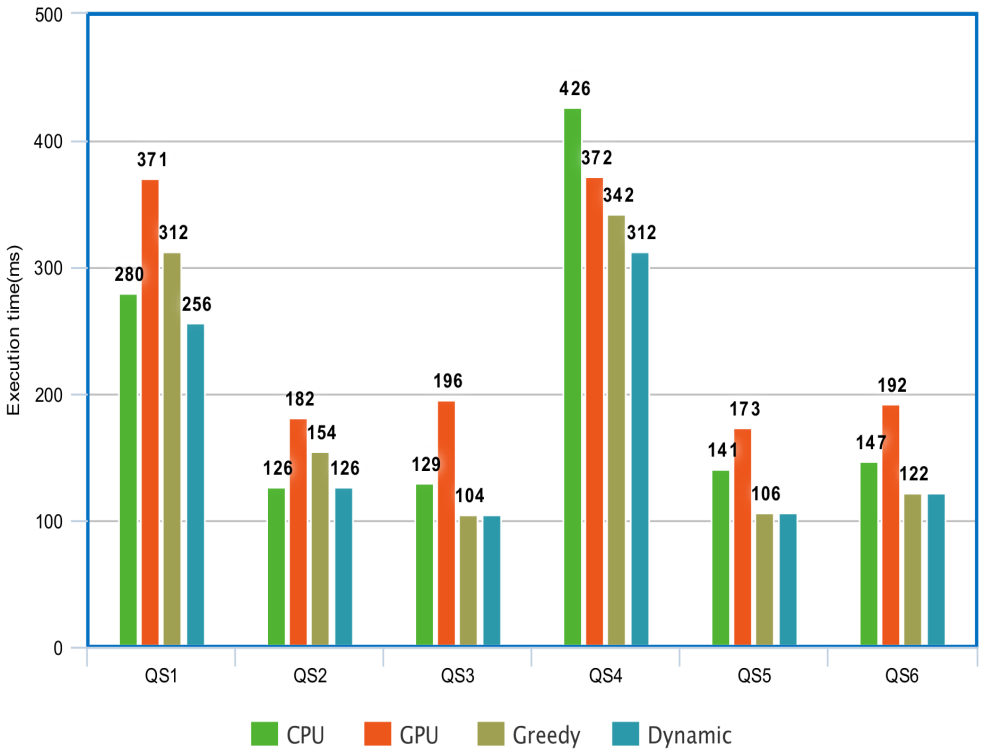

Figure 10.1: Chain Plan Tree Performance

10.2.2 Non-chain Plan Trees

The HEFT approach is applied to various modified TPCH benchmark queries to get the CPU-GPU assignment and scheduling and the performance is compared with the existing database system MonetDB, our CPU and GPU implementations using operator algorithms as given in [10][9]. The performance of CPU-GPU implementation using HEFT is shown in Figure 10.2. The CPU-GPU implementation performs very well as compared to CPU or GPU implementation. Its performance with MonetDB is comparable as in some cases CPU-GPU implementation outperforms MonetDB where as in other cases it is close to MonetDB. The MonetDB performance is much better as compared to our CPU implementation even though both are using multiple cores, it is because we have not used MonetDB operator implementation but the implementation given in [10][9]. By using MonetDB operators implementation the CPU-GPU performance can be increased even more. The execution of these queries is done by using streams to overlap the execution time with the data transfer time, in this way the data transfer time is reduced.

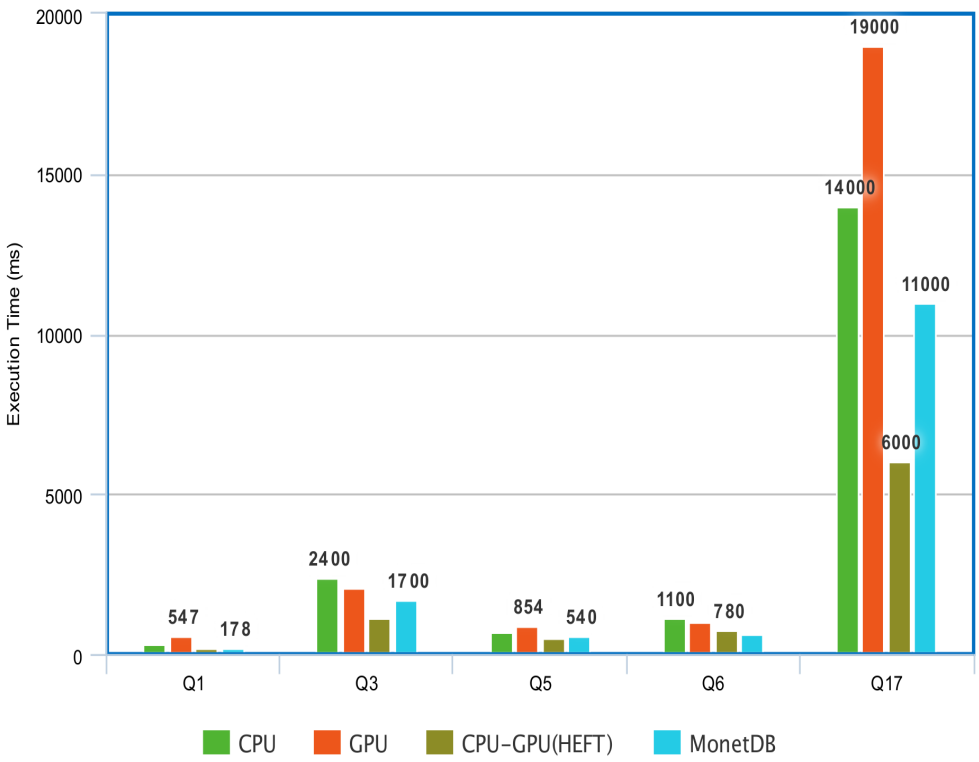

Figure 10.2: HEFT Performance

10.2.3 Intra-Operator Parallelism

The intra-operator parallelism is done for four operators- filter, projection, hash join and group-by. These operators are implemented to use both CPU and GPU to execute a single operation. The amount of data to be processed in divided among CPU and GPU in such a way that the execution time of CPU is equal to the summation of the execution time of GPU and the data transfer time.

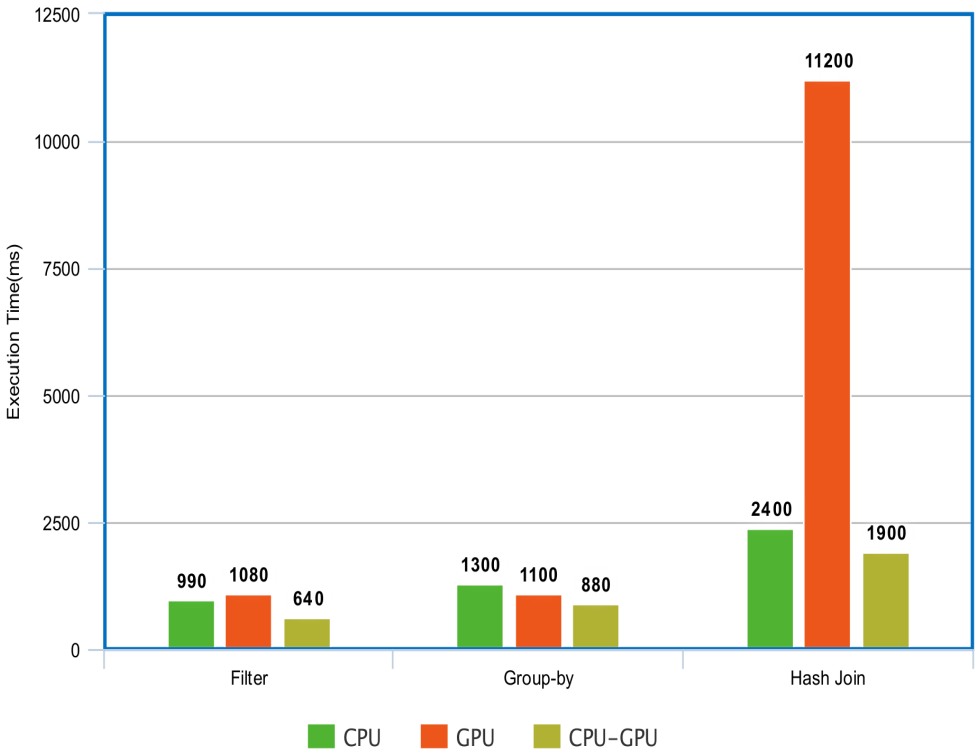

Figure 10.3: Intra-Operator Parallelism

The performance of various operators is shown in Figure 11. The intra-operator parallelism (shown as CPU-GPU in Figure 10.3) outperforms both CPU and GPU implementation as both the devices are used in CPU-GPU implementation.

Chapter 11

Conclusions

A brief description of your chapter.

- We have automated the process of converting the MonetDB plan which is in the form of MAL instruction into the tree structured plan.
- The HEFT algorithm gives performance close to MonetDB and better than standalone CPU or GPU implementations for non-chain structured plan trees.
- For chain plan trees, the dynamic programming approach gives the optimal assignment and for database application the greedy approach performs close to dynamic programming approach.
- The problem of data transfer from CPU host memory to GPU device memory can be solved with the help of streams by overlapping the computations with the communication cost but still the initial transfer could be bottleneck in certain cases.
- The intra-operator parallelism can be used for smaller queries and queries involving single relations to make sure the other device is not sitting idle.

Chapter 12

Future Work

A brief description of your chapter.

- Integrating other algorithms like nested loop join, sort merge join, index nested loop join, indexes on GPU etc. to the system.
- Currently we are using the CPU optimized plan so the future work also includes designing an optimizer which takes into consideration all the devices present in the system.
- Another bottleneck for performance is the operator implementations which are not using new features of cuda so changing the current operator implementations to use newer features of cuda.
- Integrating MonetDB operator implementation to our existing system.

Appendix A

Queries

The queries which we run for chain plan trees are the following-

Listing 1 Query QS1

```
select sum(l_quantity) as sum_qty, l_linenumber
from lineitem where l_suppkey <= 2000
group by l_linenumber
order by sum_qty;
```

Listing 2 Query QS2

```
select sum(c_acctbal) as sum_qty, c_nationkey
from customer
where c_nationkey <= 10 and
c_custkey between 1000 and 500000
group by c_nationkey;
```

Listing 3 Query QS3

```
select count(*)
from ( select o_totalprice
        from orders
        where o_orderkey between 2000 and 20000
        and o_shippriority = 0
        group by o_totalprice) as R;
```

Listing 4 Query QS4

```
select sum(l_extendedprice) as sum,
count(*) as cnt
from lineitem
where l_suppkey <= 50000 and l_partkey <= 1000
group by l_linenumber
order by sum;
```

Listing 5 Query QS5

```
select avg(p_retailprice) as avg,
sum(p_retailprice) as sum,
count(*) as cnt from part
where p_size >= 1 and p_partkey
between 5000 and 50000
group by p_size;
```

Listing 6 Query QS6

```
select max(ps_availqty) from partsupp
where ps_suppkey between 200 and 20000 and
ps_partkey >= 10000 and
ps_supplycost > 999
group by ps_supplycost;
```

Bibliography

[1] https://www.monetdb.org/. MonetDB. ii, 1, 4, 12

[2] https://developer.nvidia.com/cuda-toolkit/. 1, 6, 9

[3] https://www.khronos.org/opencl/, . 6

[4] http://www.openmp.org/, . 9, 28

[5] https://link.springer.com/referenceworkentry/10.\10072F978-0-387-09766-4_42. 19

[6] Peter Boncz, Stefan Manegold, and Martin Kersten. The Yin and Yang of Processing Data Warehousing Queries on GPU Devices. In *Proceedings of Very Large Data Bases (VLDB) Endowment*, 2013. 8

[7] M.R. Gary and D.S. Johnson. In *Computers and Intractability: A Guide to the Theory of NP-Completeness*, 1979. 19

[8] Rahul Hasija. Executing Database Query on Modern CPU-GPU Heterogeneous Architecture. In *M.E. Thesi*, 2016. 6

[9] Bingsheng He, Mian Lu, Ke Yang, Rui Fang, Naga K.Govindaraju, Qiong Luo, and Pedro V. Sander. Relational Joins on Graphics Processors. In *ACM SIGMOD International Conference on Management of Data*, 2008. 8, 9, 10, 11, 24, 26, 29

[10] Bingsheng He, Mian Lu, Ke Yang, Rui Fang, Naga K.Govindaraju, Qiong Luo, and Pedro V. Sander. Relational Query Co-processing on Graphics Processors. In *ACM Transactions on Database Systems (TODS)*, 2009. 8, 9, 29

[11] M. Heimel, M. Saecker, H. Pirk, S. Manegold, and V Markl. Hardware-Oblivious Parallelism for In-Memory Column-Stores. In *Proceedings of the Very Large Data Bases (VLDB)*, 2013. 8

[12] Stratos Idreos, Fabian Groffen, Niels Nes, Stefan Manegold, Sjoerd Mullender, and Martin Kersten. MonetDB: Two Decades of Research in Column-oriented Database Architecture. In *IEEE Computer Society Technical Committee on Data Engineering*, 2012. 12

[13] Johns Paul, Jiong He, and Bingsheng He. Gpl: A GPU-based Pipelined Query Processing Engine. In *ACM SIGMOD International Conference on Management of Data*, 2016. 8

[14] Haluk Topcuoglu, Salim Hariri, and Min-You Wu. Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing. In *IEEE Trans. Parallel and Distributed Systems*, volume 13, 2002. ii, iii, 3, 20

[15] J.D. Ullman. NP-Complete Scheduling Problems. In *Computer and Systems Sciences*, volume 12, 1975. 19

[16] Y. Yuan, R. Lee, and X. Zhang. Database Architecture Optimized for the New Bottleneck: Memory Access. In *Proceedings of the 25th International Conference on Very Large Data Bases (VLDB)*, 1999. 10