Executing Database Query on Modern CPU-GPU Heterogeneous Architecture

A PROJECT REPORT
SUBMITTED IN PARTIAL FULFIMENT OF THE
REQUIREMENTS FOR THE DEGREE OF
Master of Engineering
IN
Faculty of Engineering

BY

Rahul Hasija

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

June, 2016

Declaration of Originality

I, **Rahul Hasija**, with SR No. **04-04-00-10-41-14-1-11161** hereby declare that the material presented in the thesis titled

Executing Database Query on Modern CPU-GPU 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 **2014-2016**.
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: 30 June 16 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: Prof. Jayant R. Haritsa Advisor Signature

DEDICATED TO

My Family and Friends
for their love and support.

Acknowledgements

I am deeply grateful to Prof. Jayant R. Haritsa for his unmatched guidance, enthusiasm and supervision. His great personality and great thoughts have always been a source of inspiration to me. I have been extremely fortunate to work with him.
I am deeply grateful to Prof. R. Govindarajan for his technical advice in my project and imparting knowledge of computer architecture in me.

I am thankful to Anshuman Dutt, Srinivas Karthik, Jayvant Anantpur, Adarsh Patil for their assistance and guidance. It had been a great experience to work with them. My sincere thanks goes to my fellow lab mates as well for all the help and suggestions. Also I thank my friends Pradeep Pandey, Vipin Nagar and other CSA friends who made my stay at IISc pleasant, and for all the fun we had together.
Finally, I am indebted with gratitude to my parents, brother and sister in law for their love and inspiration that no amount of thanks can suffice. This project would not have been possible without their constant support and motivation.

Abstract

Graphics processor (GPU) have emerged as a powerful co-processor for general-purpose computation. Compared with commodity CPUs, GPUs have an order of magnitude higher computation power as well as memory bandwidth. The execution time of database query can be reduced by using GPU as a coprocessor to CPU. This can be done by dividing the computation task between both the processors optimally. Depending upon the data size, algorithm used for operator and input data distribution, either of the CPU or the GPU could perform better than the other respectively. We try to come up with the solution to partition the operators of query plan tree on the CPU and GPU, so as to execute the query faster than executing query on contemporary database engine. We have initial promising result in which our CPU-GPU based query plan implementation is performing better than the contemporary database engines like MonetDB [5].

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Since the invention of the relational database management system (**RDBMS**), performance demands of the applications have been increasing. To achieve high performance in RDBMS, queries processed in database engine should be executed as fast as possible. A database execution plan tree which is usually a binary tree, consist of set of nodes. Each node of the plan tree represents a operator. To execute a operator (e.g. : Join) an algorithm (e.g.: Hash Join, Sort Merge Join etc.) is executed. A query plan tree is shown in Figure 1.1.
In [1] a query plan tree was executed on discrete heterogeneous architecture having GPU of compute capability 1.x, which does not support multiple stream execution [3]. They reported no speed up of their GPU implementation or CPU-GPU implementation of plan tree compared to the CPU implementation of their plan tree. In [2], researchers used coupled architecture for processing the query. After advent of modern GPUs like Kepler 3.x [7] and Maxwell 5.x architecture the power of multiple streams can be exploited to reduce the execution time of database query.

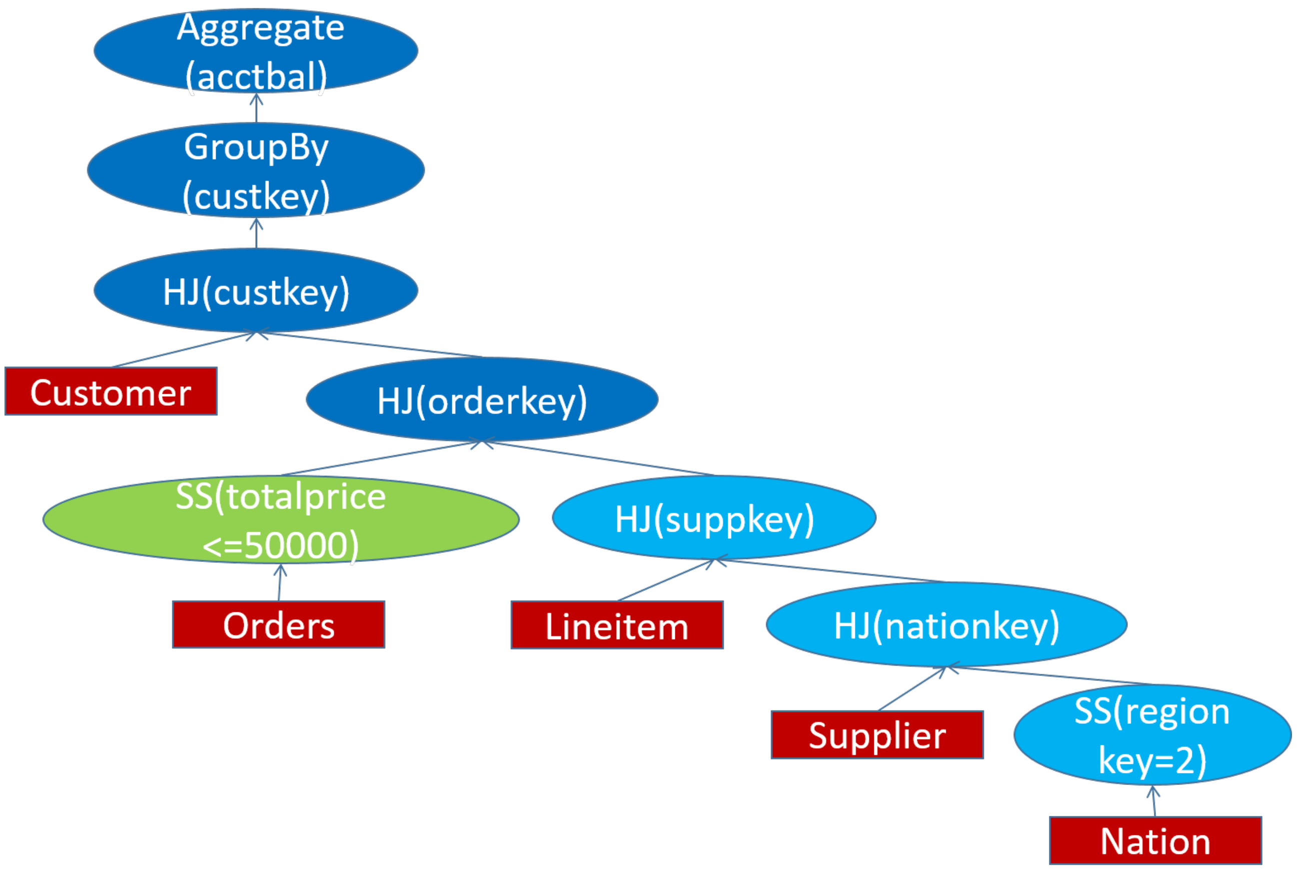

Figure 1.1: Execution plan tree

Chapter 2

Background on GPU

There are two types of heterogeneous architecture

1. Discrete Heterogeneous Architecture

2. Coupled Heterogeneous Architecture

In discrete heterogeneous architecture, CPU and GPU are connected through PCIe bus as shown in Figure 2.1. Both CPU and GPU have their own memory designed specifically to their requirements. In coupled architecture, CPU and GPU both are integrated on single chip. The coupled architecture is less powerful than the discrete architecture because of the following reasons:

- The number of cores dedicated to the GPU are lesser than the GPU in discrete heterogeneous architecture. This is because CPU and GPU have to share space on single chip.

- GPU has the requirement of high bandwidth memory, as CPU and GPU are integrated on single chip, GPU have to use memory designed specifically for the CPU which is low latency and low bandwidth.

From now onwards we will regard CPU as the *host* and GPU as the *device*.

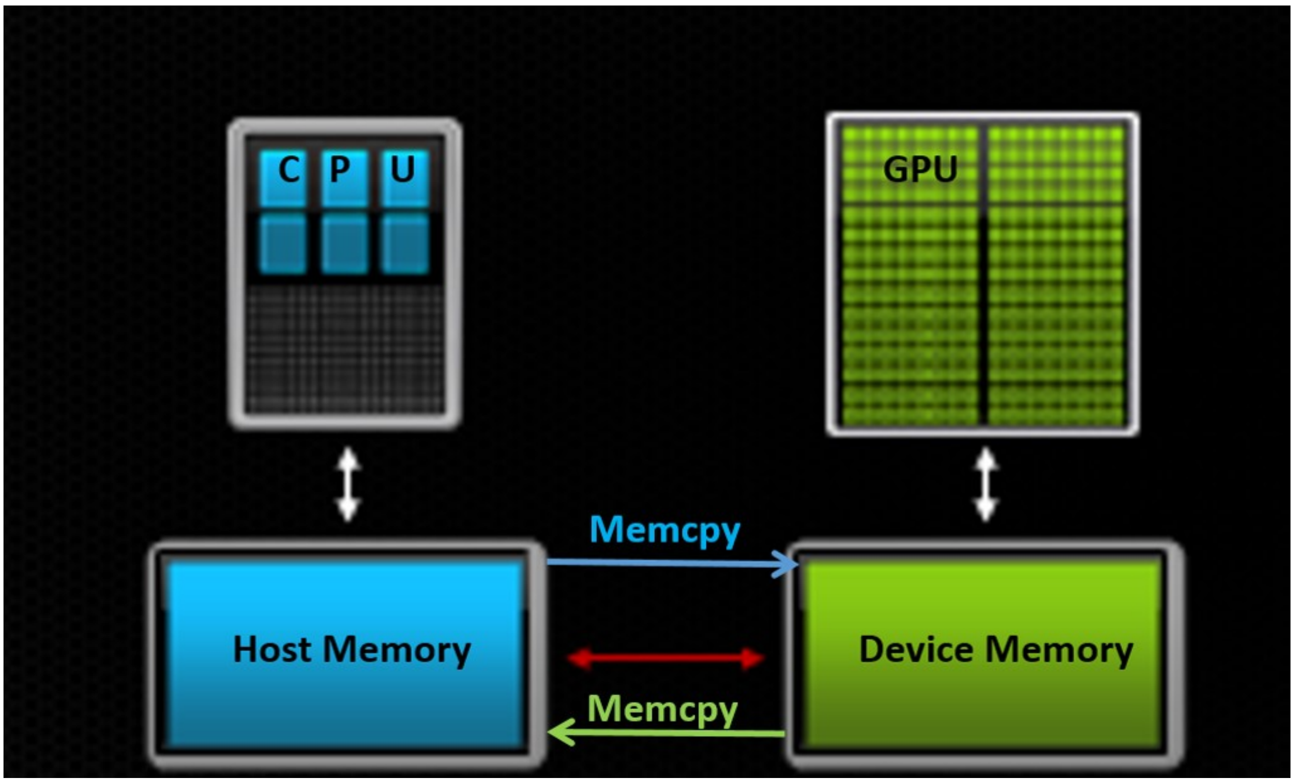

Figure 2.1: Discrete Architecture

Kernel is a basic function which executes on GPU. It can be written in CUDA or OpenGL. Whenever the host thread *launches (calls)* the kernel, kernel is scheduled on GPU and then kernel is executed on the GPU along with the 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.
Thread block is a collection of threads. Grid is a collection of thread blocks. When kernel is launched on a GPU, the programmer have to specify the dimension of the grid i.e. number of thread blocks, dimension of the thread blocks i.e. number of threads in the thread block. Each thread runs the instructions in the kernel independent of each other, therefore each thread requires its 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. From the above we conclude that the thread block is collection of threads and thread

block requires registers, shared memory to execute. From now onward we will call threads in the thread block, registers and shared memory required by the thread block as the *resources* required by the thread block to execute.

GPU contains few streaming multiprocessor ***(SMX)***. Each SMX has fixed

1. number of registers.
2. amount of shared memory (software controlled cache in contrast to CPU which has hardware control cache).

Each SMX can have at most certain number of

1. threads to execute.
2. thread blocks to execute.

Table 2.1 shows the configuration of Tesla k40m GPU.

Resources	**Quantity**
# of SMX	15
# of registers on SMX	64K of 4 Bytes each
Shared Memory Per SMX	48KB
Max # of threads on SMX	2048
Max # of thread blocks on SMX	16

Table 2.1: Configuration of Tesla k40m

We will define some terms which we will use further in the report:

1. A **thread block is scheduled** when it gets resources to execute on any SMX of the GPU.
2. A **thread block is executed** when all of its threads have executed the instructions in the kernel and released their resources back to their SMX.
3. A **kernel is scheduled** when atleast one thread block of kernel has been scheduled.
4. A **kernel is executed** when all of its thread blocks have been executed and released their resources back to their SMX respectively.

5. A **kernel is not resource configurable** when we cannot control the number of thread blocks launched for that kernel. The number of thread blocks launched depends upon the size of the input data to the kernel and number of threads launched in the thread block.

Two kernels in GPU cannot communicate with each other therefore no pipelining is possible between two kernels in GPU, hence every kernel is blocking node in tree. A thread block of a kernel is a schedulable unit on SMX i.e. either all the threads of a thread block are scheduled on SMX or non of them is scheduled on SMX. Once the thread block is scheduled to SMX it will not be preempted from that SMX i.e. until all of its threads had executed all of the instructions in its kernel, it will not be preempted from that SMX.

2.1 Maximum resident thread block of kernel

Each thread block of the kernel requires the resources in the GPU, therefore only limited number of thread blocks could be launched on the GPU. The maximum number of thread blocks of the kernel that can be resident on the GPU simultaneously is called *max residency of kernel*. The maximum resident thread block of any kernel can be computed by Algorithm 1, given the resources available with GPU and the resources required by a thread block.

```
int MaxResidentTBOfKernel( $ResourcesGPU$, $ResourcesTB$) {

$registerTB \leftarrow \frac{GPU.SMX.registers}{TB.registers}$ {$GPU.SMX.registers = 64K$ for k40m}

$threadsTB \leftarrow \frac{GPU.SMX.threads}{TB.threads}$

$sharedMemoryTB \leftarrow \frac{GPU.SMX.sharedMemory}{TB.sharedMemory}$

return min($registerTB, threadsTB, sharedMemoryTB$)
}
```

Algorithm 1: Max resident thread block of kernel.

From now onwards, we will specify kernel K by
K(number of registers used per thread, shared memory used per thread block, number of threads in a thread block, number of thread blocks).

2.2 Streams

Stream is a sequence of operations that execute in issue-order on the GPU i.e. the operation in a stream start execution only when earlier operations in that stream have completed their execution. Operations in different streams may run concurrently or may be interleaved i.e.

different streams may execute their operation concurrently or out of order with respect to each other. Operation could be memory transfer operation from host to device memory or from device to host memory or it can be a kernel launch. In Figure 2.2 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.
From now onwards we will say

1. A **stream is executed** when all its kernels are executed on the GPU.
2. A **stream is activated** when it is ready to schedule its front kernel on the GPU.
3. A **stream is deactivated** when it is not ready to schedule its front kernel on the GPU.

For example in Figure 2.2, $stream_1$ is ready to schedule kernel K_1 on the GPU. Once the GPU scheduler has schedule 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.

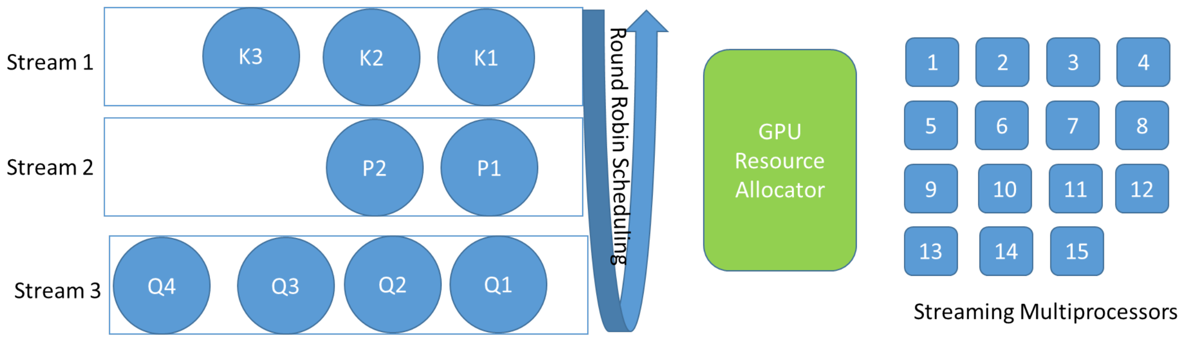

Figure 2.2: GPU Scheduler

2.2.1 How streams can be created

Streams can be created in two ways:

1. By calling CUDA function *cudaStreamCreate* explicitly and launching the kernel in the respective streams.
2. By creating POSIX pthread (only available after CUDA 7.0), each pthread will have its respective default stream. Kernels launched by the pthread will be executed in its own pthread's default stream. Therefore kernels launched by different pthreads can run concurrently and may be interleaved.

Chapter 3

Our Contribution

3.1 Assigning the stream Id to nodes in plan tree

In a database query plan tree, many nodes in a tree could be executed concurrently, for example all the leaf nodes of the tree could be executed concurrently. With the help of multiple streams, multiple kernels could be executed concurrently by spawning each of them into distinct streams. Assigning different streams to nodes in the plan tree, which could run concurrently, helps us in following ways:

1. To have more possible ways to divide the task between CPU and GPU.
2. If one stream of GPU is blocked, then other stream could be processed by GPU scheduler, hence efficient utilization of resources of the GPU.

Database plan tree executes in following manner. If a node has child nodes then it should start executing only after its child nodes had completed their execution. Therefore we will require synchronization between the execution of parent node with child node. We assign distinct streams to the nodes of the binary plan tree by the Algorithm 2. The Algorithm 2 assigns distinct streams to nodes which could run concurrently and synchronize the streams which requires synchronization to execute the plan tree correctly.

```
void AssignStreamId ($root$, $streamId$) {
if $root$=$NULL$ then
    return;
end
root.streamId=$streamId$;
streamStack[$streamId$].push($root$);
if root has two child then
    $leftStreamId$=getNewStreamId();
    $rightStreamId$=getNewStreamId();
    AssignStreamId (root.left, $leftStreamId$);
    AssignStreamId (root.right, $rightStreamId$);
    $streamId$ has to synchronize with the execution of the $leftStreamId$ and
     $rightStreamId$
else
    AssignStreamId (root.left, $streamId$);
    AssignStreamId (root.right, $streamId$);
end
}
```

Algorithm 2: Converting binary tree into set of streams

Figure 3.1 gives running example of our Algorithm2. The binary tree shown in Figure 3.1 is cut down into five distinct stream each with different color. Stream yellow have to synchronize with stream blue and stream green and stream red have to synchronize with the execution of stream yellow and stream purple.

By the Algorithm 2, for a parent node having one child node, both the nodes will be put up in a single stream. Implicit synchronization (i.e. nodes in stream will be executed in order) will maintain the execution order $child \rightarrow parent$. For nodes having two child nodes, all three nodes will be put in three distinct streams therefore we have to use explicit synchronization i.e. making wait the stream in which parent node belongs for the streams in which child nodes belongs, to maintain the execution order. $child_1 \rightarrow parent$, $child_2 \rightarrow parent$.

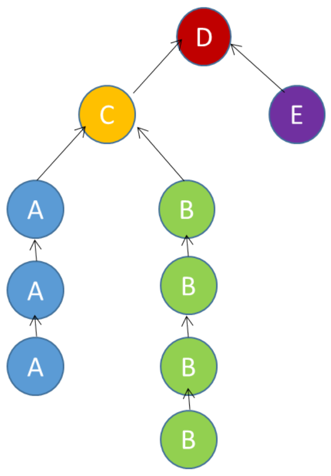

Figure 3.1: Assigning stream to nodes of binary tree

3.2 Round robin scheduler of Nvidia

Nvidia does not provide any official documentation of how the scheduling of kernels from multiple streams takes place, therefore we conducted some experiments to know the algorithm. There are other internal information about the GPU which are not revealed by Nvidia because of the business strategy or rapid change in the architecture of GPU. It might happen that some features are upgraded in such a way that program written for previous generation are not portable to next generation of GPU. Therefore Nvidia doesn't reveal some features which might affect program portability.

We launched two kernels $K_1(20, 48KB, 128, 16)$, $K_2(18, 0KB, 64, 1)$ in two different streams S_1 and S_2. The maximum thread block residency of K_1 is fifteen on k40m GPU from Algorithm 1 and Table 2.1. We launched sixteen thread blocks which is one greater than maximum thread block residency of K_1. We observe that

- When two streams were activated in order $S_1 \rightarrow S_2$, then the 16^{th} thread block was scheduled only after any one thread block of K_1 was executed and release its *shared memory resources* to SMX. Kernel K_2 was not scheduled until the 16^{th} thread block of kernel K_1 had been scheduled, although resources were available to execute the single thread block of kernel K_2 with the first fifteen executing thread blocks of kernel K_1.

- When we altered the launching order of streams i.e $S_2 \rightarrow S_1$ then both the kernels K_1 and K_2 start simultaneously, hence may reduce the total execution time of two kernels.

Similar experiment was conducted by launching two kernels $K_1(28, 0KB, 992, 31), K_2(28, 0KB, 32, 1)$. The max residency of K_1 is 30 from Algorithm 1 and Table 2.1. We observed that

- $S_1 \rightarrow S_2$: The 31^{st} thread block was scheduled only after any one thread block of K_1 was executed and release its *thread resources* to SMX. The kernel K_2 was not scheduled until 31^{st} thread block of kernel K_1 is scheduled.
- $S_1 \rightarrow S_2$: Both the kernels K_1 and K_2 start simultaneously.

From the above observation we conclude that

1. The stream scheduler will schedule all the thread blocks of kernel in the stream it processes currently. If it is not able to schedule all the thread blocks, it will wait for resources to be freed and will not process any other stream, even though kernel in different streams can be launched concurrently.
2. After scheduling all the thread blocks of kernel in current stream, it will start processing another stream in round robin order.

The round robin algorithm is illustrated in Algorithm 3.

```
Result: Schedule all the streams launched on GPU.
void RoundRobin($streamStack$, $numberOfStreams$) {
for $i \leftarrow 0, numberOfStreams$ do
    if $streamStack[i].empty()$ then
        $numberOfStreams \leftarrow numberOfStreams - 1$;
        remove the $streamStack[i]$;
    else if $streamStack[i]$ is active then
        $kernel \leftarrow streamStack[i].top()$;
        $streamStack[i].pop()$;
        for all $threadBlock$ in $kernel$ do
            ScheduleThreadBlockSMX($threadBlock$);
        end for
        deactivate the $streamStack[i]$ until all the thread block of $kernel$ are executed;
    end if
end for
}
```

Algorithm 3: Round Robin Scheduler of k40m

```
Result: Schedule the given $threadBlock$ on any one SMX.
void ScheduleThreadBlockSMX($threadBlock$) {
while True do
    for $i \leftarrow 0, numberOfSMX$ do
        if CompareResources($threadBlock.resources$, $SMX[i].resources$) then
            AllocateResources($threadBlock.resources$,
            $SMX[i].resources$);
            return;
        end if
    end for
end while
}
```

```
Result: Returns true when resources are available to execute the given $TB$ on given
        $SMX$.
bool CompareResources($TB, SMX$) {
$b0 \leftarrow TB.registers \leq SMX.registers$;
$b1 \leftarrow TB.sharedMemory \leq SMX.sharedMemory$;
$b2 \leftarrow TB.threads \leq SMX.threads$;
$b3 \leftarrow SMX.threadBlock \geq 1$;
return$b0\&b1\&b2\&b3$;
}
```

```
Result: Allocate resources for the thread block $TB$ from given $SMX$
    AllocateResources($TB, SMX$) {
    $SMX.registers \leftarrow SMX.registers - TB.registers$
    $SMX.sharedMemory \leftarrow SMX.sharedMemory - TB.sharedMemory$
    $SMX.threads \leftarrow SMX.threads - TB.threads$
    $SMX.threadBlock \leftarrow SMX.threadBlock - 1$
    }
```

As the scheduling of streams should be efficient and fast, it should be less complex and have less storage requirement. Therefore Nvidia might have choose algorithm 3 for scheduling the streams.

3.3 Executing the plan tree

Assume that the plan tree has directed edge from child node to parent node, the topological ordering of tree gives the valid execution sequence of kernel nodes because in the topological ordering of tree all the child nodes will be executed before the parent node which is our requirement to execute the database plan tree. To execute the given plan tree, we will cut down the tree into set of streams by Algorithm 2. Then the streams having leaf nodes will be activated initially. For the stream which have to synchronize with the execution of two other streams, will be activated as soon as two others streams had been executed. By this way, whole plan tree is executed by the GPU.
We observed that the *order in which streams are initially activated can give different execution time of the tree*. For example: In Figure 3.2, we have tree with six kernel nodes. Node $K_{SM}(20, 48KB, 64, 15)$ signifies that each of its thread blocks will acquire all the shared memory on each SMX of GPU k40m. While node $K_{NSM}(28, 0KB, 64, 4)$ signifies that each of its thread blocks will not acquire any shared memory on SMX. We cut down this tree into five streams by the Algorithm 2. $Stream_y, Stream_b, Stream_r$ will be activated initially as they contain the leaf node while $Stream_p$ has to wait for execution of $Stream_y$, $Stream_b$ and $Stream_n$ has to wait for execution of $Stream_p$, $Stream_r$.

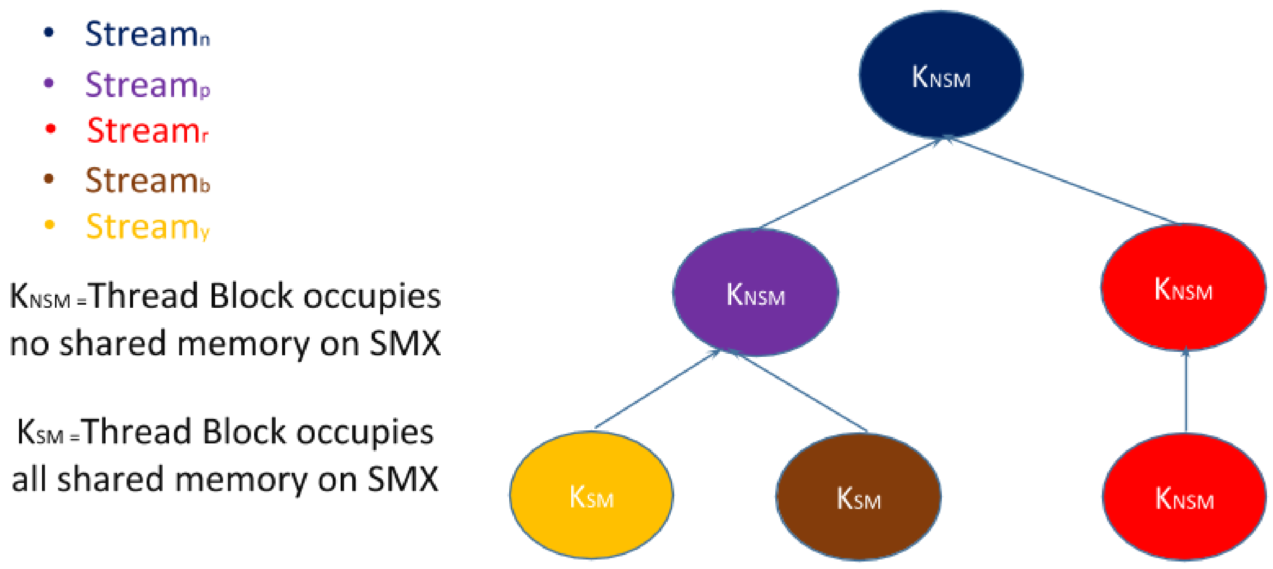
Stream$_n$
Stream$_p$
Stream$_r$
Stream$_b$
Stream$_y$
K$_{NSM}$ =Thread Block occupies no shared memory on SMX
K$_{SM}$ =Thread Block occupies all shared memory on SMX
K$_{NSM}$
K$_{NSM}$
K$_{NSM}$
K$_{SM}$
K$_{SM}$
K$_{NSM}$

Figure 3.2: Tree with six kernel nodes and five streams.

We executed the plan tree on GPU and observed the following

- If streams are activated in the order $Stream_y \rightarrow Stream_b \rightarrow Stream_r$ then kernel in $Stream_b$ has to wait for completion of a thread block of kernel in $Stream_y$ because of the lack of shared memory resource. This will in turn make wait kernels in $Stream_r$ because round robin scheduler will not process $Stream_r$ unless it schedules kernel in $Stream_b$.
- If they are activated in the order $Stream_y \rightarrow Stream_r \rightarrow Stream_b$ then the kernel in $Stream_r$ will be executed concurrently with kernel in $Stream_y$.

The time line diagram for both the cases generated by nvcc pro-filer is shown in Figure 3.3 and Figure 3.4. The overlapping sequence has less execution time than non overlapping sequence. For some cases it might happen that two kernels which are both memory intensive might degrade the performance of each other and get more execution time if they are overlapped rather than if they were not overlapped. From above we conclude that the order in which streams are initially activated can give different execution time of the tree.

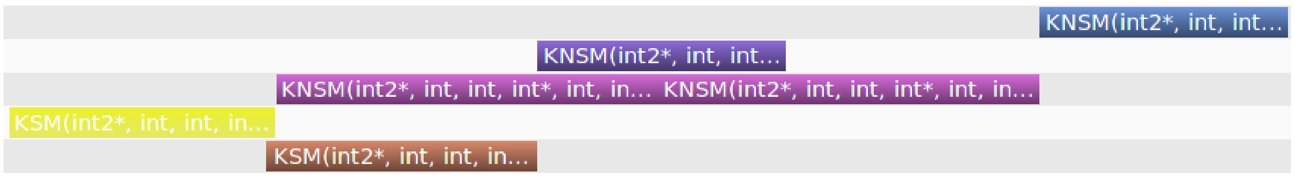

Figure 3.3: Non overlapping $Stream_r$ and $Stream_y$

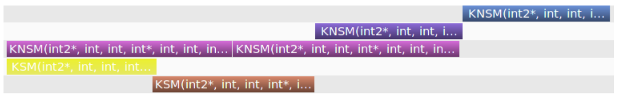

Figure 3.4: Overlapping $Stream_r$ and $Stream_y$

Chapter 4

Experiments

4.1 Machine configuration

The architecture of machine we used for our experiments is discrete heterogeneous architecture. Table 4.1 and Table 4.2 shows the configuration of the Xeon CPU and Tesla k40m GPU [7] used, respectively.

Number of cores in CPU	6
L3 cache size	15MB
Host memory	24GB

Table 4.1: CPU: Intel(R) Xeon(R) CPU E5-2620

Number of SMX	15
Number of cores per SMX	192
Number of registers per SMX	64K
Size of on chip shared memory per SMX	48KB
Size of each register	4B
Device memory	12GB

Table 4.2: GPU: Tesla k40 GPU

4.2 Available algorithm for database operators

We have an implementation of most of the logical database operators algorithm for GPU in CUDA, for CPU in OpenMP from [6] shown in Table 4.3.

Operator	**Algorithm on CPU**	**Algorithm on GPU**
Sort	Quick sort	Bitonic sort, Radix sort
Aggregate	Parallel aggregate	Parallel reduce
Group By	Sort group by	Sort group by
Filter	Sequential scan	Sequential scan
Join	HJ, SMJ, INLJ, NLJ	Hash join

Table 4.3: Algorithms for database operators on CPU and GPU

Currently algorithms are implemented in OpenMP for CPU and in CUDA for GPU [6]. Algorithms are not able to handle:

- Multiple attribute group by operator.
- Multiple attribute sort operator.
- Non equi joins.
- Date, string and float data type.
- Like and EXISTS operator.
- Nested queries.

Because of the above limitations, we cannot execute any TPC-H query [8] without modification. Therefore we selected three TPC-H queries from twenty two TPC-H queries and modified them by replacing

- Multi attribute group by by single attribute group by.
- Multi attribute sort by single attribute sort.
- Non equi join with equi join.
- Date or string data type attribute of a relation with similar selectivity integer or float data type attribute of respective relation.
- Rounded off float values of attribute to greatest integer values.
- Unnested the nested TPC-H Query 17.

The algorithms were using multiple global variable which lead to inconsistent value in multithreaded environment. We modified the algorithm as per the requirement of multithreaded environment.

4.3 TPC-H Database

TPC-H Database [8] with scaling factor of 10 was generated i.e. size of whole database was 10GB. While executing query on GPU, we didn't want to transfer some intermediate output to host memory because of insufficient space on GPU device memory therefore we scaled our database to 10GB only which is less than 12GB size of device memory. A higher scaling factor may be tried out as current scaling factor didn't cause insufficient device memory for queries which we used for our experiments. The generated database contains eight relations. Relations and their respective cardinalities are shown in Table 4.4.

Relation	**Cardinality**
Customer	1.5m
Lineitem	59m
Part	2m
Partsupp	8m
Region	5
Nation	25
Supplier	0.1m
Order	15m

Table 4.4: TPC-H Database of size 10GB

4.4 TPC-H Queries

Modified TPC-H queries used for our experiments are following:

4.5 TPC-H Query 5

- **Select** c_custkey, sum(c_acctbal)
  From orders, lineitem, supplier, nation, customer, region
  Where c_custkey = o_custkey
  and l_orderkey = o_orderkey
  and l_suppkey = s_suppkey
  and s_nationkey = n_nationkey
  and n_regionkey = r_regionkey
  and r_regionkey = 3
  and o_totalprice $\leq$ 50000

and c_custkey
and c_custkey
Limit 20

4.6 TPC-H Query 10

- **Select** c_custkey, sum(l_extendedprice) as revenue
  From customer, orders, lineitem
  Where c_custkey = o_custkey
  and l_orderkey = o_orderkey
  and o_totalprice $\leq$ 20000
  and l_quantity $\leq$ 12
  Group by c_custkey
  Order by c_custkey
  Limit 20

4.7 TPC-H Query 17

- **Select** sum(R.l_extendedprice)
  From
  (
  select *
  from lineitem, part
  where p_partkey = l_partkey
  and p_size=1
  and p_retailprice $\leq$ 1040.99
) as R,
  (
  select
  p_partkey, floor(avg(l_quantity)/33) as avg_quantity
  from lineitem, part
  where l_partkey = p_partkey
  group by p_partkey
  having floor(avg(l_quantity)/33) $\geq$ 1
) as S
  Where R.l_quantity=S.avg_quantity

4.8 Experimental setup

- **Columnar Storage**: We used columnar storage model for storing our relations on the file-system as well as in the memory. Each attribute of relation was represented by an array of structures. Each structure element is represented by a pair of integers $< row\_id, value >$. Each integer was of 4B.

- **In Memory Database**: To be fair with execution of plan tree on CPU and GPU, we assume that both of the processing unit have required input data for processing, in their respective memory i.e. we excluded initial disk transfer cost and host to device memory transfer cost in the reported execution time.

- **MultiCore OpenMP**: We had exported *GOMP_CPU_AFFINITY* environment variable of operating system to "0 1 2 3 4 5 0 1 2 3 4 5" which signals the operating system to schedule the OpenMP threads consecutively to the cores i.e. i^{th} thread of OpenMP will be scheduled to $(i\%6)^{th}$ core of the Xeon six core CPU.

4.9 Performance

We analysed the performance of each available algorithm and modified TPC-H queries. We validated the output of algorithms by running queries shown in Table 4.5 on MonetDB database engine [5] so as to be sure that the algorithms are working correctly. We similarly validated the output of modified TPC-H queries with MonetDB database engine. The MonetDB database engine is multicore, in memory, columnar storage database engine. All the performance reading of MonetDB are given by running the query iteratively until the execution time is same for three consecutive reading. This is done so that required data is in the host memory. Comparing our performance with MonetDB is unfair for our OpenMP, CUDA implementation because of the following reasons:

1. The database on MonetDB has indexes built already on some of the columns in relation according to TPC-H benchmark index queries. While currently we don't have provision for building the indexes on the columns in OpenMP or CUDA.

2. The MonetDB also stores some intermediate results and creates indexes on the fly for the columns, so that it could improve the performance of the query.

3. MonetDB has provision for pipelining of data between operators. In pipelining between operator nodes, parent node operator can start processing data from child node operator

even though child operator has not completely processed its input data. Pipelining helps to improve the performance of plan execution.

4. The plan chosen by MonetDB for particular query is generated by estimating cardinality of the nodes, computing cost of each node in tree etc. while plan chosen for our discrete heterogeneous architecture is random.

Therefore MonetDB is expected to perform much faster than our OpenMP, CUDA based implementation which does not have any indexes built, storing of intermediate result and pipelining between operators. In favor of MonetDB, MonetDB has little additional overhead of parsing the query, finding the optimal execution plan tree to execute the query than our implementation of OpenMP and CUDA.

Operator	**Query**
Filter	Select count(*) from lineitem where l_quantity $\geq$ 1 and l_quantity $\leq$ 5
Aggregate	Select max(l_quantity) from lineitem
Sort	Select l_quantity from lineitem order by l_quantity limit 10
Group By	Select sum(l_linenumber) from lineitem group by l_quantity

Table 4.5: Operator and its corresponding query

4.10 Performance of database algorithm

Performance of each database operator on CPU, GPU and MonetDB is shown in Table 4.6.

Operator	**CPU(ms)**	**GPU(ms)**
Filter	127	127
Aggregate	47	41
Sort	3200	1000(radix sort)
Group By	3300	1017

Table 4.6: Performance of operators on CPU(6 threads), GPU(1024 threads, 128 thread blocks) and MonetDB

- **Performance of algorithms on CPU:** As Xeon CPU has six cores, we launched six threads in OpenMP. The increase in number of OMP threads didn't increase the performance of filter and aggregate operator as the algorithm used, have sequential memory access pattern which exploit the spatial locality property of the cache. In other words the algorithms are cache friendly which lead to very high CPU utilization (close to 100%). Sorting of the array in OpenMP is done by splitting the array into n number of partition and sorting each partition individually by quick sort. In six thread configuration, 16 independent partitions and for ten threads, 32 independent partitions were created for same data-set. The Table 4.7 shows thread assignment per core **TAPC** and partition assignment per core **PAPC**. In 10 thread configuration, only eight thread will be active for sorting the partitions, because OpenMP starts giving four partitions (ceil(32/10)) to each thread consecutively and end up giving no partition to 9^{th} and 10^{th} thread. The 10 thread configuration have execution time 600ms less than six thread configuration as shown in Table 4.7 although split time for the relation was same for both of the configuration.

# of OpenMP threads	**TAPC**	**PAPC**	Time(ms)
6 threads	(1,1,1,1,1,1)	(3,3,3,3,3,1)	3200
10 threads	(2,2,1,1,1,1)	(8,8,4,4,4,4)	2600

Table 4.7: Performance of sort on CPU(6threads) and CPU(10 threads)

Group by requires array to be sorted first and then aggregate is performed on each of the group. Sort has more complexity than aggregate, therefore group by performance is much similar to sort. We didn't compare our performance with MonetDB because of the reasons mentioned in Section 4.6.

- **Performance of algorithms on GPU:** The performance of the filter operator is shown in Figure 4.1 with varying the number of threads in thread block and thread blocks (Blue line represents operator with one thread block).Filter operator has following five kernels: memset, map, prefix scan, uniform add and scatter. Filter operator had non resource configurable prefix scan kernel. Rest of the kernels in the filter are high throughput kernel. Therefore execution time of configuration (*,*,1024,1) i.e. 351ms is just three times lower than (*,*,1024,128) although number of thread block given to execute in later 128 times higher.
  The Performance of aggregate operator is shown in Figure 4.2. Aggregate operator uses single kernel K(32,528,*,*). It has linear speed up from 1 thread block to 30 thread block (With (32,528,1024,30) its execution time is 50ms). This is because max residency for K(32,528,1024,*) is 30 thread block.
  We use radix sort algorithm imported from thrust library [9] for the sort operator in our

experiments as it is performing three times faster than bitonic sort from [6].
Group by operators are implemented by performing sorting followed by aggregate on each group. Bitonic sort group by execution time is much lower than radix sort group by as can be seen from the Figure 4.3 and 4.4.

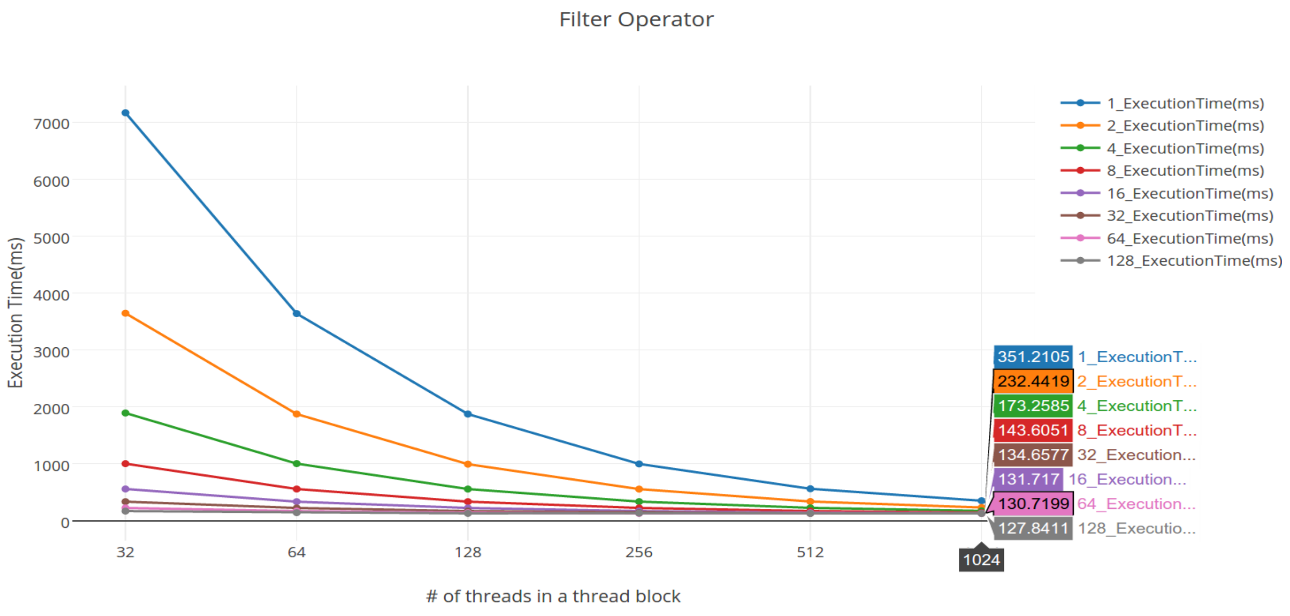

Figure 4.1: Execution time of filter operator on GPU

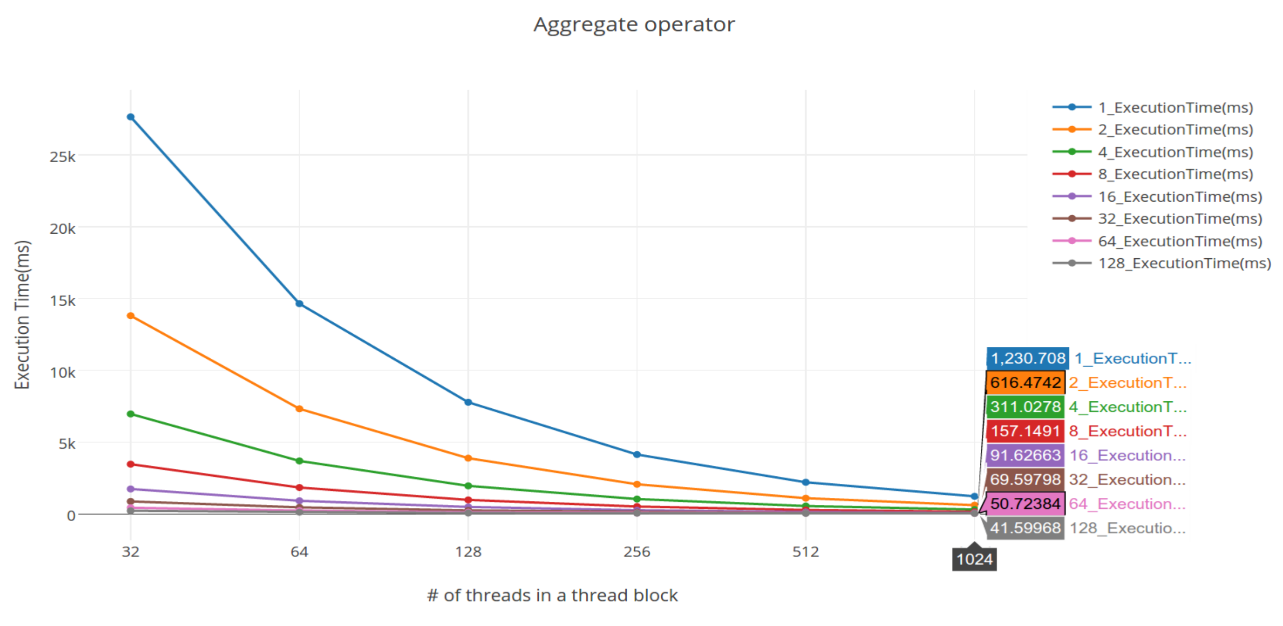

Figure 4.2: Execution time of aggregate operator on GPU

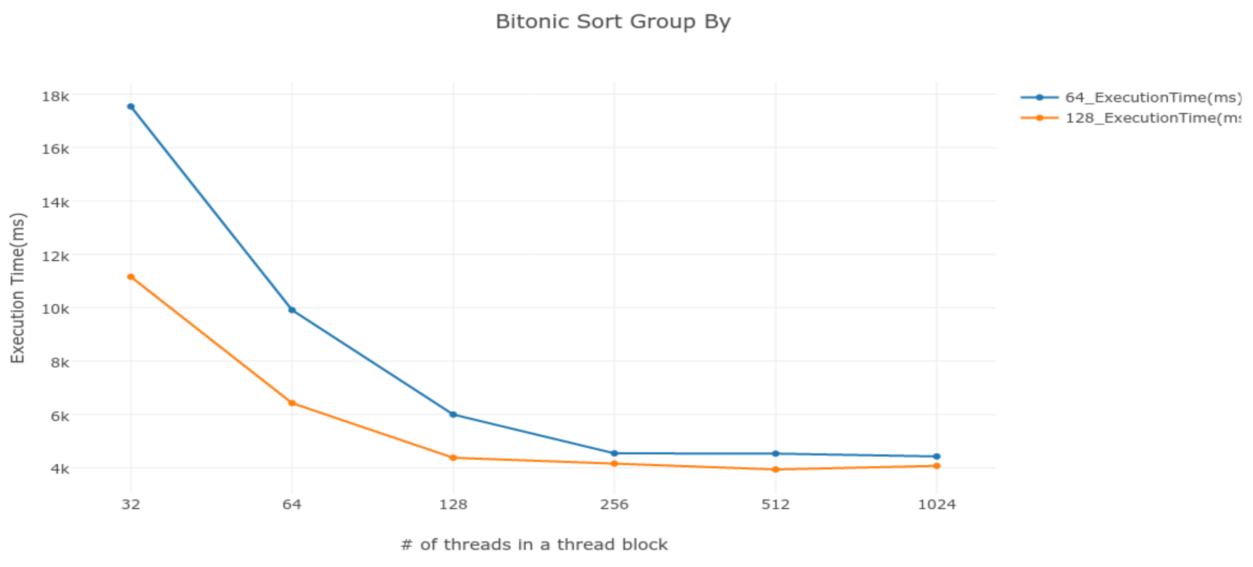

Figure 4.3: Execution time of bitonic sort group by operator on GPU

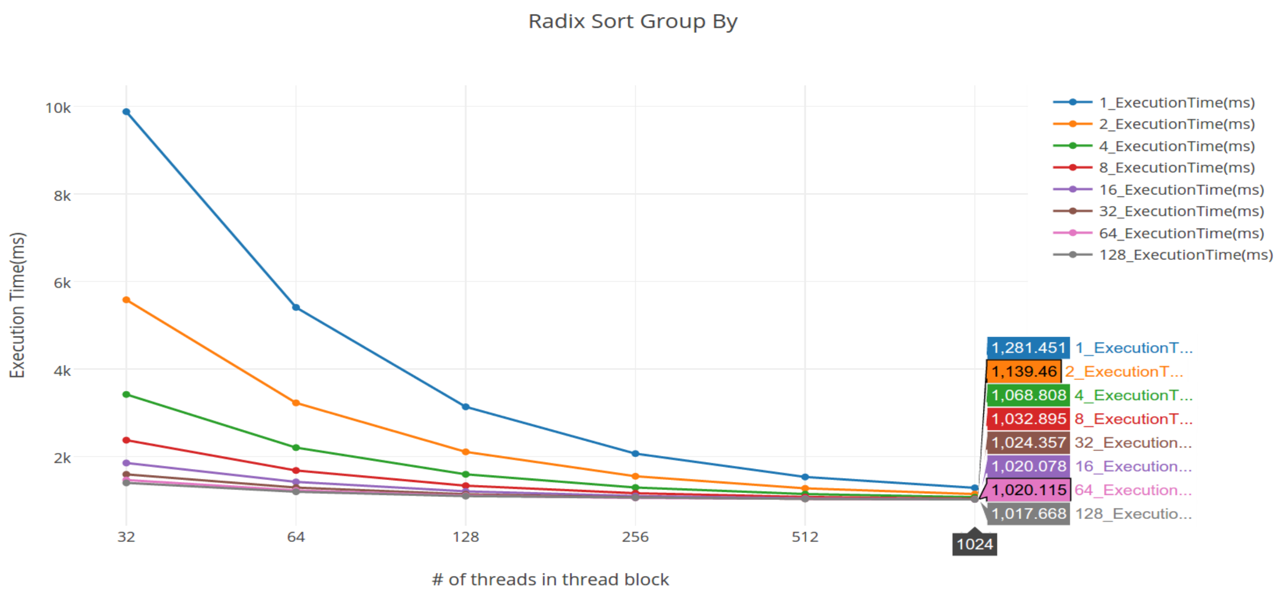

Figure 4.4: Execution time of radix sort group by operator on GPU

4.11 Performance of TPC-H Queries

Whenever a query is fired on the database engine to execute, it first parse the query and then create a physical execution plan tree for the query. The execution plan tree is created by performing cost analysis of each operator to be executed in the plan tree. Our objective in this project is to execute the *given physical execution plan tree* on the heterogeneous architecture as fast as possible. Following are the five ways to execute the plan tree on discrete heterogeneous architecture.

1. **Sequential execution on CPU (*SEC*)**: In this each node of plan tree is executed on the CPU one after the completion of another. The nodes are executed in topological order of the tree.

2. **Sequential execution on GPU (*SEG*)**: In this each node of plan tree is executed on the GPU one after the completion of another.

3. **Concurrent execution on the GPU (*CEG*)**: In this we divide the tree into set of streams, then execute the stream concurrently on GPU. Currently the kernels of the algorithms like hash join, filter are not resource configurable. Hence each operator node in tree consumed all the resources and didn't allow any other operator node to execute concurrently with it on the GPU. Therefore performance of CEG is similar to SEG.

4. **Sequential Execution on CPU-GPU (*SEH*)**: We first executed the plan tree on the GPU & CPU sequentially. Whenever a node is outsourced to CPU, GPU has to transfer the input data of the node to host memory then it has to wait for the execution of the node on the CPU. After execution of node on CPU, CPU transfers output data to device memory and then GPU starts execution on output data. We apply following greedy rule to outsource the node to CPU. Nodes whose summation of execution time on CPU, transfer time of its input data from device to host memory and transfer time of its output data from host to device memory is lesser than execution of nodes on GPU, were outsourced to CPU so that overall execution time reduces. Ideally to calculate to execution time, a cost formulae for the operator should be built which takes into consideration algorithm used for the operator, input data size and input data distribution. Similarly for calculating transfer time we should estimate the input and output cardinality of the outsourced operator accurately.

5. **Concurrent execution on CPU-GPU (*CEH*)**: It is combination of *CEG* and *SEH*.

   (a) We divide the tree into set of streams.

   (b) We follow greedy rule mentioned in *SEH*.

   (c) Execute the streams concurrently on GPU.

   In *CEH*, if GPU has some operations to execute in other stream then GPU does not have to wait for memory transfer from GPU to CPU, CPU to execute node and memory transfer from CPU to GPU i.e. nodes are concurrently executed on CPU or GPU.

It is truly unfair to run same execution plan tree on CPU and GPU separately and compare the execution times with respect to each other. Because they both might require different plans to execute optimally. Therefore *SEC* and *SEG* should not been compared with each other. The physical execution plan tree for modified TPC-H queries are shown in Figure 4.5, Figure 4.6 and Figure 4.7 respectively. The performance of modified TPC-H queries are shown in Table 4.8, 4.9 and 4.10.

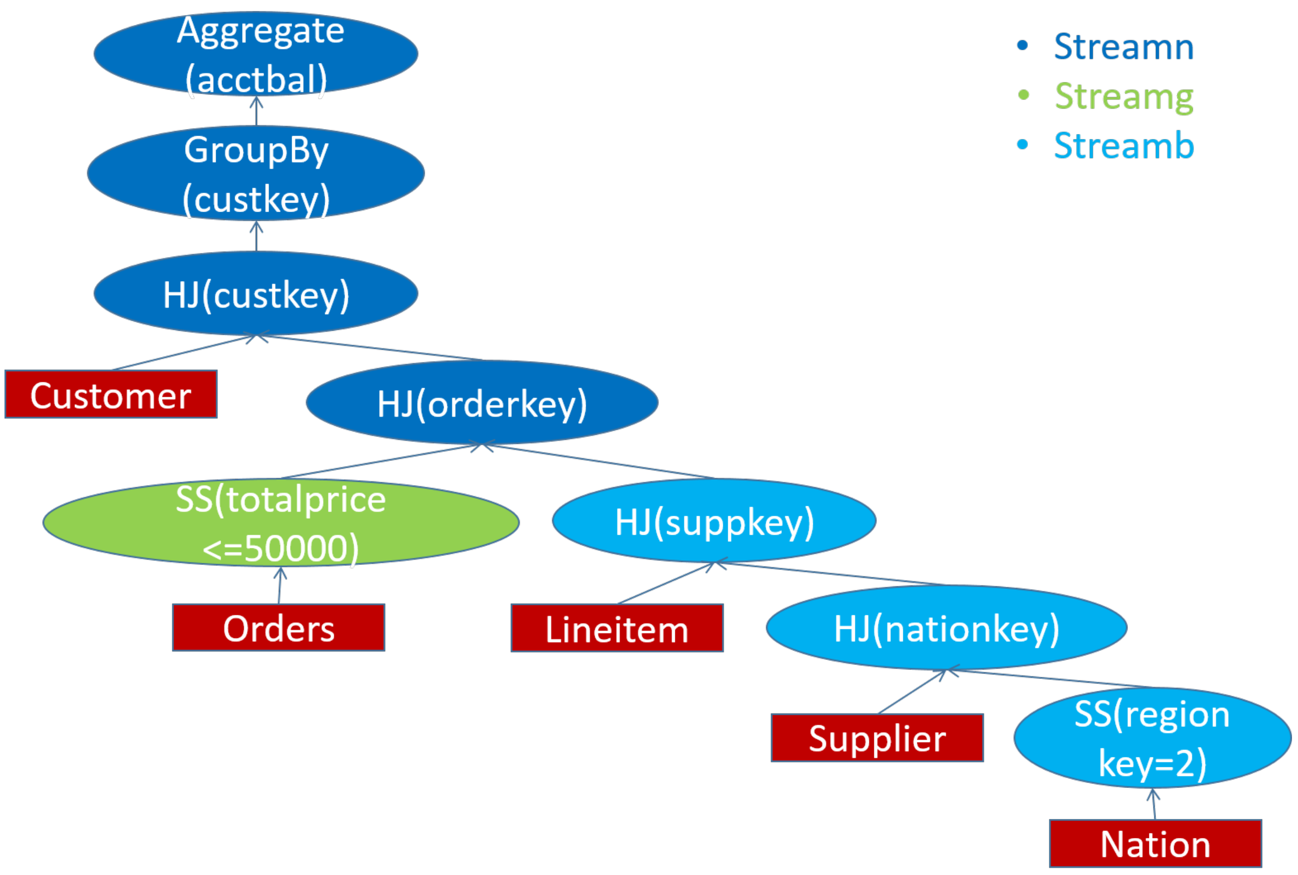

Figure 4.5: TPC-H Query 5 execution plan tree for CPU and GPU

	Time(ms)
MonetDB	830
SEC	3200
SEG	1581
SEH	Not Beneficial
CEH	Not Beneficial

Table 4.8: Performance of TPC-H Q5 on CPU(6 threads), GPU(256 threads, 90 thread blocks)

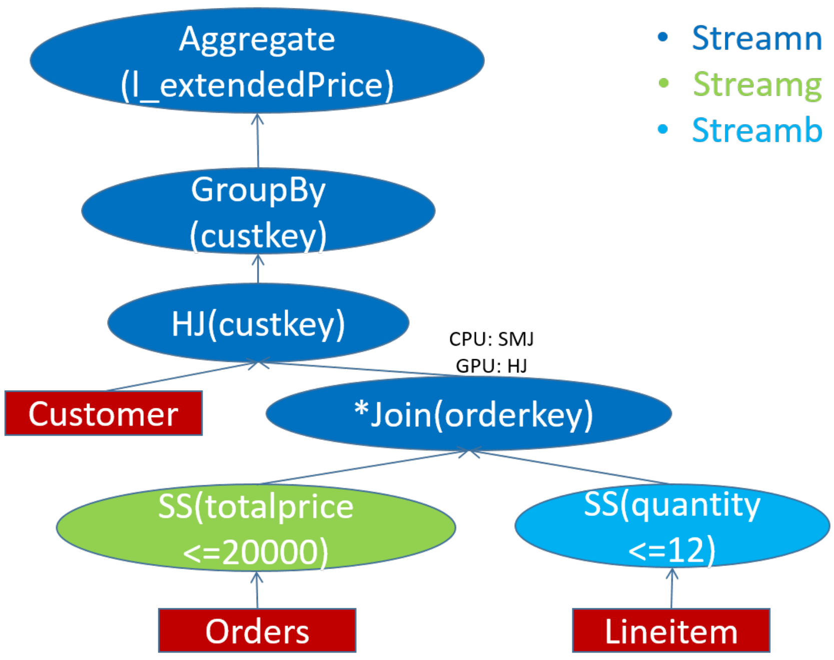

Figure 4.6: TPC-H Query 10 execution plan tree for CPU and GPU

	Time(ms)
MonetDB	580
SEC	750
SEG	720
SEH	Not Beneficial
CEH	Not Beneficial

Table 4.9: Performance of TPC-H Q10 on CPU(6 threads), GPU(1024 threads, 16 thread blocks)

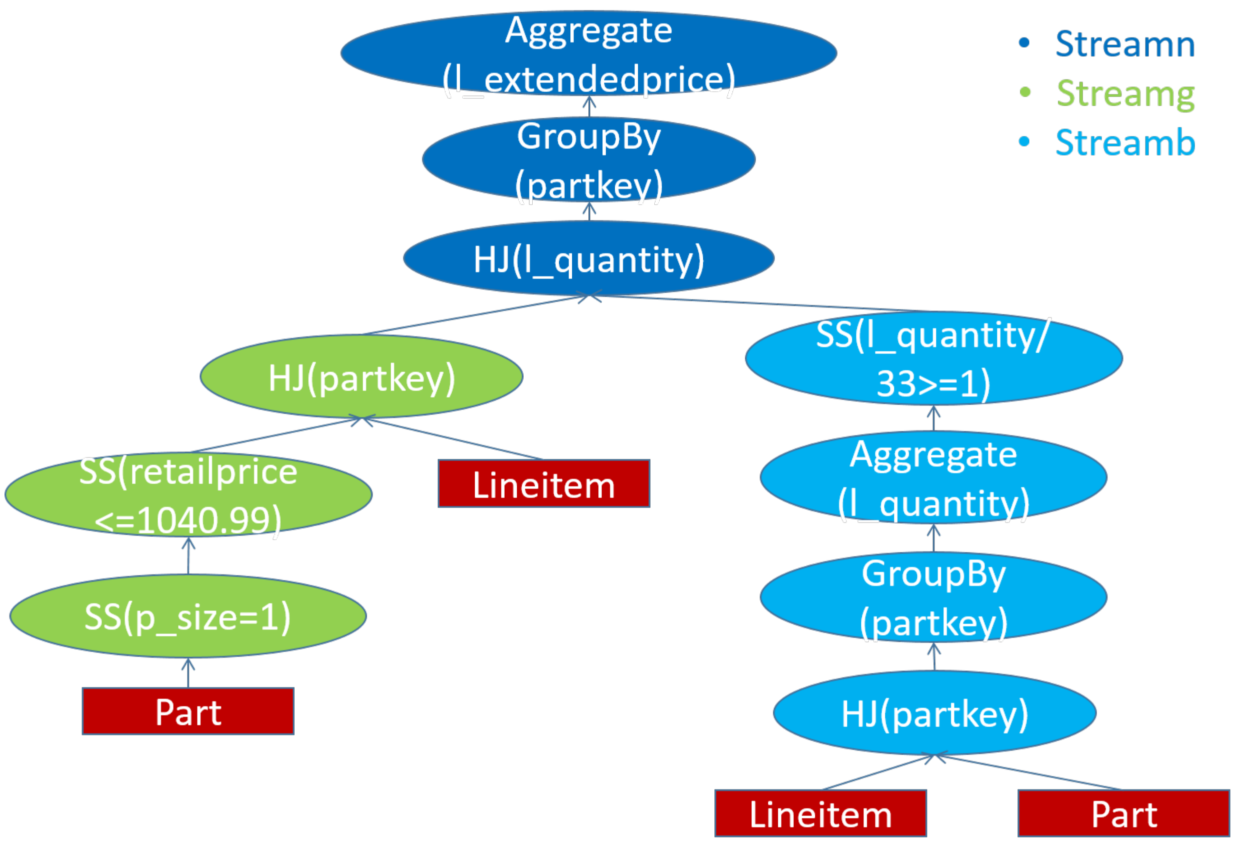

Figure 4.7: TPC-H Query 17 execution plan for CPU and GPU

	Time(s)
MonetDB	11.5
SEC	13
SEG	21
SEH	8
CEH	6

Table 4.10: Performance of TPC-H Q17 on CPU(6 threads), GPU(1024 threads, 30 thread blocks)

Observations:

- *SEH* & *CEH* of TPC-H 5, TPC-H 10 is not beneficial because GPU is performing either better or close to CPU on all the operator nodes of their respective tree.

- In *SEH* of TPC-H 17: Hash join in $Stream_g$ was outsourced to CPU as this join was taking 15s to execute on GPU. All the other nodes were executed on the GPU.
- In *CEH* of TPC-H 17: when the hash join of $Stream_g$ was outsourced to CPU then GPU has kernels to execute on $Stream_b$. Therefore *CEH* has less execution time than *SEH*.
- The execution time of *SEC* is similar to MonetDB execution time proves that the choice of execution plan tree is not too bad for TPC-H 10 and TPC-H 17.

Chapter 5

Conclusion and Future work

5.1 Conclusions

1. We propose the algorithm to cut down the tree into set of streams which can execute concurrently on the GPU.

2. We analyse the scheduler of Tesla k40m GPU and conclude that the order in which stream are activated has an effect of overall execution of the plan tree.

3. With the help of multiple streams, we were able to execute the operators on the GPU concurrently with CPU which reduced the overall execution time of the TPC-H query 17.

5.2 Future work

1. The assignment of the operator to the processing unit used in *CEH* is greedy algorithm. The greedy rule is not optimal for *CEH*. Therefore it is still an open problem to design an algorithm which do optimal assignment of the operators on the CPU or GPU.

2. Designing an algorithm which assigns optimal number of thread block to each node of the plan tree so that total execution time of the plan tree on *CEG* is reduced. The assignment should be such that both the child nodes of parent node should complete in same time because their is no benefit for one child node to finish significantly fast and other significantly slow because parent node had to wait for both of its child node.

3. Hash join in CUDA implementation is not resource configurable i.e number of thread block launched by the kernels of hash join is dependent upon the data size which is typically large enough to make number of thread blocks launched much greater than max residency

thread block of that kernel. Therefore two hash joins sent on different streams will not overlap each others executions. Hash Join should be made resource configurable.

4. Join algorithm such as INLJ, NLJ, SMJ are yet to be integrated into system and made to be resource configurable.

Bibliography

[1] Bingsheng He, Mian Lu, Ke Yang, Rui Fang, Naga K. Govindaraju, Qiong Luo, Pedro V. Sander. "Relational query co-processing on graphics processors." ACM Transactions on Database Systems (TODS) 34.4 (2009): 21.

[2] J. He, S. Zhang, and B. He. In-cache query co-processing on coupled CPU-GPU architectures. Proc. VLDB Endow., 8(4):329340, Dec. 2014 1

[3] `http://on-demand.gputechconf.com/gtc-express/2011/presentations /StreamsAndConcurrencyWebinar.pdf` 1

[4] He, Bingsheng, et al. "Relational joins on graphics processors." In Proceedings of the 2008 ACM SIGMOD international conference on Management of data. ACM, 2008. 1

[5] Manegold, S., Boncz, P. and Kersten, M.L., 2002, August. Generic database cost models for hierarchical memory systems. In Proceedings of the 28th international conference on Very Large Data Bases (pp. 191-202). VLDB Endowment.

[6] `http://www.cse.ust.hk/gpuqp/gdb.zip` ii, 20

[7] `http://www.nvidia.in/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture -Whitepaper.pdf`

[8] `http://www.tpc.org/TPC-H` 16, 17, 23

[9] `https://developer.nvidia.com/thrust` 1, 16
17, 18
22