國立交通大學

資訊科學與工程研究所

碩 士 論 文

一 個 以 繪 圖 處 理 器 為 基
礎 之 記 憶 體 資 料 庫 實 作

Implementation of a GPU Based Main Memory Database

研 究 生：徐竣傑

指導教授：袁賢銘　教授

中 華 民 國 九 十 八 年 八 月

一個以繪圖處理圖為基礎之記憶體資料庫實作
Implementation of a GPU Based Main Memory Database

研　究　生：徐竣傑　　　　Student：Jyn-Jie Hsu

指導教授：袁賢銘　　　　Advisor：Shyan-Ming Yuan

國 立 交 通 大 學

資 訊 科 學 與 工 程 研 究 所

碩 士 碩 文

A Thesis

Submitted to Department of Computer and Information Science

College of Electrical Engineering and Computer Science

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Master

in

Computer and Information Science

August 2009

Hsinchu, Taiwan, Republic of China

中華民國九十八年八月

一個以繪圖處理圖為基礎之記憶體資料庫實作

研究生：徐竣傑　　　　　指導教授：袁賢銘

國立交通大學資訊科學與工程研究所碩士班

摘要

電腦網路的發達帶來了電腦之間資料的快速交換，資料庫扮演著相當重要的角色。近年來，NVIDIA 致力於 GPGPU 的發展，一個高度平行化的發展平台 CUDA 就此產生。使用者利用熟悉的 C 語言就可以在上面開發自己的應用程式。加上記憶體空間的快速成長，已經足夠一個資料庫的使用。因此，我們在 GPU 的記憶體上面實作了一個實驗性的資料庫，並觀察 GPU 的計算能力，如何改善一般資料庫的操作效能。

根據圖像處理單元(GPU)的特性，我們將資料庫中所有的資料儲存在繪圖卡上的記憶體中。主機上的 CPU 處理一些流程的控制，而各個功能的計算則交給 GPU 來處理。最後，我們與著名的資料庫 SQLite 記憶體資料庫做效能上的比較。根據我們的實驗結果，在總資料數固定下，當查詢結果數超過一定的程度時，我們的資料庫會有相對較佳的效能，我們稱之為轉折點。最後，我們觀察在不同資料總數下的轉折點，歸納出在不同的功能下，查詢結果中資料數佔總資料數為 0.161%~2.161%時，我們所實作的資料庫效能上會超過 SQLite。

Implementation of a GPU Based Main Memory Database

Student: Jyu-Jie Hsu Advisor: Shyan-Ming Yuan

Department of Computer Science and Engineering

National Chiao Tung University

Abstract

Because the development of computer network brings rapidly data exchanging between computers, data base is playing the quite important role. In the last years, NVIDIA has worked on the development of GPGPU, and a platform of parallel computing, CUDA, was provided. Users can design their own application using the familiar program language, C. Additionally, the growth of memory makes the feasibility of main memory data base, and so we implemented an experimental memory data base on GPU memory for observing how the computation power of GPU can improve common operations of data base.

According to the features of GPU hardware, we stored all records of data base in the memory of graphic card. The control flows handled by host CPU and the computations of each function handled by GPU. Finally, we compare the performance of our data base with SQLite memory data base. The experiment result shows that there is a turning point denotes a number of records in query result (records queried). The performance of our data base is better than SQLite memory data base while the number of records queried exceeds the number denoted by turning point. Finally, we figured out a ratio of data queried to total number of data according to the observation of the turning points in different functions. Our experimental DB has better performance than SQLite as long as the ratio exceeds 0.161%~2.161%.

Acknowledgement

首先要感謝我的指導老師袁賢銘教授，在研究期間不辭辛勞的撥空指導我研究的方向，並給我適當的建議，讓我能夠順利的完成研究。並感謝實驗室的學長們宋牧奇、林家鋒，在平時遇到研究上的困難時，給予我細部實作與研究經驗上的寶貴建議。也感謝實驗室的各位同學黎光明、洪偉翔、周東興、王志華、鄭婷文、葉秀邦、羅國亨，不管是在修課或研究上都一起討論和分享知識，互相激盪出不同的想法，讓我成長不少。最後要感謝辛苦養育我的母親和外婆，讓我一路順利的完成學業，讓我在求學與生活上沒有後顧之憂，能專心致力於研究，感謝你們大家。

Table of Contents

List of Tables

List of Figures

Chapter 1 Introduction

1.1 Preface

In the last years, data exchanging of applications is getting more necessary, particularly due to the proliferation of web data in various formats and the emergence of e-business applications that need to communicate data yet remain autonomous [14] . Responding to this demand, many data base management system have been built. Data base management system collects information and provides functions for easily querying, updating, and deleting data. There are many applications from a handhold device (ex: cell phone or PDA) to a server computer of webs have used data base to manage their information for requirements.

1.2 Motivation

The performance of data base management system is important for many applications such as a middle size web site normally has thousands of members, and the responding time of querying the data base must be small for the requirement of clients. Besides, the tool kits development of general purpose GPU (GPGPU) is getting more convenient to use. Developers have designed many algorithms and application on GPU for better performance.

A data table is a collection of records, and these records are compared one by one on query operation. Even though many non-candidate data are filtered by one comparison operation when an index structure is adopted, for example a b-tree, the complexity for each operation is still O (log n). GPU is a multi-core processor of SIMD architecture. Hundreds of threads can be issued while the GPU is running. For example a 9800 GT has 16MP and 8 streaming processor in each MP can issue 128 threads simultaneously. Implementation of data base on GPU may bring some degree of parallel computing for functions including data query, data sorting, and aggregated functions on each data groups. It is interesting how much performance of data base that GPU can improve.

1.3 Problem Description

We choose the CUDA tool kit as our programming platform. Using CUDA to developing applications has to face several constrains of GPU hardware limitation as followings:

(1) The limitation of registers and shared memory: the number of registers and shared memory in one multi-processor (MP) of GPU is finite. One MP is a unit of dispatching one block consists of several threads. The number of registers and shared memory used by threads dominate how many threads can be issued in one block.

(2) Divergent Branch: The branching within the same block could be expensive, if threads take different execution path, they must be serialized by the thread scheduler on GPU.

(3) The overhead of uploading data from CPU to GPU is large: If each data was uploaded to GPU when query was issued and downloaded after computing, the improvement of performance will be reduced.

(4) Non-Coalesced global memory access: Multiple global memory access is grouped into once memory access if the access pattern of half warp is sequential. It is called coalesced memory access. If the access pattern is not sequential, than multiple memory access are issued without grouping.

For (1), each thread use registers and shared memory as less as possible, and threads are issued as the same number of all records in a data table to exploit the computational power of GPU. The host CPU performs the partial flow control of our data base management system for the reason of (2). The entire data of our data base is stored in GPU memory for (3), which can avoid the overhead of uploading and downloading large amount of data for each query. Finally, our data are stored in data table in column major for coalesced memory access, because threads usually read/write the same column of records. When the address of the same column of records is sequential in memory space, coalesced memory access will easily occurs.

1.4 Research Objective

For the reason we mentioned above, we implemented several common functions of our data base on the GPU. We evaluated the execution time of our experimental data base and compared it with SQLite memory data base in order to observe the differences of behaviors and performance.

1.5 Research Contribution

According to the result of our experiment, there is one turning point between data base and SQLite memory database. We generalize each turning point for each function with different total number of records. The trends of each turning point changing are approximate linear variation. The turning points denote a ratio of number of data queried to total number of records in data table. The performance of our data base is always better then SQLite memory data base while the ratio is greater than the turning point.

Chapter 2 Background and Related Work

2.1 Graphic Processor Unit (GPU)

A graphics processing unit (GPU) is a processor that offloads 3D graphics rendering from the microprocessor. The primary reason that GPUs deliver such high performance is that the GPU is a highly parallel machine. GPUs keep these processors busy by juggling thousands of parallel computational threads. In theory, GPUs are capable of performing any computation that can be mapped to the stream computing model. This model has been exploited for ray-tracing [15], global illumination [16], matrix multiplies [17] , geometric computations [18] and the other applications ([25][26][27][28][29][30]).

The hardware evolutions of GPU from fix function unit to programmable pipeline consist of multiple SIMT processor satisfied not only graphic processing but also general purpose demands. In a programmable pipeline model, there are many extensions of OpenGL added by OpenGL ARB for user programming purpose. Developers treat a compute-intensive problem as multiple pieces, and compute data as pixel-rendering process to solve the whole problem [7][12]. The comparison of computation power between CPU and GPU was depicted as Figure 2 - 1.

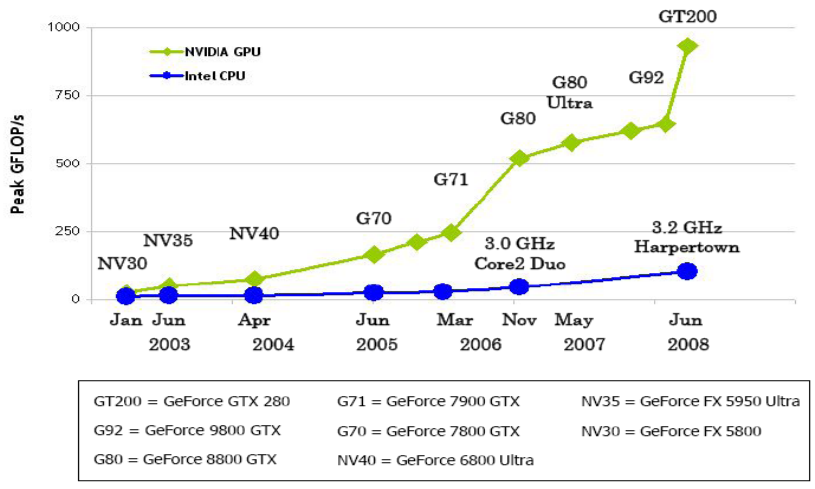

Figure 2 - 1 The comparison of computation power between CPU and GPU.[11]

The primary difference between CPU and GPU is that the GPU is specialized for rendering graphics, highly parallel computations. As illustrated by Figure 2 - 2 GPU devotes more transistors to data processing rather than data caching and flow control. GPU executes efficiently as a problem can be expressed as parallel-data computation. Unfortunately, mapping our algorithms or problems to OpenGL as pixel-rendering processing is not straightforward. Now, there are two biggest GPU manufacturers, NVIDIA and AMD/ATI, working on providing program interfaces, CUDA and CTM, to make user develop their own program easier. We will introduce them later.

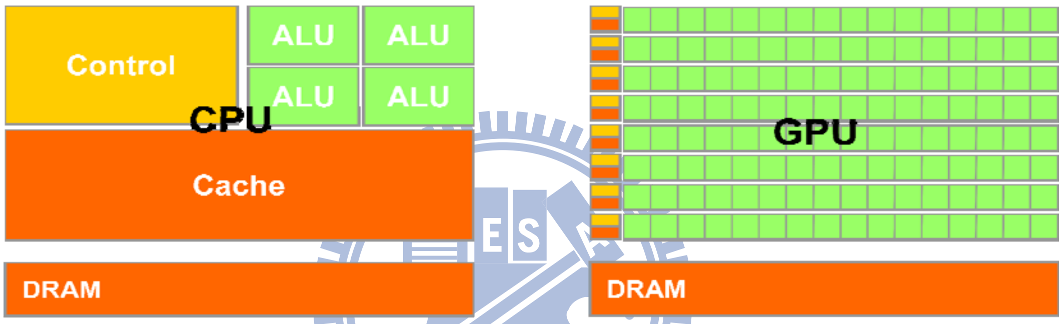

Figure 2 - 2 Layout of CPU and GPU[11]

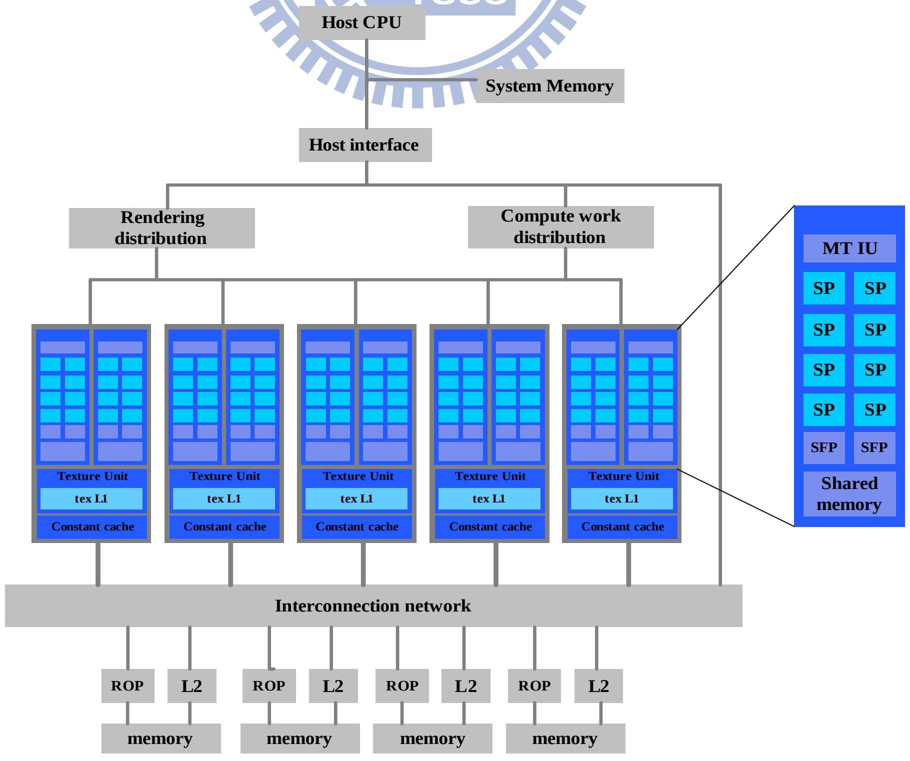

Figure 2 - 3 Architecture of Telsa GPU[8]

As showed in Figure 2 - 3, a multi-processor (MP) has 8 streaming process which can perform one single precise floating point computation [9]. There 16 MPs in 9800GT and can perform 128 single precise floating point computations simultaneously [5].

2.2 Compute Unified Device Architecture by NVIDIA

CUDA is a general purpose parallel computing architecture including a new parallel programming model and an instruction set architecture. The CUDA programming model allows developers to exploit that parallelism by writing natural, straightforward C code that will then run in thousands or millions of parallel invocations, or threads[6] [8]. Source files include a mix of host code (i.e. code that executed on the host) and device code (i.e. code that executed on the device). When running a CUDA program, developers simply run the program on the host CPU. The CUDA driver automatically loads and executes the device programs on the GPU [11]

In the CUDA programming model, the GPU is treated as a co-processor onto which an application running on a CPU can launch a massively parallel compute kernel. This is called massively threaded architecture [2]. The kernel is comprised of a grid of scalar threads. Each thread is given a unique identifier (thread ID) which can be used to help divide up work among the threads. Within a grid, threads are grouped into *blocks*, which are also referred to as *cooperative thread arrays* (CTAs). Within a single CTA threads have access to a common fast memory called the *shared memory* and can, if desired, perform barrier synchronizations [9].

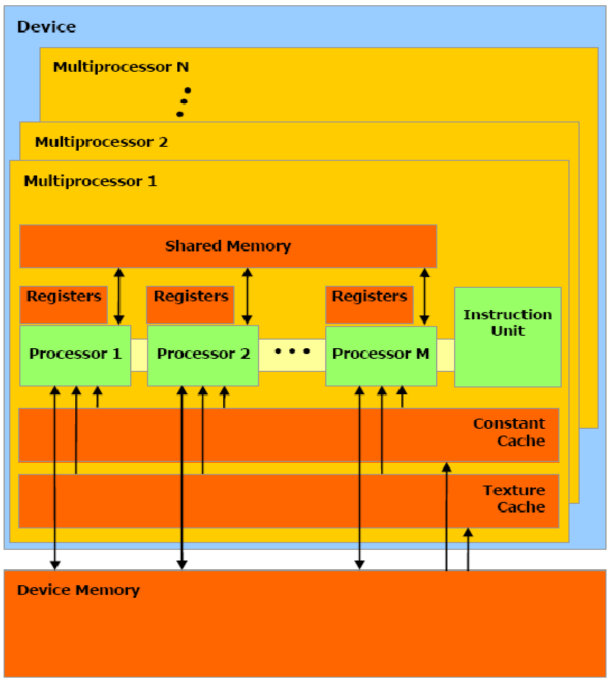

Figure 2 - 4 A set of SIMT multiprocessors with on-chip shared memory [10]

Threads running on the GPU in the CUDA programming model have access to several memory regions including on chip memory, registers, shared memory, constant cache, texture cache, and off chip memory, global memory constant memory and texture memory [12].

As illustrated by Figure 2 - 4 each multi-processor can access on/off chip memory of the several types as following:

Table 2 - 1 Classification of GPU Memory

Name	**Accessibility**	**Scope**	**Speed**	**Cache**	**factor**
Registers	read/write	per-thread	immediate (on chip)	X	--
Local Memory	read/write	per-thread	400~600 clock	N	Compiler Auto
Shared Memory	read/write	per-block	4 clock (on chip)	N	Memory conflict
Global Memory	read/write	per grid	400~600 clock	N	Non-coalescing
Constant Memory	read only	per-grid	4 or 400~600	Y	Cache miss
Texture Memory	read only	per-grid	4 or 400~600	Y	Cache miss

Device memory consists of constant memory, texture memory, global memory, and local memory. Constant memory and texture memory are read-only regions of device memory and can be allocated before calling kernel functions. The on chip caches, constant cache and texture cache (read only), have brought significant performance increasing of both memories.

Threads can read/write the global memory directly, but there is no cache supported for global memory. Finally, automatic variables that are likely to be placed in local memory are large structures or arrays that would consume too much register space, and arrays for which the compiler cannot determine that they are indexed with constant quantities.[11] Because the local memory space is not cached, the usage of local memory should be avoided. The overhead of accessing local memory is large

To improve memory system efficiency, it thus makes sense to group accesses from multiple, concurrently issued, scalar threads into a single access to a small, contiguous memory region. The CUDA programming guide indicates that parallel memory accesses from every half-warp of 16. That is called memory coalescing**.** Coalescing is achieved for the pattern of sequence addresses requested by the half-warp. In GeForce 9800 GT (G92), if a half-warp addresses words in 16 different segments, 16 memory transactions are issued (one for each segment). One coalesced memory access results in a single memory transaction not 16 memory transactions. Example of coalesced memory access patterns is depicted as Figure 2 - 5.

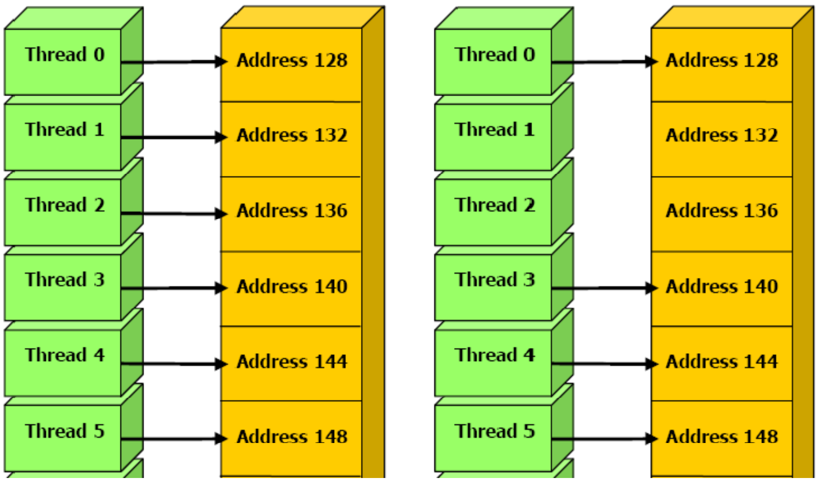

Figure 2 - 5 Patterns of coalesced memory access (left or right)[11]

2.2.1 Parallel Prefix Sum and Sorting Algorithm

CUDPP is the CUDA Data Parallel Primitives Library. CUDPP is a library of data-parallel algorithm primitives such as parallel prefix-sum (scan) [4] , parallel sort [13] and parallel reduction. Because the original sorting algorithm didn't support key-value pair sorting, we have to modify the sorting algorithm for key-value pair sorting.

The **prefix sum** (also known as the **scan**) is an operation on lists in which each element in the result list is obtained from the sum of the elements in the operand list up to its index [3]. There are two kinds of prefix sum, exclusive prefix sum and inclusive prefix sum. In exclusive prefix sum, the first element in the result array is identity (0 for following operation) and the last element of the operand array is not used; whereas inclusive prefix sum, all elements in operand array are used.

In chapter 3, we will discuss how to use prefix sum to calculate the position of selected data and use the sorting algorithm of modified version to sort entire data table.

2.3 Close To Metal (CTM) by AMD/ATI

Abbreviated as CTM, *Close to Metal* is an ATI device that is designed to expose the parallel array of floating-point processors found in ATI graphics hardware. Compared to CUDA, CTM has much lower level programming style, assembly-like, than CUDA without the comprehensive toolkits, compiler, or high-level C language construct. CTM is controlled with a few commands to set parameters, invalidate and flush caches, and start the processors in the processing array. It's not easy to mapping applications to CTM program for developers, therefore CUDA is much popular than CTM.

2.4 Main Memory Database

A main memory database (MMDB; also In-memory database system or IMDB) is a database management system that primarily relies on main memory for computer data storage. Traditional databases are built to store data on disk. Disk I/O, as a mechanical process, is tremendously expensive in terms of performance. One approach to achieving high performance in a database management system is to store the database in main memory rather than on disk. One can then design new data structures aid algorithms oriented towards making efficient use of CPU cycles and memory space rather than minimizing disk accesses and using disk space efficiently [1]. As semiconductor memory becomes cheaper and chip densities increase, it becomes feasible to store larger and larger databases in memory, making MMDB's a reality. A computer's main memory clearly has different properties from that of magnetic disks as following:

1. The access time for main memory is orders of magnitude less than for disk storage.
2. Main memory is normally volatile, while disk storage is not. However, it is possible (at some cost) to construct nonvolatile main memory

In addition, index structure of data table affects the performance of database operations. The B-tree (or the B+-tree) is the most popular index structure in current database systems. In a tree, records are stored in locations called leaves. The starting point is called the root. The maximum number of children per node is called the **order** of the tree. The maximum number of access operations required to reach the desired leaf (data stored on the leaf) is called the **depth** (level). The bigger the order, the more leaves and nodes you can put at a certain depth. This means that there are fewer levels to traverse to get to the leaf (which contains the data you want). There several properties of B-tree as following: The root is either a leaf or it has at least two non-empty sub-trees and at most m non-empty sub-trees. Following we discussed are the properties of the three main operations of B-tree search, insertion, and deletion.

1. **Search**: The algorithm is similar to binary search tree. Starting at the root, the tree is traversed top to bottom, choosing the child pointer whose separation values are on either side of the value that is being searched. Binary search is typically (but not necessarily) used within nodes to find the separation values and child tree of interest.

2. **Insertion**: In insertion a B-tree undergoes changes that must maintain**:**

    i. Its height balance.

    ii. Its leaves to be at the same level.

    iii. Each of its nodes, except the root, to be at least half full (i.e., to contain a minimum of ⌈m / 2⌉ - 1 keys, where m is the order of the tree).

3. **Deletion**: Like insertion, deletion must be on a leaf node. If the key to be deleted is not in a leaf, swap it with either its successor or predecessor.

An important observation is that the number of preliminary operations for each of the major functions (search, insert, delete) in B-tree can be done in time proportional to the height of the B-tree, which is no more than O(1+log(n+1)). Besides, the sequence access performance of B-tree

Increased attention has been given on redesigning traditional database algorithms for fully utilizing the available architectural features and for exploiting parallel execution possibilities, minimizing memory and resource stalls, and reducing branch miss predictions.[19][20][21][22][23].

2.5 Related Works

2.5.1 Implementing Database Operation Using SIMD Instructions

In the 2002, the paper “Implementing Database Operation Using SIMD Instructions” proposed and implemented several algorithms using SIMD Instructions for Data base operations. For this particular instruction, both operands are using 128-bit register. Each source operand contains four 32-bit single-precision floating-point values, and the destination operand contains the results of the operation performed in parallel on the corresponding values in each operand. The result showed that using a SIMD parallelism of four, the CPU time for the proposed algorithms is from 10% to more than four times less than for the traditional algorithms.

2.5.2 Fast Computation of Database Operations using Graphics Processors

In the 2004, the paper "Fast Computation of Database Operations using Graphics Processors" [31] implemented several operations of data base on NVIDIA GeForce FX 5900 without CUDA tool kits. Data was stored on the GPU as textures and used alpha, stencil and depth test unit in pixel processing unit to perform corresponding algorithms. The result shows the GPU implementation of operations is about 2 times faster than CPU implementation.

Chapter 3 Building a Data Base on GPU Memory

In this chapter, we listed several functions of conventional database which are not parallel processing on CPU, and we proposed architecture and implemented them for our data base system. Through exploiting multi-threading of GPU, we can divide a large data set to a small data unit executed on each GPU processing core. We implemented an experimental data base and store entire data on GPU memory (called GPU DB). The implementation methods of GPU DB are described in section 3.3 in detail.

3.1 Architecture of the Data Base

As Table 3 - 1 listed the common functions of Data Base that we implement on GPU.

Table 3 - 1 Implemented functions in our data base

Function Name	Corresponding SQL Language Example

Selection Query	SELECT store_name
	FROM Store_Information
	WHERE Sales > 1000

Selection Query and Sorting Data	SELECT store_name, Sales, Date
	FROM Store_Information
	ORDER BY Sales

Selection Query and Data Grouping operations(SUM, MAX, MIN, COUNT, AVG)	SELECT store_name, SUM(Sales)
	FROM Store_Information
	GROUP BY store_name

Data Insert	INSERT INTO Store_Information
	(store_name, Sales, Date)
	VALUES ('Los Angeles', 900, 'Jan-10-1999')

Data Insert According to Selection Query	INSERT INTO Store_Information
	(store_name, Sales, Date)
	SELECT store_name, Sales, Date
	FROM Sales_Information
	WHERE Year(Date) = 1998

Data Update	UPDATE Store_Information
	SET Sales = 500
	WHERE store_name = "Los Angeles"

	AND Date = "Jan-08-1999"

Data Delete	DELETE FROM Store_Information
	WHERE store_name = "Los Angeles"

The data base system consists of two different device, host computer and GPU device. Because GPU communicates with CPU through PCI-Express bus; the cost to download/upload data from/to GPU would be expensive. In the initial stage, all data and tables was loaded to GPU memory via main memory of host computer and then memory spaces were freed after loading. Our data base system keeps information and pointers of tables on main memory for data base manipulation. The entire architecture was depicted in Figure 3 - 1. Because primary functions are executed on GPU, overhead of communication between GPU and CPU is avoided.

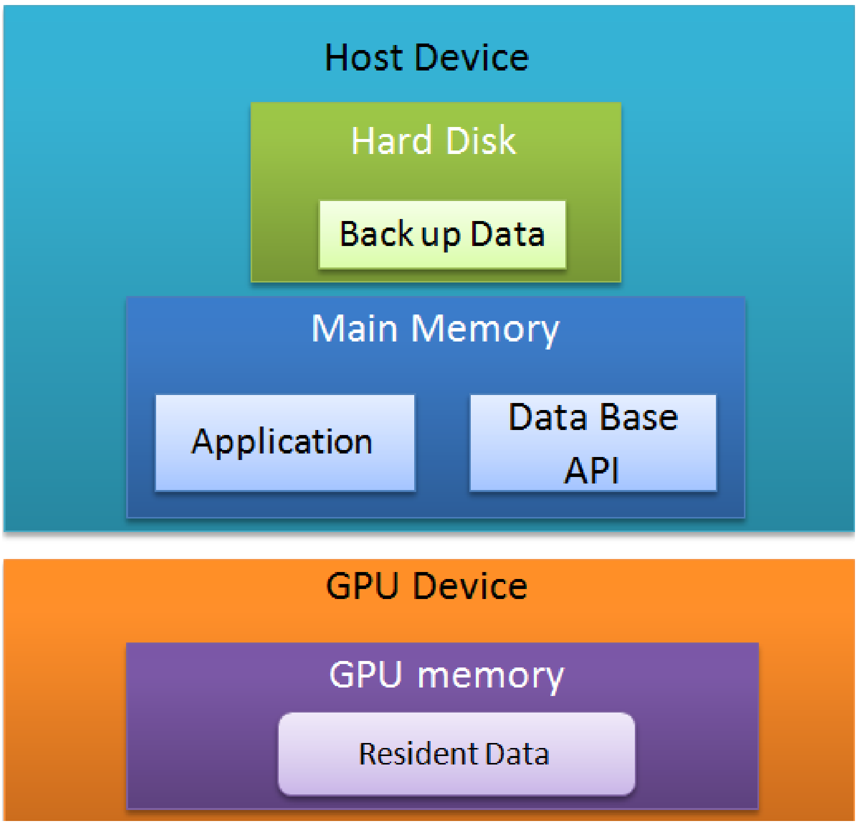

Figure 3 - 1 Architecture of entire system

We also implemented table management functions, Create Table and Drop Table. Because table management functions are not impotent to our research, and the methods are trivial, we will not describe them in this paper.

3.2 Modify the CUDPP Library

CUDPP offers a serial efficient library to developers. Several important algorithms are implemented in these libraries, for instance parallel prefix sum algorithm and parallel sorting algorithm. Unfortunately, the sorting algorithm is not available for sorting entire table according to the value of one of the columns

Sorting algorithm of CUDPP can sort a 1-D array as input, and output a 1-D array as a result. We modified the sorting algorithm such that outputs are two 1-D arrays, one of them is sorted data, and the other is a index array denotes original position of data after sorting. Thus, we can move the data of the other columns according to the index after executing sorting process to sort entire table.

The modified sorting function has a 1-D array as an additional parameter. Before calling sorting function, the additional 1-D array is initiated as sequential increasing numbers. Assuming there are N elements data should be sorted, then the additional 1-D array will be set numbers {0…N-1} as the initial index of data.

During the modified function sorts the N elements data, it moves both data and the index value of the additional array to their corresponding position. After sorting process, the index array point to original position of each data. Thus all records in data table can be sorted according to this index array.

3.3 Functions of Data Base on GPU

In this section, we will focus on how the functions listed in section 3.1 be implemented in detail.

3.3.1 Data Structure

At first, conventional data base uses tree structure to manage data or indices. Obviously, B-tree searching algorithm is not appropriate in parallel computing architecture. Unparallel computing in each core can’t bring the potential computing capability of GPU into full play.

The access to device memory usually takes up to 200~300 clocks, which is relatively slow to on-chip memory. Searching in B-tree will bring too many access times to device memory that attacks the performance of GPU. Normally, tree data structure is constructed by link list nodes. Link list nodes are connected by pointers, so achieving continuous memory access is not easy. Non-continuous memory access reduces the opportunity of coalesced memory access.

In order to solve problems we mentioned before, array is used as a data structure in GPU memory which stores tables and temporal data during computation. Searching in array can be easier than tree structure. An array supports random access. Data can be compared in parallel according to thread ID.

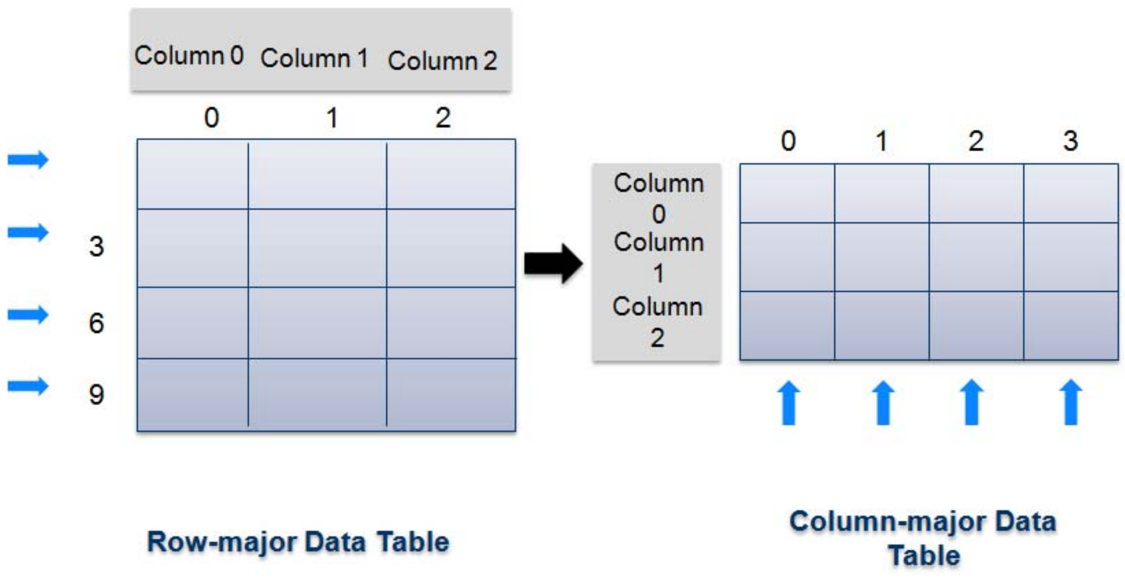

Figure 3 - 2 Data structure of data table.

Second, to reduce the complexity of design, a column is usually a unit of parallel computation in data base. Because the address of GPU memory is in row-major order, storing our tables in row major order will increase the opportunity of non-coalesced memory access. To avoid this problem, tables are stored in column major in GPU memory.

Figure 3 - 2 Data structure of data table. shows the relationship between records and memory

address. The data of entire column can be easily compared with conditions by storing data table in column major. Threads performs the same operation in each data and sequentially access memory that increases the opportunity of coalesced memory access.

3.3.2 Selection Query

At first, let's consider how Selection Query works in basis. After condition parameters are set, the process compares each column of every record. Assume all columns of one record meet all condition parameters; the record is selected and then set "1" to the corresponding position of flag array. Finally, we check the flag array of table, then copy these records marked and to another table as a result. To parallelize Selection Query process, one thread is assigned to one record. Each thread compares all columns of their assigned record iteratively, but all threads compare each record in parallel.

Acutely, there are two perplexities we have to consider before starting the implementation work:

1. In logical operation, the priority of "AND" and "OR" are different. The different priorities determine which operator supposed to be executed fist. In common case, the priority of "AND" is higher than "OR", so "AND" operator supposed to be executed first.
2. The branching within the same block could be expensive as they are executed on a SIMD processor, where only one instruction can be performed (with multiple different data source). So if threads take different execution path, they must be serialized by the thread scheduler on GPU (divergence branch).

To solve these perplexities, the ordinary prefix notation has to be transformed into postfix notation before starting process. Each record needs one stack used by threads during the postfix notation is processed. In Figure 3 - 3, threads compared each record with condition parameters set by user and manipulate stacks to calculate which record is selected.

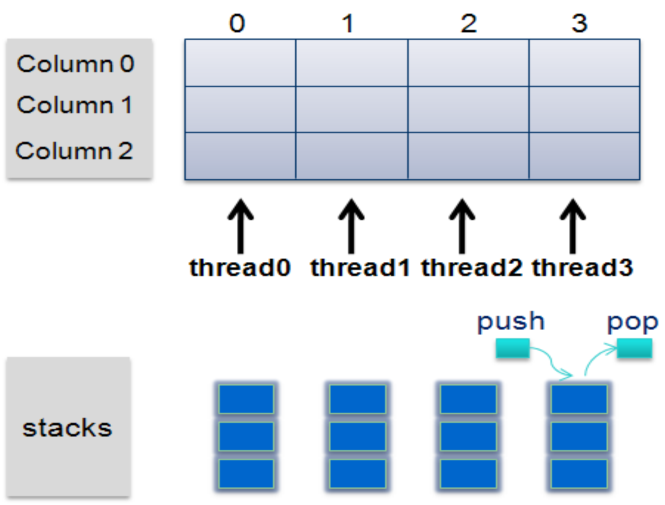

Figure 3 - 3 Process of Selection Query

Because the branching within the same block could be expensive, host computer should be responsible for the partial flow control to avoid divergence branch.

As entire thread finished, the value in the bottom of each stack denote which record was selected. Then, we plan to copy this selected records and move to another 2-D array as a final result and transmit it to host. Now we are focus on what the position of these selected records in the 2-D Array is correct.

This problem is easy to solve by using the function provided by CUDPP, cudppScan. The cudppScan performs a prefix sum operation on the flag array in GPU memory and outputs the array of corresponding position. Beside, the number of total selected records is the last value of the output which determines the size of result table. The concept of Selection Query we implemented is described as Figure 3 - 4, Figure 3 - 5, Figure 3 - 6.

```
Algorithm SelectionQuery_SetSelectFlag(QueryTable, QueryData, selected_flag )
Input: QueryData (denotes what data supposed to be selected)
Output: selected_flag (an array of flags denotes which record is selected)
Begin
    declare stack_d[][];
    declare top_d[];
    declare integer cnt;
```

```
cudaMemoryAlloc2DArray(stack_d, stack_size);
cudaMmeoryAlloc(top_d, topPointer_size);

for i=1 to 2*numberOfOperand-1 do
    switch(token of postfix) {
        case operand:
                    switch(operator) {
                        case ">": call selectGreater_kernelProgram
                        case "<": call selectSmaller_kernelProgram
                        case "=": call selectEqual_kernelProgram
                                    :
                                    :
                    }
        case LogicOperator_AND:
                        call selectAND_kernelProgram
        case LogicOperator_OR:
                        call selectOR_kernelProgram
    }
```

Figure 3 - 4 Algorithm of Selection Query

```
Algorithm SelectGreater_kernelProgram (dataTable, stack, top, selected_flag, column)
Input: dataTable (the address of data table in the GPU memory)
       column(index of column)
Output: selected_flag (an array of flags denotes which record is selected)
begin
    for idx = 1 to (the size of data table) do in parallel
        top[idx]++;
        if (dataTable[column] [idx] > value) then
            stack[index][top[idx]] =1;
        else
            stack[index][top[idx]] =0;
end
```

Figure 3 - 5 Algorithm of Greater Process in Selection Query

```
Algorithm selectAND_kernelProgram (dataTable, stack, top, selected_flag)
Input: dataTable (the address of data table in the GPU memory)
Output: selected_flag (an array of flags denotes which record is selected)
begin
    for index = 1 to (the size of data table) do in parallel
        stack[index][ top[index]-1] = stack[index][ top[index]-1] &
                                      stack[index][ top[index]];
        top[index]- -;
end
```

Figure 3 - 6 Algorithm of AND Process in Selection Query

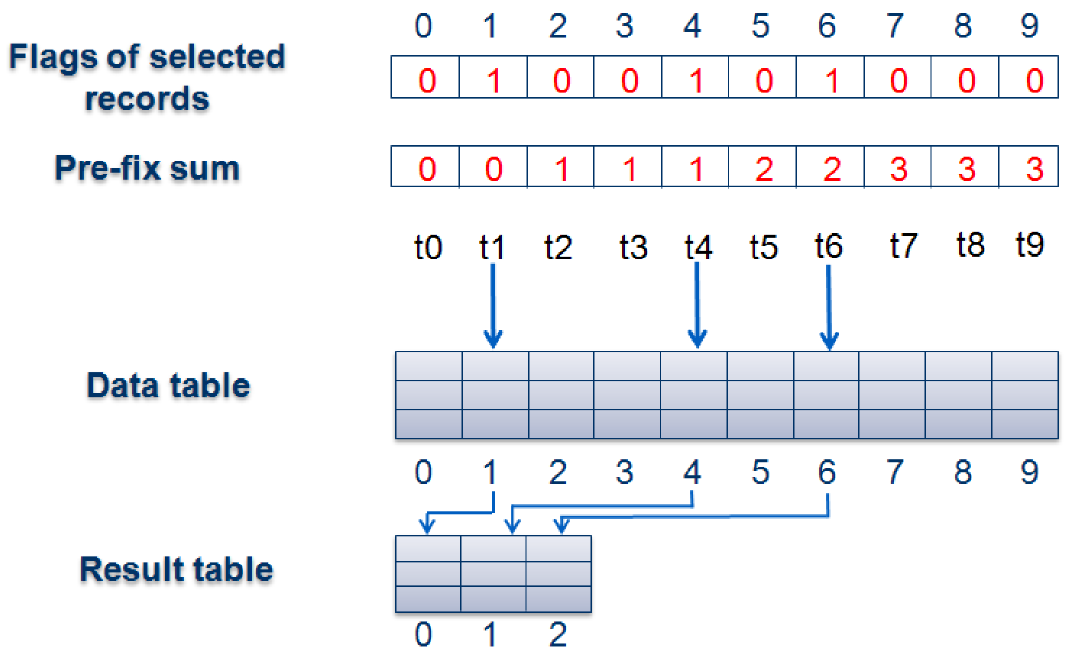

Figure 3 - 7 Process of moving data queried to the result table.

The Figure 3 - 7 shows the algorithm of moving records to result table. Assume we have an 10 records table, so we assign 10 threads to each record. In Figure 3 - 7, t0~t9 means threads with tread ID [0] ~ thread ID [9]. Each thread of thread ID **[i]** checks two values, one is in the flags of selected records and the other one is in the result array of pre-fix sum function. If the value in the flag array of selected records is "1" which means the value of corresponding address in the result array of pre-fix sum function is the new position of selected record in result table. Finally, duplicate selected records and move to the result table.

3.3.3 Sorting Data

After selection query, the entire result table can be sorted according to a key column identified by user. This entire process is commonly called sorting key-value pairs.

The index of data to denote original position is necessary for sorting key-value pairs. As we mentioned in section 3.2, we modified the **cudppSort**, a function provide by CUDPP library. The cudppSort of modified version takes two 1-D array as inputs. The outputs of modified cudppSort include one sorted array and index array of original data array. The example of index array is showed as Figure 3 - 8. The algorithm of Data Sorting we implemented is described as Figure 3 - 9, and Figure 3 - 10.

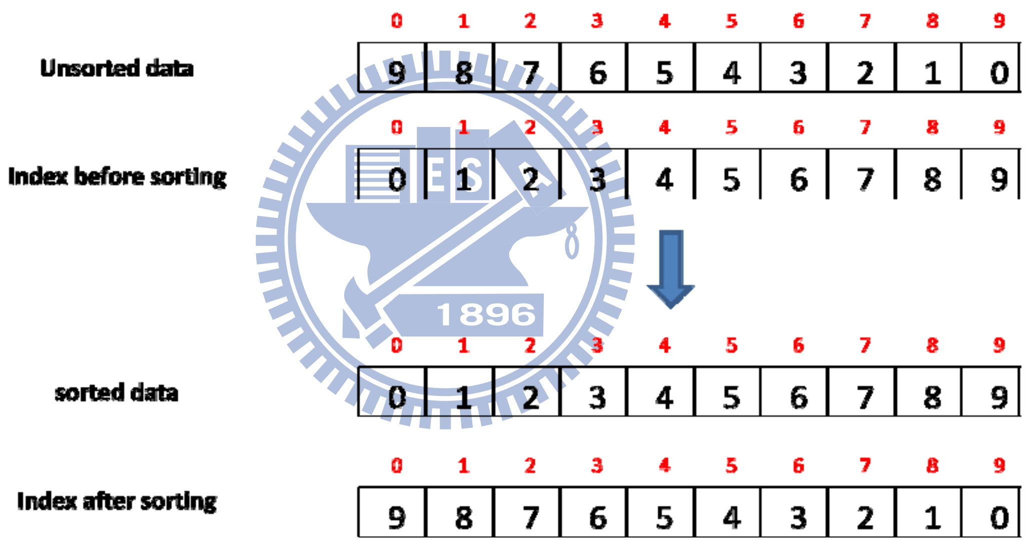

Figure 3 - 8 Modified algorithm of parallel sorting

```
**Algorithm DataSorting (dataTable, sortedTable, selected_flag, key)**
**Input:** dataTable (the result Table of Selection Query)
          key(a column number for sorting key-value pairs)
**Output:** sortedTable
**begin**
          **cudaMmeoryAlloc** (sortdata_input, number of records queried);
          **cudaMmeoryAlloc** (sortdata_output, number of records queried);
          **cudaMmeoryAlloc** (sortdata_index, number of records queried);
```

```
    copy the data of key column from dataTable to sortdata_input[]

    for index=1 to number of records queried do in parallel
        sortdata_index[index]=index;

    cudppSortModfied (sortdata_output, sortdata_index, sortdata_input, number of elements);

    mvSort_kernelProgram (sortedTable, dataTable, number of elements,
  sortdata_index );      //copy data from dataTable according to sortdata_index
```

Figure 3 - 9 Algorithm of Data Sorting

```
Algorithm mvSort_kernelProgram (sortedTable, dataTable, number of elements,
sortdata_index )
Input: dataTable (the starting address of data table in the GPU memory)
       sortdata_index (an array of index denote the original address of records.)
Output:  sortedTable (a table with sorted data according to the sortdata_index)
Begin
    for columnIdx = 0 to (the number of columns) do
        for index = 1 to (the number of records queried) do in parallel
        sortedTable[columnIdx][ index] =
                                 sortedTable[columnIdx][ sortdata_index[index]]
        stack[index][ top[index]-1] = stack[index][ top[index]-1] &
                                   stack[index][ top[index]];
end
```

Figure 3 - 10 Algorithm of moving data after sorting

As showed in Figure 3 - 8, values of address [i] in index means which data in original array was moved to address [i] after sorting. For parallelism, threads are assigned to each column of records and move them in parallel according to index array.

3.3.4 Data Grouping

The Data Grouping is used in conjunction with the aggregate functions to group the result-set by one or more columns. For example, the left of Table 3 - 2 is our data table, and the right table is our result table after Data Grouping.

Table 3 - 2 Data Table (**left**) and Result Table (**right**)

group	value
1	145
1	254
2	645

group	value
1	399
2	645

At beginning, data scattered irregularly over the data table. In order to divide data into several groups, the entire table has to be sorted according to the column "group". After sorting, the data with the same group is put together in the table. Then, we can move on next step, calculate the aggregate functions to group.

There is a function provided by CUDPP Library, cudppSegmentationScan, can help us to calculate the aggregate functions to group. Flags will be set to dedicate the start address of every group according to sorted column of group such as Figure 3 - 11.

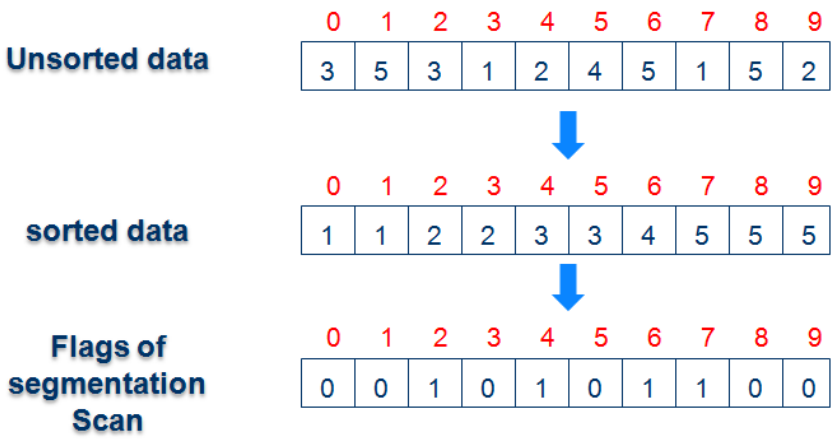

Figure 3 - 11 Setting flags of segmentation Scan

Every address [i] and address [i-1] is checked by the thread with thread ID [i]. If the values in address [i] and address [i-1] are not equal, the flag of address [i] will be set in flag array. The flag array is an input of cudappSegmentationScan. There are several algorithms of cudppSegmentationScan can perform. We implemented SUM, COUNT, MAX, MIN and AVG which included in the most of data base. We can directly implement SUM, MAX and MIN using sum, min, and max algorithms provided by cudppSegmentationScan. For COUNT, we need another array and all values are set as '1' in the array. Then, the output of cudppSegmentationScan with sum algorithm applied to this array will be the result of COUNT (Figure 3 - 12). Finally, the result of AVG can be calculated through divide SUM by COUNT. The basic concept of Data Grouping was listed in Figure 3 - 13, Figure 3 - 14

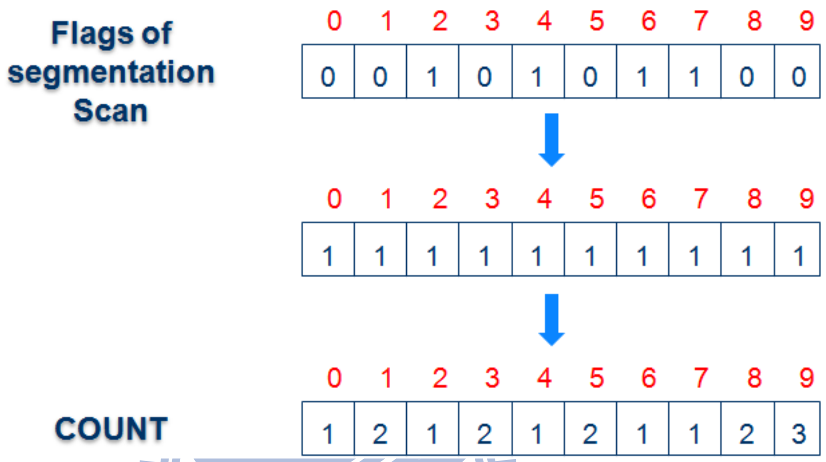

Figure 3 - 12 Process of COUNT function

```
Algorithm DataGroup (resultOfSelect, QueryData, number of data queried)
Input: resultOfSelect (a data table of Selection Query result)
Output: sortedTable
begin
        cudaMmeoryAlloc(sort_idata, number of data queried);
        cudaMmeoryAlloc(sort_odata, number of data queried);
        cudaMmeoryAlloc(sort_index, number of data queried);
        cudaMmeoryAlloc(grp_idata, number of data queried);
        cudaMmeoryAlloc(grp_odata, number of data queried);
        cudaMmeoryAlloc(flag_d, number of data queried+1);
        cudaMmeoryAlloc(grp_icnt, number of data queried);
        cudaMmeoryAlloc(grp_ocnt, number of data queried);

        for index=1 to number of records queried do in parallel
            sort_index [index]=index;
```

```
     copy the data of group from data Table to sort_idata;
     cudppSortModfied(sort_odata, sort_index, sort_idata, number of elements);

setGrpInput_kernelProgram(grp_idata, flag_d, sort_odata, resultOfSelect)
          // set the flag for segmentation scan and data of the aggregate functions

          switch(QueryData ->funcOp){
                              case DB_SUM:
                                        segmentationScan.op = CUDPP_ADD;
                                        break;
                              case DB_AVG:
                                        segmentationScan.op = CUDPP_ADD;
                                        break;
                              case DB_MIN:
                                        segmentationScan.op = CUDPP_MIN;
                                        break;
                              case DB_MAX:
                                        segmentationScan.op = CUDPP_MAX;
                                        break;
                              default:
                                        ;
                         }

          cudppSegmentedScan(grp_odata, grp_idata, flag_d, number of elements);

          if (QueryData->funcOp==DB_COUNT|| QueryData->funcOp==DB_AVG){

          for index=1 to number of records queried do in parallel
                         grp_icnt [index]=1;

          segmentationScan.op = CUDPP_ADD;
          cudppSegmentedScan(grp_ocnt, grp_icnt, flag_d, number of elements);

          switch(QueryData ->funcOp){
                              case DB_COUNT:
                                        mvGrp_kernelProgram;
                                        break;
```

```
                case DB_AVG:
                        mvGrpAVG_kernelProgram;
                        break;
                default:
                        mvGrp_kernelProgram;
                        break;

        return the result Table of DataGroup
end
```

Figure 3 - 13 Algorithm of Data Group

```
Algorithm mvGrpAVG_kernelProgram (resultTable, sort_odata, grp_odata,
           grp_ocnt, flag, number of data queried)
Input: sort_odata (group data)
      grp_odata (result of the aggregate function, SUM)
      grp_ocnt (number of elements in each group)
      flag (the fist position of data in each group)
Output: resultTable (result of Data Group)
begin

        for idx = 1 to idx=number of data queried do in parallel
            if(flag[idx]=1){
                resultTable[column0][resultIndex] = sort_odata[idx-1]
            resultTable[column1][resultIndex] = grp_odata[addr-1]/grp_ocnt[addr-1];
            }

        if (threaded=0) {
        resultTable[column0][number of data queried]=sort_odata[number of data
                                                   queried -1];
        resultTable[column1][number of data queried] = grp_odata[number of data
          queried -1] / grp_ocnt[number of data queried -1];
        }
end
```

Figure 3 - 14 Algorithm of AVG process in Data Group

3.3.5 Data Update

The idea of data update is trivial. There is a flag array processed by Selection Query denote which record is selected. Then, update date table according to the flag array. Each records marked in the flag array is updated in parallel. One thread updates one record; the speed up advance is limited by speed of memory write.

3.3.6 Data Delete

2-D array is our data structure of data table. As some data are deleted, the other data in the table has to be moved for alignment. In opposition to array, deleting data in link list is easier without moving other data but modify the pointers. However, the data which are not deleted will be moved to a new 2-D array allocated as new data table. The pointer in host of original data table will be replaced by new data table, then free the space of the original table. Like Data Update, the flag array is used to specify which data is selected. These selected data should be deleted. Because the non-deleted data should be moved from original table to new table, the flag are reversed to denote the non-deleted data. Finally, we move the non-deleted data from original table to result table. Then the original table was freed and replaced by the result table.

3.3.7 Data Insert and Data Insert by Selection Query

The idea of inserting data is easy to understand. New data is transmitted to GPU memory and inserted to the position next the last records of data table. Unfortunately, we have to face the problem brought by data structure. Because data table is a 2-D array, total number of data which the table can accommodate is fixed. A new table should be issued when the size of data exceeds the size of table after inserting data. The size of new table will be twice as big as the old table. For example, a table with table size 1024 means it can accommodate 1024 records. When the 1025th record was inserted to the table, the new table with table size 2048 will be issued.

The concept of data insert by selection is similar to data insert. According to the flag array of selection query, insert selected data to the destination table. If the size of destination table is not enough for new data, a new table will be issued with twice table size, and all data will be moved to this new table.

Chapter 4 Experimental Results and Analysis

We try to evaluate the performance of our GPU data base and compare it with naïve CPU approach to those functions listed in chapter 3 with variable total number of data in data table and number of data in result table.

4.1 Experimental Setup

We adopt CPU of 4 cores and GeForce 9800 GT for our computation platform. The configuration information is described as following.

4.1.1 A Memory Data Base - SQLite

SQLite is an embedded relational database management system contained in a C programming library. The word "embedded" means that SQLite engine is not a standalone process with which the program communicates. Instead, the SQLite library is linked in and thus becomes an integral part of the program. The belief that SQLite is the most widely deployed SQL database engine stems from its use as an embedded database. There many applications used SQLite as their data base including Mozilla Firefox, Mac computer, iPhones, and millions of websites etc. There two modes, disk mode and memory mode, SQLite can switch. In memory mode, SQLite exist in memory purely as a main memory data base. In this experiment, we compared the performance of our GPU DB with corresponding functions of SQLite on memory mode.

The index structure of SQLite is B-tree. The time complexity of all operations in B-tree is O (log n). The execution time is sensitive to the number of records queried, because more data is selected after selection query means more nodes should be traversed in B-tree.

4.1.2 Hardware Configuration

To construct the environment for our experiment and support CUDA computing, we have the following hardware configuration.

Table 4 - 1 Hardware configuration

CPU	Intel Core 2 Quad Q6600 (2.4GHz, four core)
Motherboard	ASUS P5E-VM-DO-BP, Intel® X38 Chipset
RAM	Transcand DDR-800 2G
GPU	NVIDIA 9800 GT 512MB (GIGABYTE OEM)
HDD	WD 250G w/ 8MB buffer

Since we implement our data base on GPU memory, we list the specification of the GPU in detail as following.

Table 4 - 2 NVIDIA 9800GT Hardware Specification

Core Name	GeForce 9800 GT (G92)
Number of Multi-Processor	16
Number of Registers	8192 (per SIMD processor)
Constant Cache	8KB (per SIMD processor)
Texture Cache	8KB (per SIMD processor)
Processor Clock Frequency	Shader: 1.751 GHz, Core: 700 MHz
Memory Clock Frequency	900 MHz
Shared Memory Size	16KB (per SIMD processor)
Device Memory Size	512MB GDDR3

There are 8 SP (stream processor) included in each MP (multi-processor). Each SP can process one single precision floating pointer calculation. Totally, the GPU in ideal situation can processes 8x16=128 single precision floating pointer calculations simultaneously.

4.1.3 Software Configuration

Table 4 - 3 Software Configuration

OS	Open SUSE with version 11.1 (32bit version)
GPU Driver Version	185.18.14
CUDA Version	2.2
GNU Compiler	gcc41

4.2 Evaluation and Analysis

We start from normal table size with 50,000 records. Insert 50,000 records to the data table and compare the insertion performance of GPU DB with SQLite memory DB. Because several comparison results of functions are similar, we choose results of Selection, Data Grouping, and Insert Data by Selection Query to represent other similar comparison results. Then we will discuss the relationship of GPU execution time between total number of records and number of records in query result. Finally, we make a figure to point out how many number of records in query result will be the turning point of GPU DB and SQLite memory DB with variable total number of records in data table.

4.2.1 The Pattern of Test Data

The bench mark on SQLite web site is referred for our test data. There are tree column in our data table. The data in first column of each record is a unique random number between 1 to total number of records, so we can control how many records we queried. The second column is the group number which divided all records to 100 groups. The last column of our test data is a random number from 1 to 65535. The following is an example of test data which is belong to group 50.

Column 1	Column 2	Column 3
2	50	598

4.2.2 Performance Evaluation of Functions

In section 4.2.2, we will discuss each performance comparisons of Data Insert, Data Grouping, and Data Insert According to Selection Query in detail.

4.2.2.1 Insertion

From Figure 4 - 1, we can see the insertion function of GPU DB suffered from the overhead pass data from host memory to GPU memory, although the difference between execution times of two systems is small about from 20 ms to 120 ms. Average execution time increasing of GPU DB is 65.512 ms for each additional 500 records. Because, we expand table size 2 times for each time 2-D array is full. The expanding overhead takes another 10.453 ms in average.

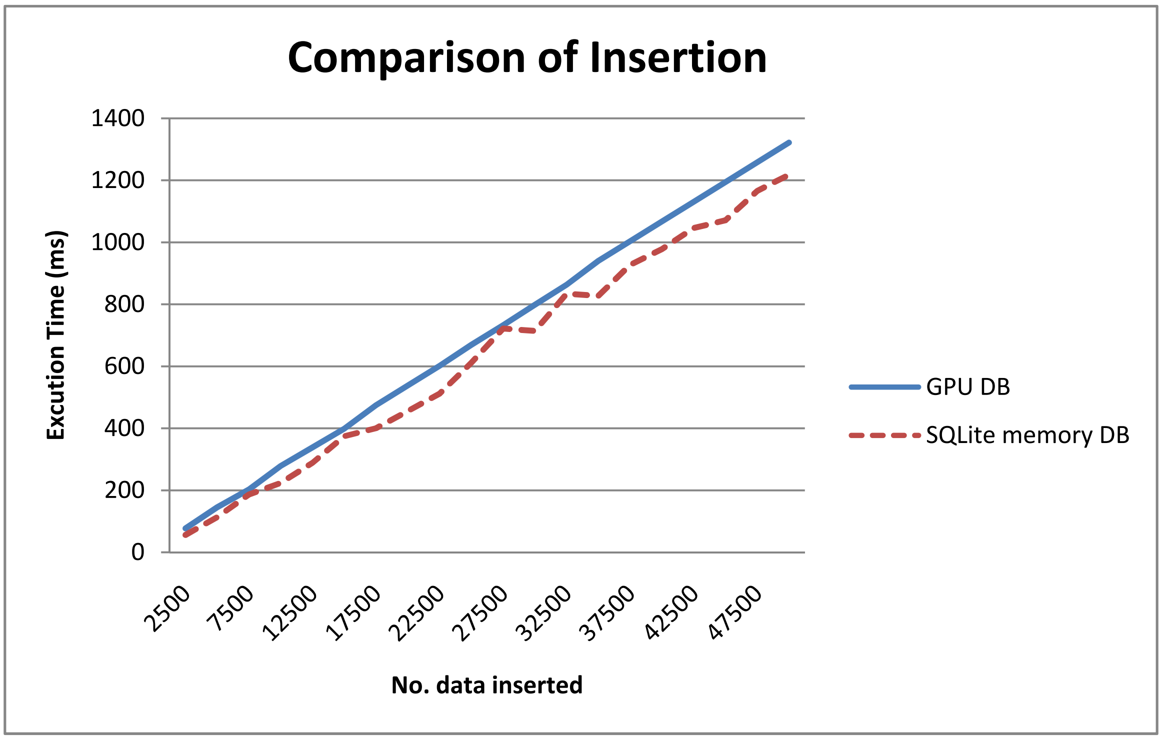

Figure 4 - 1 Execution Time Comparison of Insertion

4.2.2.2 Selection Query

As showed in Figure 4 - 2, GPU DB takes 2.598 ms in average to complete Selection Query process.The line in of GPU DB in Figure 4 - 2 is almost unchanged and unrelated to number of data queried. The performance of SQLite memory DB is better than our GPU DB before the crossing point of two lines. The number of data queried of the crossing point is about 1,700 records. We also evaluate the execution time of number of records queried from 2,500 to 10,000 records, and the execution time of GPU DB always less than SQLite along with the increasing number of records queried.

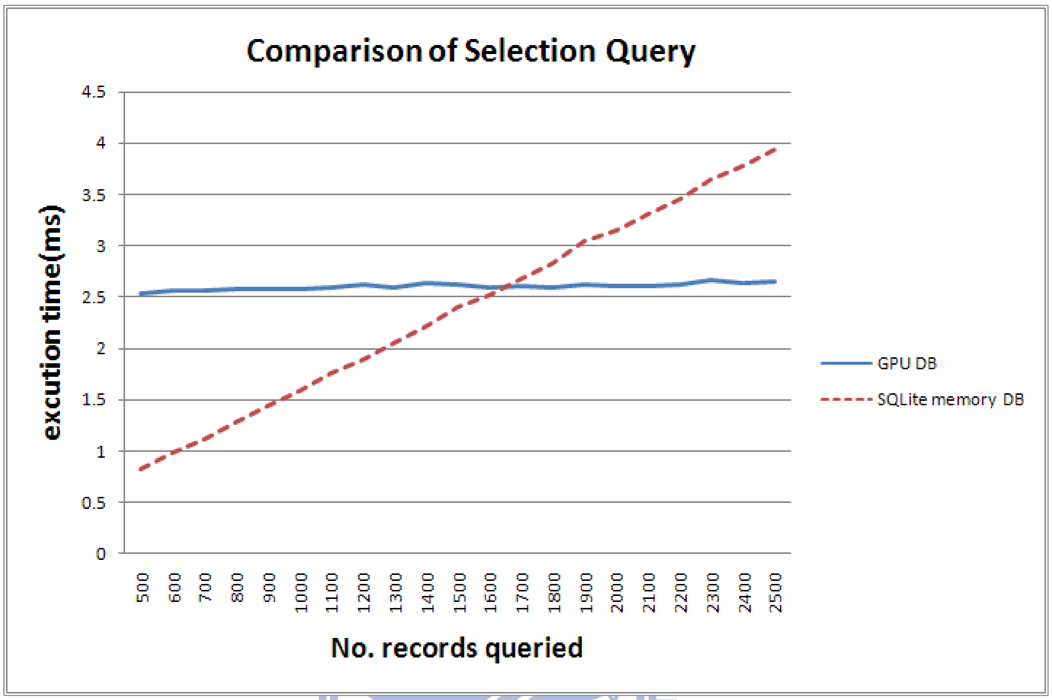

Figure 4 - 2 Execution Time Comparison of Selection

4.2.2.3 Data Grouping

In Figure 4 - 3, the execution time of GPU DB is increasing slowly caused by increase of number of records queried. One thing can be believed, more number of records queried, more records should be sorted before calculating the aggregate functions. Of course, the number of records queried also affects the execution time of calculation the aggregate functions. We will go deep into how much GPU time this process takes in the section 4.3.

Similar to evaluation of Selection Query, there is a crossing point of the lines of GPU DB and SQLite memory DB. According to evaluation of execution time from number of records queried 2,500 records to 10,000 records, the performance of our GPU DB will be always better than SQLite.

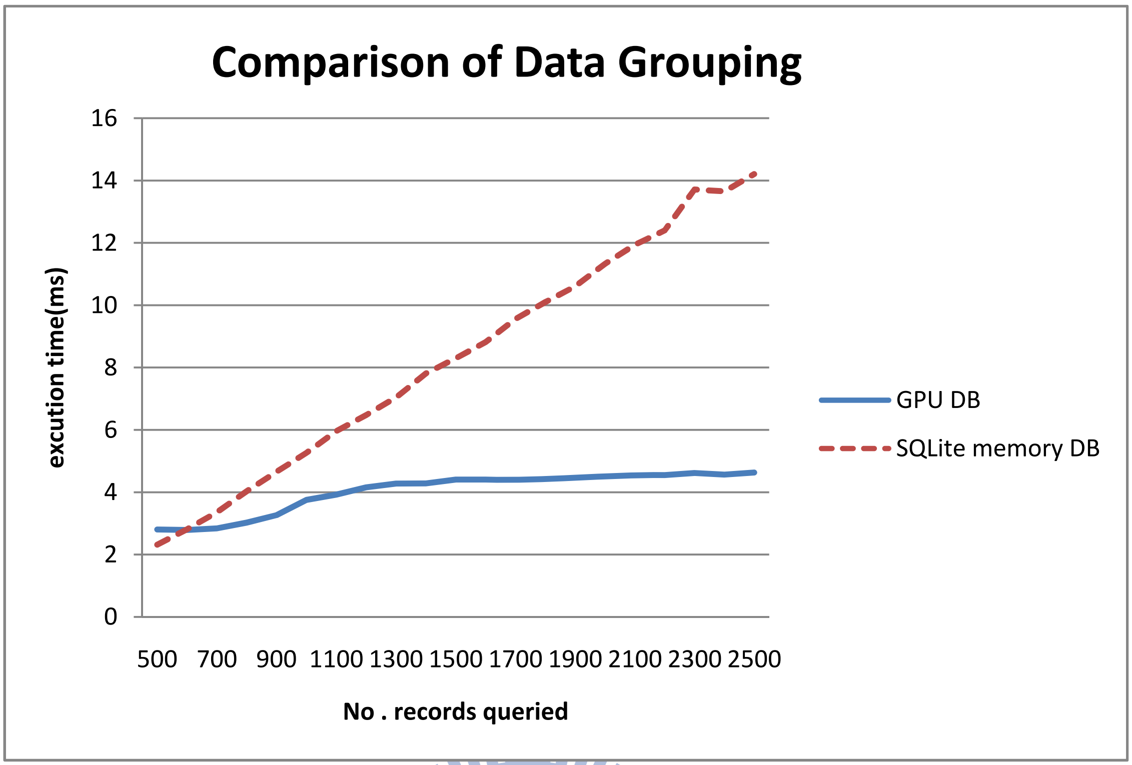

Figure 4 - 3 Execution Time Comparison of Data Grouping

4.2.2.4 Data Insert According to Selection Query

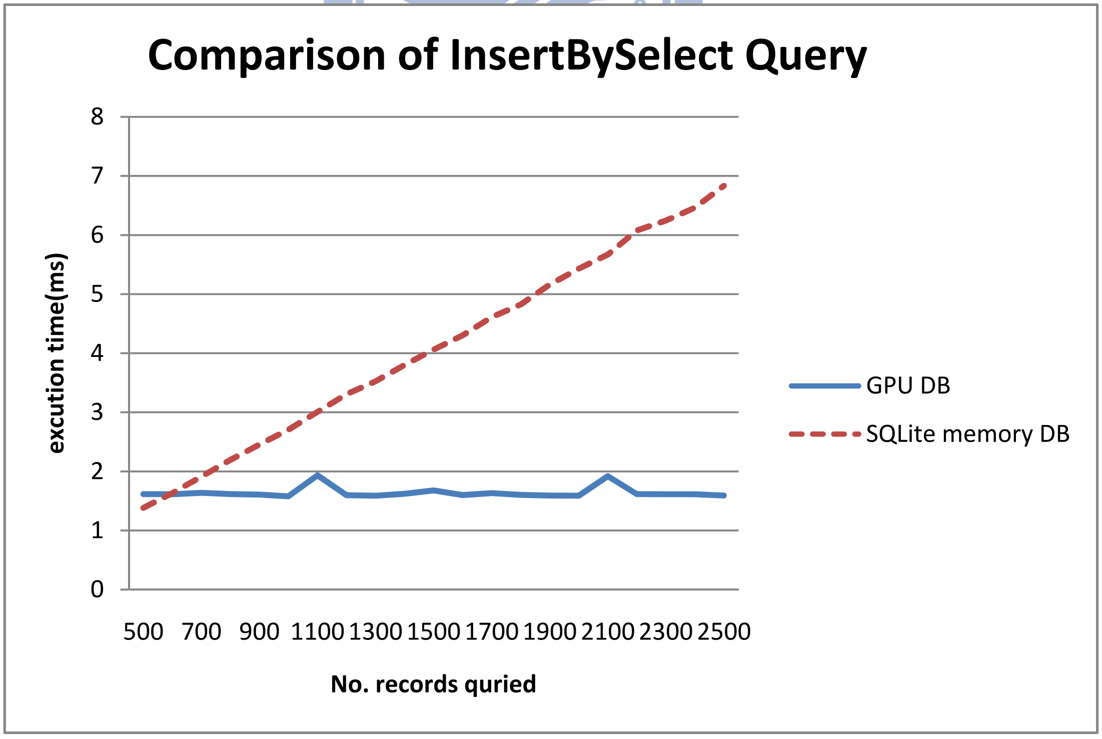

Figure 4 - 4 Execution Time Comparison of Data Insert According to Selection Query

Similar to the evaluation result of Selection Query and Data Grouping, there is also a crossing point of the lines of GPU DB and SQLitse memory DB, and the performance of GPU DB is always better than SQLite memory DB after that crossing point.

There two jumping points at the line of GPU DB is 1,100 records queried and 2,100 records queried. Because the total data size is over the table size after inserting data, the table was expanded 2 times table size to accommodate all records.

4.2.3 Analysis of GPU Computation Time of Data Grouping

Because there are two steps included by Data Grouping process, one is Selection Query, and the other one is Sorting Data and aggregate functions calculation, we evaluated the computation time of kernel programs which run on GPU to observe its behavior.

At first, we evaluated the variation of GPU execution time followed the increase of number of records queried from 2,500 records to 1,000 records and total number of data is 100,000 records in the data table. As we see in Figure 4 - 5, more records queried more GPU computation time Data Grouping step takes.

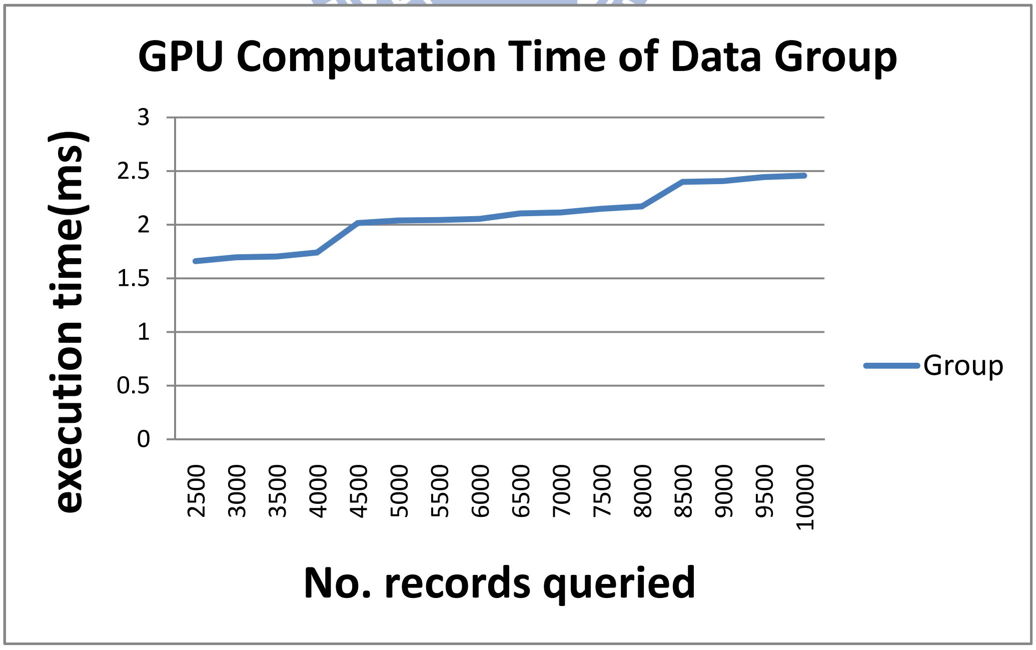

Figure 4 - 5 GPU Execution Time of Data Group

There are two jumping points at 4,500 records queried 8,500 records queried. Pre-fix sum algorithm and sorting algorithm implemented by CUDPP increase 2 in the power of n threads each time while the number of threads is not enough. Because number of records 4,500 is greater than 4096, the program issued more threads but take more time. The detail implementation of pre-fix sum and sorting algorithms will not be discussed here. Please refer to [4] and [13].

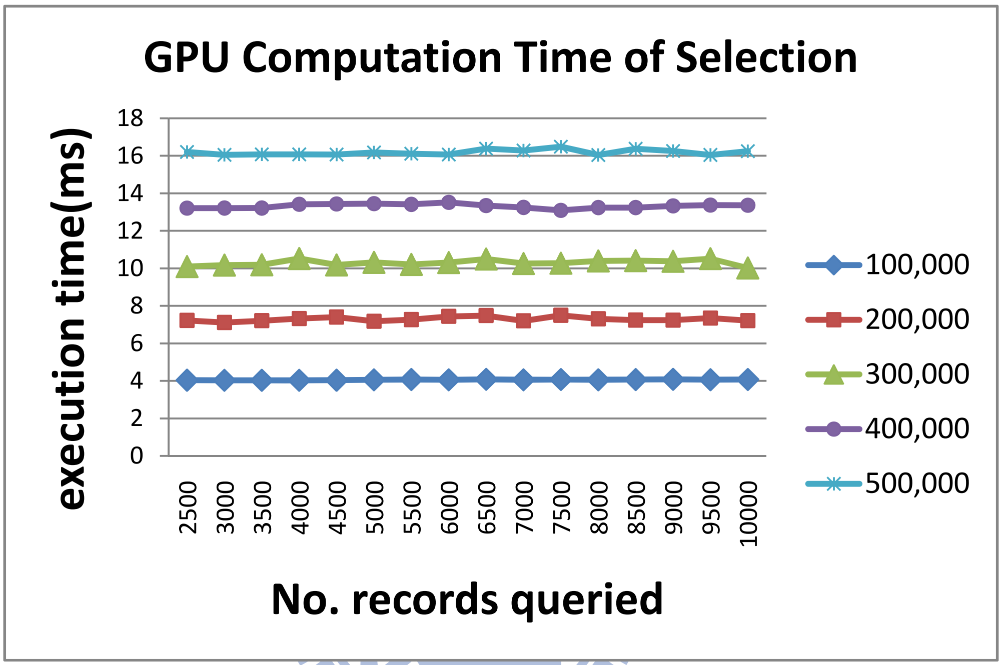

Figure 4 - 6 GPU Computation Time of Selection Query

We evaluated the GPU computation time of Selection Query in Data Grouping process separately with total number of 100K, 200K, 300K, 400K, and 500K records in data table. As showed in Figure 4 - 6, GPU computation time of Selection Query is not sensitive to number of records queried. According to our implemented methods, one thread is designate on one record for execution selection query process. Because the limitation of hardware, compiler will issue the maximum number of threads that the hardware can sustain by evaluating registers or shared memory usage of kernel program.

When all turns of each thread are complete, the selected data will be marked in flag array. Selection Query takes stable execution time for the same total number of records in data table and is not related to how much number of records queried. The execution time of Selection Query is related to total number of records in data table because more data records in table, more turns threads executes, and of course, it takes more time to complete the whole process.

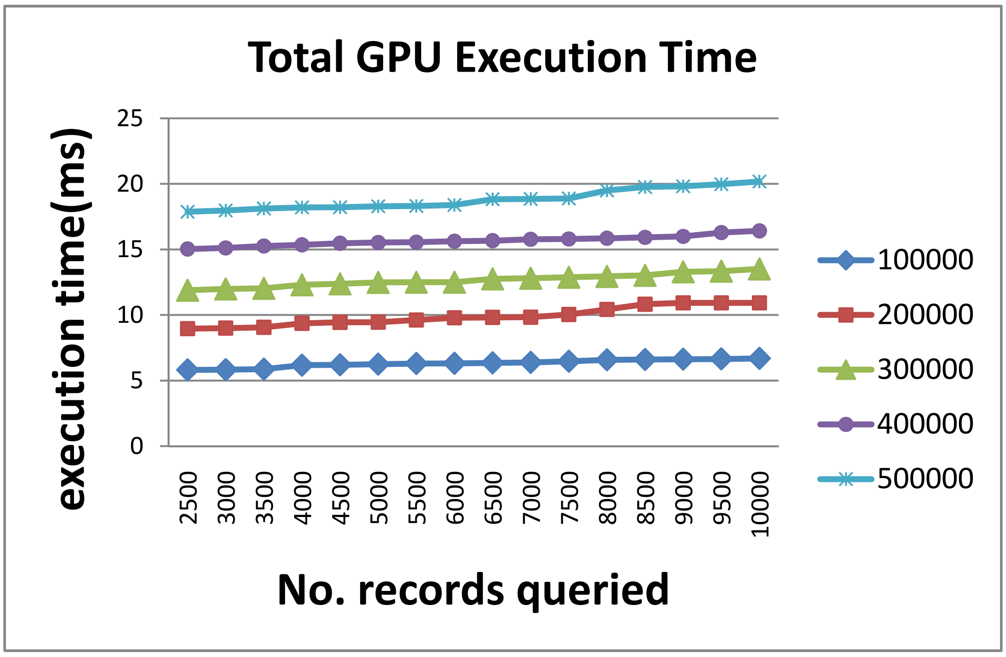

Figure 4 - 7 Total GPU Computation time of Data Grouping

Finally, we evaluated the total GPU time of Data Grouping. The proportion of number of records queried to total number of records in data table is small, so the increase of total GPU execution time followed increase of number of records queried is related small.

4.3 Evaluation of Turning points

We evaluated the turning points of GPU DB and SQLite memory DB. Each turning point was the crossing point of the performance evaluation of GPU DB and SQLite memory DB. The performance of each function of our GPU DB is always better than SQLite memory DB since increase of number of records queried is more than the turning point has dedicated.

The height of lines in Figure 4 - 8 implicates the difference of performance between our GPU DB and SQLite memory DB. For example, the lines of Selection and Sorting are higher than Grouping which implicates the difference of performance between our GPU DB and SQLite memory DB in Data Grouping is smaller than Selection Query and Sorting Data.

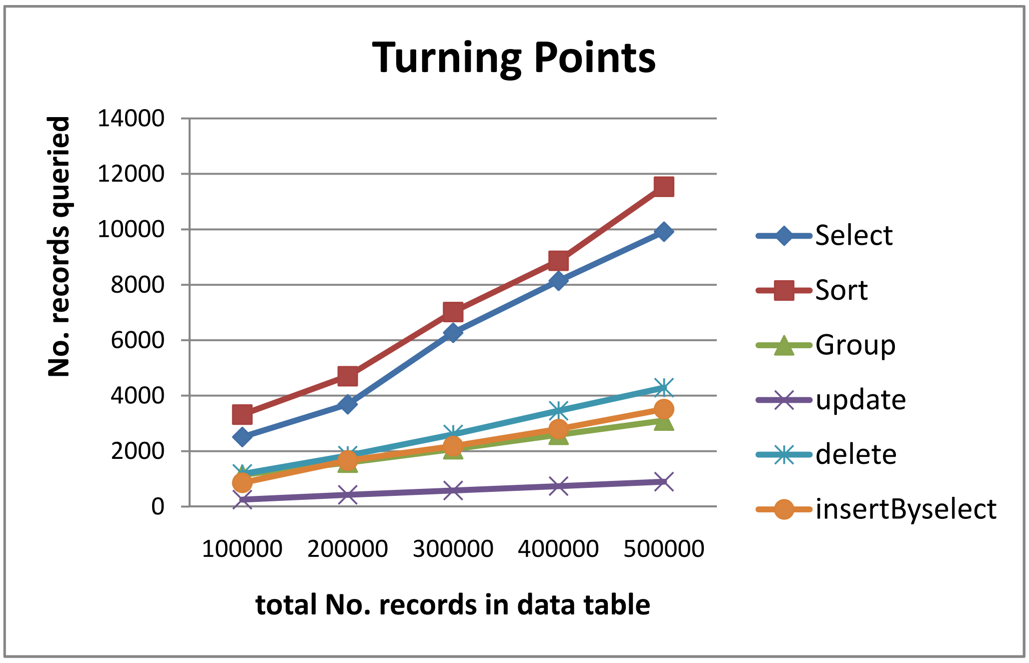

Figure 4 - 8 Turning points of GPU DB and SQLite memory DB

From Figure 4 - 8, we figured out Table 4 - 4 the ratios of each function which showed when about 0.161% to 2.061% of number of total records is queried, the performance of our GPU DB will be better than SQLite memory DB. The ratios listed in Table 4 - 4 the ratios of each function are small, and we believe that it is easily to be exceeded in common case.

Function Name	**Ratio of No. data queried to total No. records (%)**
Selection Query	1.926%
Selection Query and Sorting Data	2.061%
Selection Query and Data Grouping operations	0.491%
Data Insert According to Selection Query	0.784%
Data Update	0.161%
Data Delete	0.784%

Table 4 - 4 the ratios of each function

Chapter 5 Conclusions and Future works

5.1 Conclusions

Using GPU for problems with High density computation normally brings remarkable improving of performance. Of course, these problems should be able to be parallelized. The advance development of GPGPU has already speeded up many applications which are used to be computed in host CPU. In this paper, we survey the background of existing main memory data base and CUDA programming model. We proposed an entire new architecture for experimental data base. Major computation bound of our data base operations are data sorting, prefix-sum process and the aggregate functions calculation. The proportion of these operations to memory I/O bound operations such as selection query is small.

The experiment result shows the operations of our GPU DB takes almost the same execution time for the same total number of records. The execution time of our data operations are not sensitive to how many number of records queried. The result of experiment is different between our GPU DB and SQLite. After query operations, more records in the result table more execution time SQLite take. Oppositely, our GPU DB is not sensitive to how many number of records in result table but sensitive to how many number of total records in data table. Based on this, we evaluated the turning point between our GPU DB and SQLite memory DB. The change of turning points trended to linear variation and we figured out the approximate ratio of records queried to total number of records. Finally, the ratio is small, about 0.161% to 2.061% according to different functions, and we believe that it is easily to be exceeded in common case.

5.2 Future Works

Although we have seen the capability of our GPU DB, there are many issues considered for improving our data base.

(1) Parallel string data query

Because there is no appropriate solution for parallel string data query, our GPU DB is designed for only numeric data now. One 2-D array stores data in the same data type. String data has to be stored as numeric type or in other 2-D array different from the numeric data stored. Each word has its own alphabet order, so the comparison on words supposed to begin from the prefix to suffix. To parallel computation, the characteristic on sequence of string seem to be undefeated. However, the string comparison is necessary to complete implementation of data base.

(2) Join Query

An SQL join query combines fields from two tables by using value common to each. Our GPU DB does not support the join query now. In the future, we will design a relation table to manage the relationship between each table.

(3) Concurrency data base query

Because our GPU DB can process only one global function call at the same time, we plan to design a scheduler. This scheduler can combines multiple requests to one GPU function call and maintains the data consistency for concurrency data base query in a multiuser database environment.

Bibliography

[1] Tobin J. Lehman and Michael J. Carey, "A Study of Index Structures for Main Memory Database Management Systems". VLDB, Kyoto, Japan, 1986.

[2] Ali Bakhoda, George L. Yuan, Wilson W. L. Fung, Henry Wong and Tor M. Aamodt, "Analyzing CUDA workloads using a detailed GPU simulator", Performance Analysis of Systems and Software-ISPASS, Boston, Massachusetts, 2009.

[3] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens, "Scan Primitives for GPU Computing", ACM, 2007.

[4] Mark Harris, "Parallel Prefix Sum(Scan) with CUDA", NVIDIA, 2008

[5] Garland M, Le Grand S, Nickolls J, Anderson J, Hardwick J, Morton S, Phillips E, Yao Zhang, Volkov V, "Parallel Computing Experiences with CUDA". IEEE, Los Alamitos, CA, USA, 2008.

[6] Qihang Huang, Zhiyi Huang, Paul Werstein, and Martin Purvis, "GPU as a General Purpose Computing Resource". IEEE , Washington, DC, USA, 2008.

[7] Ziyi Liu, Wenjing Ma, "Exploiting Computing Power on Graphics Processing Unit" IEEE, Washington, DC, USA, 2008.

[8] David Luebke, "CUDA: Scalable Parallel Programming for High-Performance Scientific Computing", NVIDIA 2008.

[9] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA Tesla: A Unified Graphics and Computing Architecture", IEEE, 2008.

[10] "NVIDIA CUDA Programming Guide, 2.0 edition", NVIDIA Corporation 2008.

[11] "NVIDIA CUDA Programming Guide, 2.2 edition", NVIDIA Corporation 2009.

[12] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Mike, and H. Pat, "Brook for gpus: Stream computing on graphics hardware", ACM, New York, USA, 2004.

[13] Shin-Jae Lee, Minsoo Jeon, Andrew Sohn, and Dongseung Kim, “Partitioned Parallel Radix Sort”, ISHPC, Tokyo, Japan, 2000.

[14] Pablo Barcel´o, “Logical Foundations of Relational Data Exchange,ACM,SIGMOD, Mrch”, New York, US, 2009.

[15] T. Purcell, I. Buck, W. Mark, and P. Hanrahan, “Ray tracing on programmable graphics hardware. ACM Trans on Graphics”, SIGGRAPH, San Antonio, Texas USA, 2002.

[16] T. Purcell, C. Donner, M. Cammarano, H. Jensen, and P. Hanrahan, “Photon mapping on programmable graphics hardware”, SIGGRAPH/Eurographics Conference on Graphics Hardware, San Diego, California, USA, 2003.

[17] E. S. Larsen and D. K. McAllister. “Fast matrix multiplies using graphics hardware”, IEEE Supercomputing, 2001.

[18] D. Manocha, “Interactive Geometric and Scientific Computations using Graphics Hardware”, SIGGRAPH, 2003.

[19] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis. “Weaving relations for cache performance”, In Proceedings of the Twenty-seventh International Conference on Very Large Data Bases, 2001.

[20] S. Manegold, P. Boncz, and M L. Kersten, “What happens during a join? Dissecting CPU and memory optimization effects”, VLDB, Proceedings of 26th International Conference on Very Large Data Bases, Cairo, Egypt, , September 10-14, 2000.

[21] Shintaro Meki and Yahiko Kambayashi. “Acceleration of relational database operations on vector processors”. Systems and Computers, Japan, August 2000.

[22] Jun Rao and Kenneth A. Ross. “Cache conscious indexing for decision-support in main memory”, VLDB, 1999.

[23] Kenneth A. Ross. “Conjunctive selection conditions in main memory”, Proceedings of the Twenty-First ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems: PODS, 2002.

[24] M. Macedonia. , “The gpu enters computing's mainstream”. IEEE, October 2003.

[25] Honghoon Jang; Anjin Park; Keechul Jung, “Neural Network Implementation Using CUDA and OpenMP”, Computing: Techniques and Applications, 2008.

[26] Guobin Shen, Lihua Zhu, Shipeng Li, Heung-Yeung Shum, Ya-Qin Zhang, “Accelerating video decoding using GPU”**,** IEEE International Conference,2003.

[27] Kenneth Moreland, Edward Angel, “The FFT on a GPU”, Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware, San Diego, California, 2003.

[28] Simek, V, Asn, R.R, “GPU Acceleration of 2D-DWT Image Compression in MATLAB with CUDA”**,** Computer Modeling and Simulation, 2008.

[29] Wei-Nien Chen; Hsueh-Ming Hang, “H.264/AVC motion estimation implmentation on Compute Unified Device Architecture (CUDA)”, IEEE International Conference 2008.

[30] Manavski, S.A, “CUDA Compatible GPU as an Efficient Hardware Accelerator for AES Cryptograph”**,** ICSPC 2007.

[31] Naga K. Govindaraju , Brandon Lloyd , Wei Wang , Ming Lin , Dinesh Manocha, “Fast computation of database operations using graphics processors”, ACM SIGMOD international conference on Management of data, Paris, France, June 13-18, 2004.

[32] Jingren Zhou, Kenneth A. Ross, “Implementing Database Operations Using SIMD Instructions”, ACM SIGMOD, Madison, Wisconsin, USA, 2001.