Inf Syst Front (2012) 14:909–924
DOI 10.1007/s10796-011-9322-0

Scaling database performance on GPUs

Yue-Shan Chang · Ruey-Kai Sheu · Shyan-Ming Yuan · Jyn-Jie Hsu

Published online: 19 August 2011

Abstract The market leaders of Cloud Computing try to leverage the parallel-processing capability of GPUs to provide more economic services than traditions. As the cornerstone of enterprise applications, database systems are of the highest priority to be improved for the performance and design complexity reduction. It is the purpose of this paper to design an in-memory database, called CUDADB, to scale up the performance of the database system on GPU with CUDA. The details of implementation and algorithms are presented, and the experiences of GPU-enabled CUDA database operations are also shared in this paper. For performance evaluation purposes, SQLite is used as the comparison target. From the experimental results, CUDADB performs better than SQLite for most test cases. And, surprisingly, the CUDADB performance is independent from the number of data records in a query result set. The CUDADB performance is a static proportion of the total number of data records in the target table. Finally, this paper comes out a concept of turning point that represents the difference ratio between CUDADB and SQLite.

Keywords GPU · CUDA · SQLite · In-Memory Database

Y.-S. Chang
Department of Computer Science and Information Engineering,
National Taipei University,
Taipei, Taiwan
e-mail: ysc@mail.ntpu.edu.tw

R.-K. Sheu (✉)
Department of Computer Science, Tunghai University,
Taichung, Taiwan
e-mail: rickysheu@thu.edu.tw

S.-M. Yuan
Department of Computer Science and Information Engineering,
Providence University,
Taichung, Taiwan
e-mail: smyuan@gmail.com

J.-J. Hsu
Department of Computer Science,
National Chiao Tung University,
Hsinchu, Taiwan
e-mail: cchsu77@gmail.com

1 Introduction

Cloud computing is currently the hottest information processing technology and it enables location-independent computing, whereby shared servers equipped with resources, software, and data to computers and other devices on demand. Market leaders of this business, including Amazon Elastic Cloud Computing, Rackspace, or Microsoft Azure, originally provides headless on-demand computing through the provision of virtual servers (Amazon Elastic Compute Cloud http://aws.amazon.com/ec2/, Rackspace Hosting http://www.rackspace.com/index.php, Micsoft Azure, http://www.microsoft.com/windowsazure/windowsazure/. If users need a Web site or database system, they can get one by contacting the cloud computing provider, and run the service on in minutes. Although cloud computing provides more economic solutions than transitions, it is not a panacea for all kind of application requirements. For example, bandwidth constraints can make it impractical to move the vast input and out data sets used in high-performance computing over the internet. Service providers also featured little about how to serve computationally intensive applications in parallel without additional capital expense and administrative complexity. These limitations drive the trends of general purpose programming on GPU techniques such as CUDA (an acronym for Computer Unified Device Architecture).

A graphics processing unit (GPU) is a specialized microprocessor that offloads and accelerates graphics rendering from

Table 1 Functions supported by CUDADB

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_InformationSET Sales=500 WHERE store_name="Los Angeles" AND Date="Jan-08-1999"
Data Delete	DELETE FROM Store_Information WHERE store_name="Los Angeles"

the central processor. Modern GPUs are very efficient at manipulating computer graphics, and their highly parallel structure makes them more effective than general-purpose CPUs (Wikipedia, http://en.wikipedia.org/wiki/Graphics_processing_unit). General-purpose processing on the GPU, known as GPGPU is currently an active research area since GPUs are widely available and continue to improve in performance faster than CPUs. CUDA is a parallel computing architecture developed by NVIDIA (nVIDIA, http://www.nvidia.com/content/global/global.php; Lindholm et al. 2008; Wynters 2011). CUDA plays the role of computing engine in NVIDIA GPU. Software developers can access it through variants of programming languages (Wikipedia, http://en.wikipedia.org/wiki/CUDA; Qihang et al. 2008; Ziyi et al. 1062; Manavski et al. 2007). There is a growing interest of employing NVIDIA's CUDA framework to solve certain problems or enhance system performance based on its parallel data processing capability (Nickolls et al. 2008; Rodrigues et al. 2008; Schatz et al. 2007; Zhang et al. 2011; Chengen and Xu) .Yutaka Akiyama leads projects of large-scale bioinformatics applications on multi-node on GPU environment (Akiyma). They has built many efficient DNA sequencing algorithms and system based on CUDA framework. Yuan et al. (2010) employed CUDA to improve the simulation performance for surgical tissue deformation. Based on the successful stories of prior works, it comes out the idea of this paper to implement an in-memory database system on CUDA-enabled GPU environment to simplify the development and enhance the performance of data intensive applications.

Databases are the workhorses of enterprises today. Searching useful information from databases is a computational challenge from day to day. Database vendors, such as Microsoft, Oracle, IBM, and SAP are seeking CUDA-enabled GPUs for scalable solutions. Before getting simple and feasible solutions, researchers and database vendors tried to reduce the solution complexity by porting in-memory database on GPUs with CUDA. In contrast to database system which employed a disk storage mechanism, an in-memory database (IMDB; also main memory database or MMDB) is a database system primarily relies on main memory or computer data storage. In recent years, there are already several commercial IMDB products on the market, such as Oracle's TimesTen, IBM's Solid DB, SAP's In-Memory Computing Engine, and Sybase's Adaptive Server Enterprise. IMDB stores data on volatile memory devices, and thus lacks of durability portion of ACID (atomicity, consistency, isolation, durability) properties (Wikipedia, http://en.wikipedia.org/wiki/In-memory_database). Most of those IMDB products on the market try to implement the whole relational database management system into the main memory, and hope to

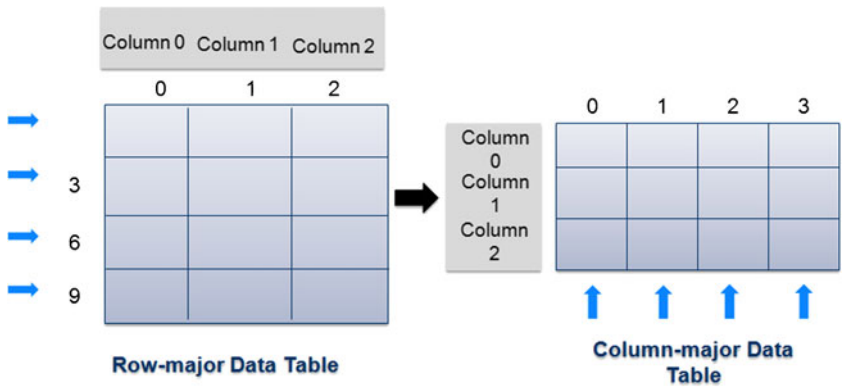

Fig. 1 Row-major GPU Memory v.s. Column-major data structure

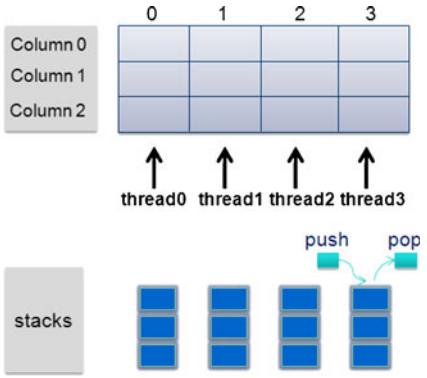

Fig. 2 Process of selection query

bring the benefits of using GPUs to consumers. It increases much complexity to develop an IMDB with full set of RDBMS functions. In addition, due to the competition between vendors, researchers cannot get the technical information or experiences about how to implement IMDB or improve the IMDB performance. Thus, it would be a good starting point and easier to implement an IMDB than the entire relational database on GPU. This also naturally escapes from the performance bottleneck of data transferred between CPU and GPU.

Recently, increased attention has been given on redesigning traditional database algorithms for fully utilizing the available architectural features and for exploiting

Table 2 Algorithm of selection query

```
Algorithm SelectionQuery_SetSelectFlag(QueryTable, QueryData, selected_flag )
Input: QueryTable // the data set to be queried
       QueryData //data selection criteria
Output: selected_flag   //an array of flags denoting the selected records of QueryTable
Begin
    declare stack_d[][];
    declare top_d[];

//Use cudaMallocPitch API to allocate a 2D array
cudaMemoryAlloc2DArray(stack_d, stack_size);
//Use cuaMalloc API to allocate a temporary stack for QueryData transformation
cudaMmeoryAlloc(top_d, topPointer_size);
//transform the QueryData from the prefix form to the postfix one.
postTrans(QueryData);

for i=1 to 2*numberOfOperand-1 do
    switch(token of postfix) {
        case operand:
                    switch(operator) {
                        //Select items in a specified column of the QueryTable
                        case ">": selectGreater_kernelProgram(
                                QueryTable, stack_d, top_d, selected_flag,
                                column_index);
                        //Sselect items in the specified column of the QueryTable
                        case "<": selectSmaller_kernelProgram
                        //Match items in the specified column of the QueryTable
                        case "=": selectEqual_kernelProgram
                                    :
                                    :
                    }
        case LogicOperator_AND:
                    call selectAND_kernelProgram
        case LogicOperator_OR:
                    call selectOR_kernelProgram
    }
end
```

parallel execution possibilities, minimizing memory and resource stalls, and reducing branch miss predictions (Ailamaki et al. 2001; Manegold et al. 2000; Meki and Kambayashi August 2000; Rao and Ross 1999; Ross 2002). Also, many researchers show that GPU performs well than GPU does for database operations. N. Govindaraju., et al. implemented several SQL operations on NVIDIA GeForce FX 5900 without CUDA framework (Govindaraju et al. 2004). The results demonstrated that performance of SQL operations on GPU is about 2 times faster than the one on CPU. They also designed several GPU-based join algorithms and achieved performance improvement of 2-7X over their optimized CPU-based counterparts (Bingsheng et al. 2008). P. Bakkum and K. Skadron accelerates SELECT queries and share the experiences of GPU implementation of SQLite command processor (Ross 2002; Bakkum and Skadron 2010). In addition to SQL database applications, P. Ferraro, et al., use CUDA to implement query-by-humming on GPU (Ferraro et al. 2009). Their design allows to retrieve a hummed query in a database of MIDI files, with good accuracy, in a time up 160 times faster than other comparable systems. S. Ding, et al. investigate a new approach to build web search engines and other high-performance information retrieve systems (Shuai et al. 2009). The experimental results for their prototype GPU-based system on 25.2 million web pages show promising gains in query throughput.

Each results of prior works gets impressive performance improvement on GPU, but most of these studies focus on specific problem domains or implement primitive query operations for SQL database. As a cornerstone of enterprise applications, developers cannot benefit from database with partial SQL operation implementation. It is the goal of this paper to help the efficiently development of database applications on GPU with CUDA. This paper contributes the design and implementation of a IMDB, called CUDADB, on GPU using CUDA programming model. The experimental results show that the designed IMDB get better performance than most commonly used SQLite database. Besides, some learned experiences are also shared in this paper.

The remainder of the paper is organized as follows. Section 2 surveys some related works and briefly introduce the backgrounds used in the paper. Section 3 describes design issues which should be considered carefully while implementation a database on GPU with CUDA. Section 4 depicts the implementation details of the proposed IMDB. Section 5 presents the experiment results and analysis. And finally, conclusions feature in Section 6.

2 Background

2.1 CUDA overview

CUDA is an extension to C based on a few easily-learned abstractions for parallel programming, coprocessor offload, and a few corresponding additions to C syntax. CUDA gives developers access to the virtual instruction set and memory of the parallel computational elements in CUDA GPUs. There are several advantages over traditional GPGPU using graphics APIs, and they are scattered reads, shared memory, faster

Table 3 Algorithm of greater process in selection query

```
Algorithm SelectGreater_kernelProgram (dataTable, stack, top, selected_flag,
column)
Input: dataTable //the target data table in the GPU memory
       stack // the stack of operators of the query criteria
       top // the top position of the stack
       column // the column index of the data table. Assign one unique column index
              // for each running thread.
Output: selected_flag //an array of flags denotes which record is selected
begin
    for idx = 1 to (the size of data table) // for each parallel running thread
        top[idx]++;
        if (dataTable[column] [idx] > value) then
            stack[index][top[idx]] =1;
        else
            stack[index][top[idx]] =0;
end
```

Table 4 Algorithm of AND process in selection query

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

downloads and read backs, and full support for integer and bitwise operations (Wikipedia, http://en.wikipedia.org/wiki/CUDA). Besides, it also provides building blocks, such as parallel data processing, such as parallel prefix-sum, parallel sort and parallel reduction for parallel data processing algorithms. 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 (Shubhabrata et al. 2007). 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 next section, we will explain how to use prefix sum to calculate the position of selected data and use the proposed algorithm to sort all elements in the data table.

2.2 SQLite overview

SQLite is an ACID-compliant embedded relational database management system implemented in a C library with relative small footprint, and popularly used by modern hand-held mobile devices, such as Apple and HTC smart phones. In contrast to other database, SQLite is integrated with client applications, and can be access by the same application process (Wikipedia, http://en.wikipedia.org/wiki/SQLite). D. Richard Hipp designed SQLite and implemented most of the SQL-92 standard for SQL in year 2000 (Owens). It has biding for a large number of programming languages, and is well suited to embedded systems such as Apple's iOS, Symbian OS, Nokias' Maemo, Google's Android, RIM's BlackBerry and so on. Due to the population usage of SQLite, it is treated as the comparison target of the performance evaluation of our IMDB implementation on GPU.

3 Design issues

3.1 Primitives

CUDA would be the best choice if developers plan to design and implementation IMDB on GPUs. However, CUDA has some limitations which will restrict the IMDB

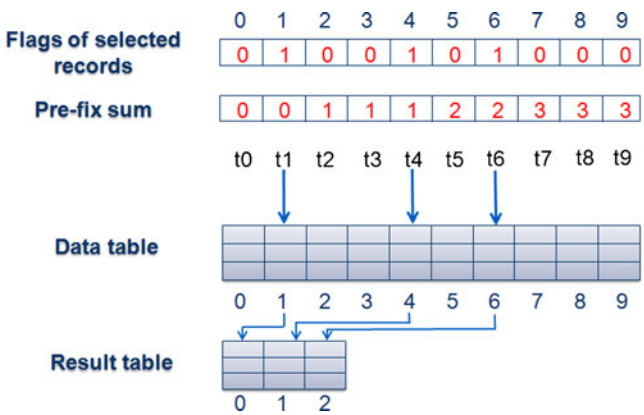

Fig. 3 Process of moving data queried to the result table

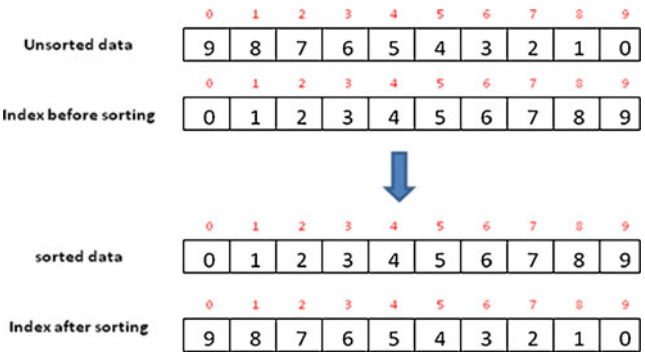

Fig. 4 Modified algorithm of parallel sorting

implementation on GPU. Also, it is not straight forward for developers to use libraries provided by CUDA to sort or compare values in a data table. Here are some primitive issues and suggestions for developers.

(1) Limited registers and shared memory: The number of registers and shared memory in multi-processor (SM) of GPU is finite. One MP is the basic unit to dispatch several threads. The number of registers and share memory used by threads dominate how many threads can be issued in one block.
(2) Divergent branch: The cost of branching within one Warp could be expensive, if threads run on different execution path because they should be serialized by the thread scheduler of the GPU.
(3) The overhead of data transfer between CPU and GPU is large: The bottleneck of GPU-based applications is the data transferred back and forward between GPU and CPU. In the initial stage, all data and tables are loaded to GPU memory to minimize the overhead.
(4) Non-coalesced global memory access: Multiple global memory access is grouped into on memory access if the access pattern of half warp is sequential. If the access pattern is not sequential, GPU should issues as many concurrent memory access as possible.
(5) CUDPP modification: CUDPP is a serial and efficient library in CUDA environment. There are several basic algorithms are included in it. For instance, parallel prefix sum and parallel sorting algorithm are imple-

Table 5 Algorithm of data 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);
      //cudppSort default: ASC. we use cudpp_option_forward by default.

        mvSort_kernelProgram (sortedTable, dataTable,
        number of elements, sortdata_index );

        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
```

Table 6 Algorithm of moving data after 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
```

mented in the CUDPP library (Harris 2008; Garland et al. 2008). Unfortunately, those algorithms in CUDPP support 1-D array as input and another 1-D array as output, but are not suitable for sorting elements in data table column by column. We modified the sorting function, and declare an additional parameter of 1-D array initialized with sequential and ascending numbers. By moving both data and the index value of the additional parameter to corresponding position, a sorted data set could be presented by the index array whose elements point to data values in a sequential order.

3.2 Scope of CUDADB implementation

The scope of this paper is to design and implement a IMDB support SQL-92 standard of SQL. The following table shows the major functions which cannot be executed in parallel on CPUs, and are supported in CUDADB (Table 1).

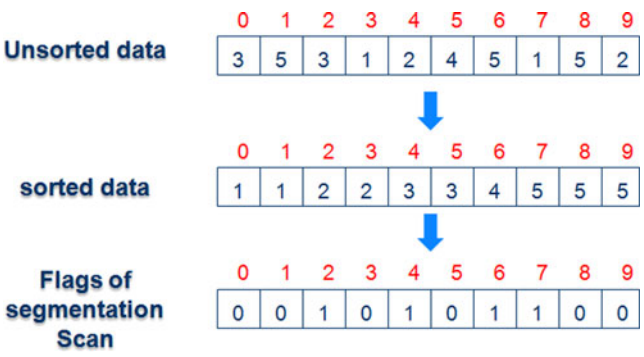

Fig. 5 Setting flags of segmentation Scan

4 System implementation

Conventional database systems use tree structure, such as B-Tree, to manage data or indices. The searching or sorting algorithms are optimized in sequential computation environment, but are obviously not suitable or cannot be used directly for parallel computer architectures. (Atallah et al. 1989; Suri et al. 2006; Haboush and Qawasmeh 2011) To fully utilize computing power of GPUs and reduce the opportunity of coalesced memory access, CUDADB uses two dimensional column-major arrays for continuous memory access. Each data in the 2-D arrays is stored in column major, and each column is mapped to a corresponding row of a GPU memory table. Figure 1 shows the relationship between column-major data structure and GPU memory. Based on the column-major data structure.

4.1 Selection query

How Selection Query works is explained in this section. The functions in bold type are not CUDA programming

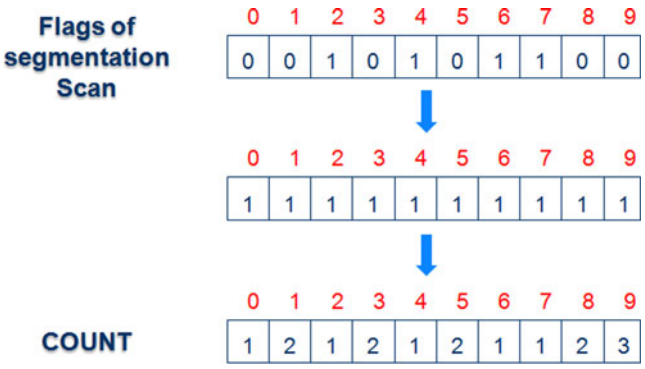

Fig. 6 Process of COUNT function

Table 7 Data Table and Result Set Table

group	*value*
1	*145*
1	*254*
2	*645*
1	*399*
2	*645*

interfaces, and are just used for clearly describing each step of the algorithm. In addition, because the implementation details are similar for most of the proposed functions, such as selectGreater_kernelProgram, selectSmaller_kernel-Proam and selectEqual_kernalProgram, here we just show the steps of selectGreater_kernalProgram. After getting condition parameters, the process compares each record column by column. Assume that there exists a record in

Table 8 Algorithms of Data Group

```
Algorithm DataGroup (resultOfSelect, QueryData, number of data queried)
Input: resultOfSelect (a data table of Selection Query result)
Output: sortedTable
//output: QueryData
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:
                                ;
        }
```

Table 8 (continued)

```
        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
```

each column matching one of the condition parameters respectively. Then, the record is selected and then set as "1" to the corresponding position of a flag array. After the matching process, we check the flag array, and copy these records marked as "1" to another result table. To parallelize the Selection Query operation, a dedicated thread is assigned to a record. All threads run in parallel, and each thread compares all columns with the assigned records iteratively.

To guarantee the integrity of selection operation, the "AND" operation has higher execution priority than the "OR" operation. Before processing the Selection Query operation, the original prefix notation has to be transformed into postfix notation. As shown in Fig. 2, one stack is used by threads for each record while transforming prefix notations into postfix notations. The GPU scheduler is responsible for keeping the serialized condition for all execution paths especially when branches within the same block occurred. Because the cost of branch execution paths is expensive, host computer also have to be responsible for the partial flow control to avoid divergence branches. After the computation, the value of the bottom element in the stack is selected. The selected record will be moved to another 2-D result array and transmitted to the host. Once the 2-D result array comes out, the final stage will be the sorting processing of the values in the result array.

Sorting elements of the result array could be easily implemented by using the *cudppScan* function provided by the CUDPP library (Shubhabrata et al. 2007). The *cudppScan* performs a prefix sum operation on the flag array in GPU memory and outputs an array of corresponding position. The details of the design of Selection Query are articulated in Tables 2, 3 and 4.

Figure 3 shows the algorithm of moving original data records to the result table. Assume we have a 10-record table, and each record is associated with a thread respectively. In Fig. 3, t0~t9 means threads with tread ID [0]~thread ID(Qihang et al. 2008). 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", it means that the value of corresponding address in the result array of pre-fix sum function is the new position of selected record in result table.

Table 9 Algorithm of AVG process in 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
```

4.2 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. As shown in Fig. 4, the index of data denoting the original position is necessary for sorting key-value pairs. As we mentioned in previous sections, by modifying the *cudppSort* function, two 1-D arrays are used as the input parameters. The outputs of the modified *cudppSort* include a sorted array and an index array specifying the position of the original data array. Tables 5 and 6 illustrates the implementation details of Sorting Data on GPU. Values of address[i] in the index array means that the i^{th} element of the original array was moved to address [i] after sorting (Figs. 5 and 6).

4.3 Data grouping

In conjunction with the aggregation function, the Data Grouping is used to group the result set among several columns. Taking the following table as example, the left part of Table is the original data table, and the right part is result set table after executing Data Group operations (Table 7).

In the beginning, data scattered irregularly over the data table. In order to divide data into several groups, the entire table has to be sorted based on the identification of each column. After the sorting process, the data of the same group are gathered together in one table. Then, we can use *cudppSegmentationScan* function provided by the CUDPP library to calculate the aggregation function. Flags will be set to dedicate the starting address of every group according to sorted columns.

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 *cudppSegmen-*

Table 10 Hardware configuration

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

Table 11 NVIDIA 9800GT specification

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

tationScan 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*. As for the COUNT function, we need an additional array with all values set as '1' in it. The output of *cudppSegmentationScan* with sum algorithm performed on this array will be the result of the COUNT function. Finally, the result of AVG can be calculated through divide SUM by COUNT. The basic concept of Data Grouping was listed in Tables 8 and 9.

4.4 Data updating, deleting and insertion

The idea of data update is trivial. There is a flag array used by Selection Query to denote the selected records. By using the same flag array, each record with marked notation in the flag array will be updated in parallel automatically by every thread. Like Data Update, the flag array is used to specify which data is selected to be deleted. Because the remaining data should be moved from original table to new table, the flag are reversed to denote the reserved data. The reserved data are moved from the original table to the result table. Finally, the original table was freed from memory and replaced by the result table. Again, the idea of inserting data is also easy to implement. New data is transmitted to GPU memory and inserted to the position next to the last records of data table. Similar to the design of data insertion, based on the flag array of selection query, selected data are inserted to the result table.

Table 12 Software Configuration

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

Table 13 Sample Test Data

Column 1	Column 2	Column 3
2	50	598

5 Experimental analysis

5.1 Experimental setup

In our experiment, a 4-core CPU and GeForce 9800 GT is used for performance evaluation and comparison. The detail of hardware configuration information is described as following (Tables 10, 11 and 12).

Detail of GPU specification is depicted as the following table. There are 8 SP (Stream Processor) included in each MP (multi-processor). Each SP can process one single precision floating pointer calculation. With ideal condition, the GPU can processes 8x16=128 single precision floating pointer calculations simultaneously.

The software environment for the performance evaluation is as following.

5.2 Experimental analysis

In the performance evaluation experiments, we log the execution time for both CUDADB and SQLite in-memory database by adding fixed number of data records for each test run. To simplify the evaluation and skip the duplicate test runs, the test cases of Selection, Data Grouping, and Insert Data by Selection Query are exercised for performance analysis. The comparison between CUDADB and SQLite is made based on the conditions of total number of records and number of records in a query result set. Finally, we try figure out the "turning point" for data tables with variant numbers of data records for both CUDADB and SQLite.

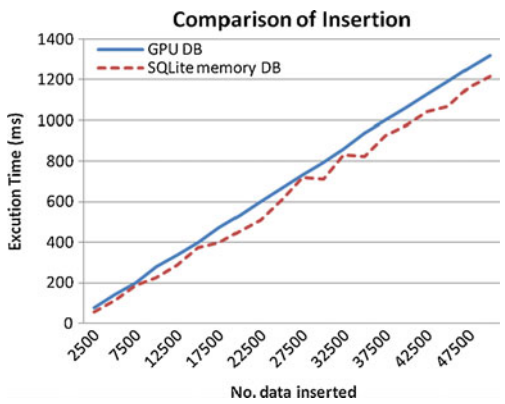

Fig. 7 Execution time comparison of insertion operation

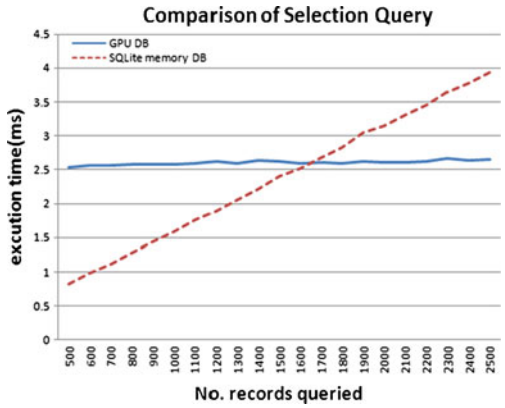

Fig. 8 Execution time comparison of selection

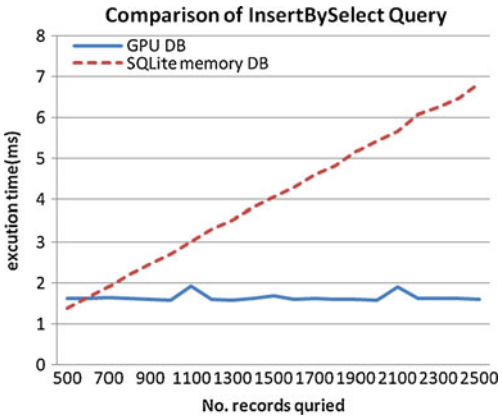

Fig. 10 Execution time comparison of data insert

5.2.1 The pattern of test data

For performance estimation, the experiments use the test data provided by official SQLite benchmarks. There are three columns in the data table. The data in first column of each record is a unique random number between 1 to total number of records. The second column is the group number which will be the number of 1 to 100. 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. All the experiments are tested for 50 times, and the averages of result data are used for the performance comparisons and discussions (Table 13).

5.2.2 Performance comparison

Insertion In Fig. 7, we can see the insertion function of CUDADB suffered from the overhead of transferring data form host memory to GPU memory, although the difference between execution times of two systems is about 20 ms to 120 ms. Average increased execution time for CUADB is 65.512 ms for each test run by adding 5000 records respectively. Because we expand twice times of the table size for each time when the 2-D array is full. The expanding overhead takes another 10.453 ms in average.

Selection query As showed in Fig. 8, CUDADB takes 2.598 ms in average to complete a Selection Query operation. The execution time of CUDADB is stable and independent to the total number of target data records. The performance of SQLite memory DB is better than CUDADB when the number of target data records is less than 1700 which is the turning point of the Selection Query operation.

Data grouping In Fig. 9, the execution time of CUDADB increases smoothly when the total number of target data records goes from 500 to 2500. We believe that the root cause of the increased execution time is the sorting processing time before executing the aggregation function. Of course, the number of records queried also affects the execution time of processing the aggregate functions. We will have further discussion about this issue in later sections. The results show that CUDADB performs better when the number of target data records is larger than 600 (Fig. 10).

Data insert according to selection query Similar to the evaluation results of Selection Query and Data Grouping, the execution time of CUDADB is almost independent

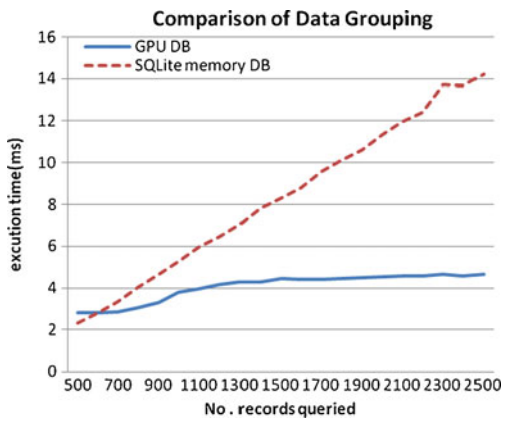

Fig. 9 Execution time comparison of data grouping

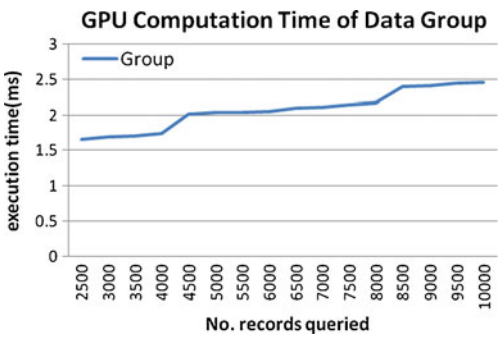

Fig. 11 GPU execution time of data group

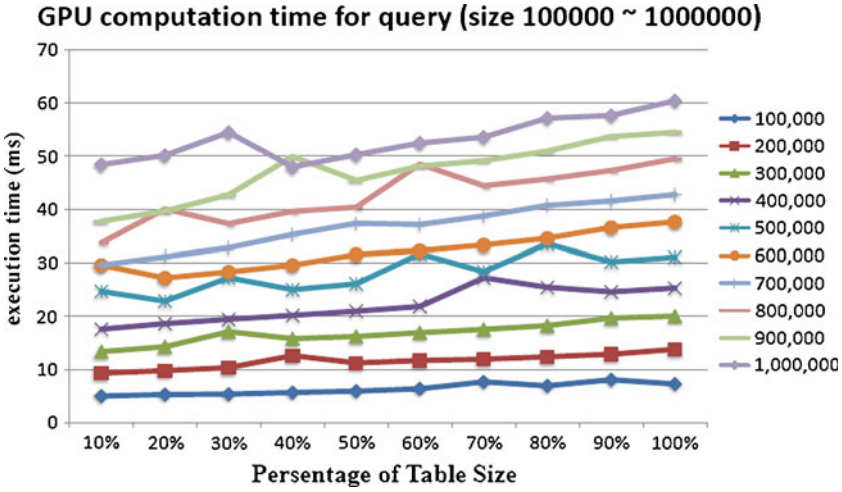

Fig. 12 GPU computation time of selection query

from the number of target data records. The two special cases of 1,100 and 2,100 records for CUDADB are caused when the inserted data exceed the data table size. The table size expands twice times of the original one to accommodate all data records.

5.2.3 Analysis of GPU computation time of data grouping

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 analyze the performance. At first, we evaluated the variation of GPU execution time by increasing the number of records queried from 2,500 records to 10,000 records and total number of data is 100,000 records in the data table. As we can see in Fig. 11, the more records queried, the more GPU time for Data Grouping step.

There are two jumps at 4,500 and 8,500 records queried. Prefix 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. The case of 4500 exceeds the power degree of 4096. It needs to issue twice number of threads which will take more execution time than the case of 4096.

We evaluated the GPU computation time of Selection Query in Data Grouping process respectively with total number of 100 K, 200 K, 300 K, 400 K, and 500 K records in data table. As showed in Fig. 12, GPU computation time of Selection Query is independent from the number of records queried. According to our implemented methods, one thread is designate on one record for execution selection query process. Due to the restriction 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. Selection Query operations perform stably. The total execution time is a portion to the total number of target data records, but is independent from the number of data records in result sets (Fig. 13).

Finally, we evaluated the total GPU time of Data Grouping. The proportion of number of date records in a result set to the total number of records in data table is small, so the impact of numbers of records in result set to the GPU execution time is pretty small.

5.2.4 Evaluation of turning points

It is interesting in the evaluation of turning points between CUDADB and SQLite. A turning point stands for the ratio of the number of data records in a result set to the total number of records in a target table. As shown in Fig. 14, each data line indicates the relationship of the SQL operation, and turning points. For example, the red line represents the turning points between CUDADB and SQLite. It also implies the performance differentiation between CUDADB and SQLite for the implementation of each SQL operation. CUDADB performs better in update and insert operations than the select and sort operations.

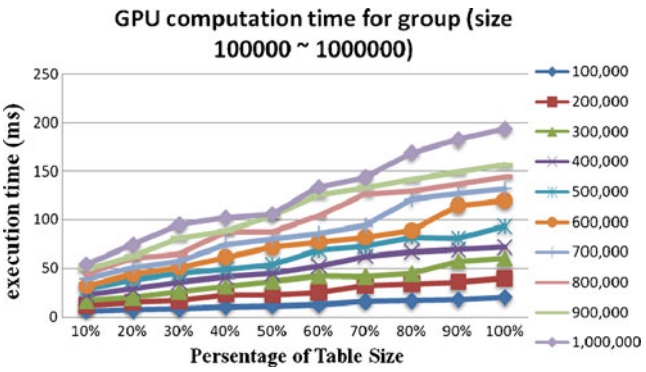

Fig. 13 Total GPU computation time of data grouping

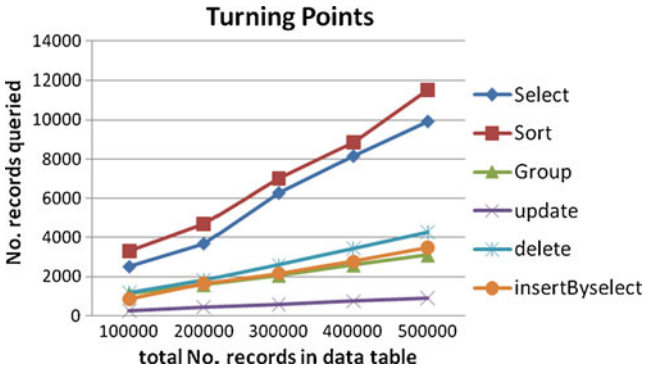

Fig. 14 Turning points of GPU DB and SQLite memory DB

Table 14 shows the turning point ratio of between our CUDADB implementation and SQLite is from 0.16 to 2.06.

6 Discussion and conclusion

▪Implementation Limitations

Most of our implementations are based on CUDA functions, The CUDADB is implemented by re-design the API signatures by adding additional data structures and parameters, and by re-wrapping the structures and calling sequences of CUDA function calls. Hence, the implementation is restricted to support integer type due to the same constraints of CUDA functions. In addition, there will be several issues which will affect the result of our implementation for changes of each version. For example, the CUDA function 'cudppSort' behaves different in version 1.0a, and 1.1. There should have minor implementation changes for each version. Besides, in current version of CUDADB, parallel string data query, join query, and concurrent data query are not supported.

▪ Potential CUDADB Applications

It is much different from the performance of GPUs of general purpose computers and handsets. It is not expected to deploy our proposed algorithms in hand-held devices. Instead, the result of our proposed IMDB algorithm will inspire those scientific computing applications to change from the file bases to the database paradigms. Take our previous ocean data extraction project as an example, all data are stored in large volume of files, and it takes very long time to extract right data out from those files (Chang and Cheng). Another good example is the employment of CUDADB for the peer to peer applications (Jung 2010). The proposed CUDADB mechanism indeed helps to speed up the performance of data extracting processing. It would be a new trend for distributed supercomputing infrastructures to join GUPS together to deliver high-performance computations. It is also the purpose of this paper to have the CUDADB implementation. Some applications with large data set, such as molecular simulations and our previous study on ocean data extraction/simulation of scientific computing, are suitable for such environment. GPUGRID.net is another example for this (GPUGRID, http://www.gpugrid.net/). The CUDADB tries to enhance the performance of these application by utilizing the idea of in-memory database because the dispatched database size of each volunteer are suitable for in-memory computing after the data decomposition processing of the server.

▪ Conclusion

Modern GPU indeed brings remarkable improvement for the performance of data intensive applications. In this paper, we first survey the background of existing main memory data base and CUDA programming model. Then, the implementation of CUDADB is introduced. The performance evaluation and comparison between CUDADB and SQLite are also discussed. The experimental results show that the execution time of CUDADB operations is independent from the number of data records in a result set, and increase smoothly in a specific ratio to the total number of records in the target data table. From the performance analysis, the turning points can be treated as the differences between CUDADB and SQLite for each SQL operations. The idea of turning point can also be used by developers to design GPU-enabled database applications in the future. Generally speaking, the more records in the result set, the more execution time

Table 14 Turning Point ratio of each SQL operations

Function name	Ratio of No.of records in result set to the total No. of data 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%

SQLite operations will take. CUDADB performs well, and the ratio is about 0.161% to 2.061% for different functions.

References

Ailamaki, A., DeWitt, D. J., Hill, M. D., & Skounakis, M. (2001). "Weaving Relations for Cache Performance," In Proceedings of the 27th International Conference on Very Large Data Bases, pp. 169–180, San Francisco, USA.

Akiyma, Y. "Large-scale Bioinformatics Applications on Multi-node GPU Environment," URL: http://research.nvidia.com/content/CAS_CCOE_Part4

Atallah, M. J., Kosaraju, S. R., Larmore, L. L., Miller, G. L., & Teng, S.-H. (1989). "Constructing Trees in Parallel." in Proceedings of the first annual ACM symposium on Parallel algorithms and architectures, pp. 421–431

Bakkum, P. & Skadron, K. (2010). "Accelerating SQL Database Operations on a GPU with CUDA." In Proceedings of the 3 rd International Workshop on GPGPU, pp.94–103, New York, USA.

Chang, Y. S. & Cheng, H-T. "A scientific data extraction architecture using classified metadata," Journal of Supercomputing, doi:10.1007/s11227-010-0462-7.

Ding, S., He, J., Yan, H., & Suel, T. (2009). "Using Graphics Processors for High Performance IR Query Processing." In Proceedings of the 18th International Conference on World Wide Web, pp. 421–430, April. 20–24, 2009, Madrid, Spain.

Ferraro, P., Hanna, P., Imbert, L. & Izard, T., (2009). "Accelerating Query-Humming on GPU." In Proceedings of the 10th Information Society for Music Information Retrieval Conference, pp. 279–284.

Garland, M., Le Grand, S., Nickolls, J., Anderson, J., Hardwick, J., Morton, S., et al. (2008). Parallel Computing Experiences with CUDA. *IEEE in Micro, 28*(4), 13–27.

Govindaraju, N. K. Lloyd, B., Wang, W., Lin, M. & Manocha, D. (2004). "Fast Computation of Database Operations using Graphics Processors." In Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data, pp. 215–226, Paris, France.

Haboush, A., & Qawasmeh, S. (2011). Parallel Sequential Searching for Unsorted Array. *Research Journal of Applied Science, 6*(1), 70–75.

Harris, M. (2008). "Parallel Prefix Sum (Scan) with CUDA," NVIDIA.

He, B., Yang, K., Fang, R., Lu, M., Govindaraju, N. K., & Luo, Q. et al. (2008). "Relational Joins on Graphics Processors." In Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data, pp. 511–524, Vancouver, BC, Canada.

Jung, J. J. (2010). Reusing Ontology Mappings for Query Segmentation and Routing in Semantic Peer-to-Peer Environment. *Information Sciences, 180*(17), 3248–3257.

Lindholm, E., Nickolls, J., Oberman, S., & Montrym, J. (2008). NVIDIA Tesla: "A Unified Graphics and Computing Architecture". *IEEE Micro, 28*(2), 39–55.

Liu, Z., & Ma, W. (2008). "Exploiting Computing Power on Graphics Processing Unit," In Proceedings of International Conference on Computer Science and Software Engineering, pp. 1062–1065, Dec.

Manavski, S. A. (2007). "CUDA Compatible GPU as an Efficient Hardware Accelerator for AES Cryptograph." In Proceedings of International Conference on Signal Processing and Communication, ICSPC 2007, pp.65–68, November.

Manegold, S., Boncz, P., & Kersten, M. L. (2000). "What Happens During a Join? Dissecting CPU and Memory Optimization Effects". In Proceedings of the 26th International Conference on Very Large Data Bases, Cairo, Egypt, pp. 339–350, September 10–14, San Francisco, USA.

Meki, S., & Kambayashi, Y. (August 2000). Acceleration of Relational Database Operations on Vector Processors. *Systems and Computers, 31*(8), 79–88.

Nickolls, J., Buck, I., Garland, M., & Skadron, K. (2008). Scalable Parallel Programming With CUDA. *ACM Queue, 6*(2), 40–53.

Owens, M. The Definitive Guide to SQLite, ISBN-13: 978-1-59059-673-9

Pushpa, S., Vinod, P., & Maple, C. (2006). "Creating a Forest of Binary Search Trees for a Multiprocessor System." in Proceedings of International Symposium on Parallel Computing in Electrical Engineering (PARELEC'06), pp. 290–295.

Qihang Huang, Zhiyi Huang, Paul Werstein and Martin Purvis, "GPU as a General Purpose Computing Resource," In Proceedings of PDCAT'08, pp. 151–158, Washington DC, 2008.

Rao, J. & Ross, K. A. (1999). "Cache Conscious Indexing for Decision-Support in Main Memory." In Proceedings of the 25th International Conference on Vary Large Data Bases, pp. 78–89.

Rodrigues, C. I,. Hardy, D. J., Stone, J. E., Chulten, K., & Hwu, W.-M. W. (2008). "GPU Acceleration of Cutoff Pair Potentials for Molecular Modeling Applications." In Proceedings of the Conference on Computing Frontiers, May 5–7.

Ross, K. A. (2002). "Conjunctive Selection Conditions in Main Memory." In Proceedings of the 21th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, pp. 109–120.

Schatz, M., Trapnell, C., Delcher, A., Varschney, A. (2007). "High-Throughput Sequence Alignment Using Graphics Processing Units." *BMC Bioinformatics, 8*(1).

Sengupta, S., Harris, M., Zhang, Y. & Owens, J. D. (2007). "Scan Primitives for GPU Computing." In Proceedings of the 22th ACM SIGGRAPH Symposium on Graphic Hardware, pp. 97–106, Aug. 4–5.

Chengen, W. & Lida, X. "Parameter mapping and data transformation for engineering application integration." *Information Systems Frontiers, 10*(5), 589–600.

Wynters, E. (2011). Parallel Processing on NVIDIA Graphics Processing Units Using CUDA. *Journal of Computing Sciences in Colleges*, *26*(3), Jan.

Yuan, Z., Zhang, Y., Zhao, J., Ding, Y., Long, C., Xiong, L., et al. (2010). Real-time Simulation for 3D Tissue Deformation with CUDA Based GPU Computing. *Journal of Convergence Information Technology, 5*(4), 109–119.

Zhang, Y., Frank, M., Cui, X. & Potok, T. (2011). "Data-Intensive Document Clustering on Graphics Processing Unit Clusters." *Journal of Parallel and Distributed Computing, 71*(2), Feb.

Yue-Shan Chang received his PhD Degree from Computer and Information Science at the National Chiao Tung University in 2001. He joined the Department of Electronic Engineering of the Ming Hsing University of Science and Technology in August 1992. Since August 2001, he had been an Associate Professor. Since August 2004, he joined the Department of Computer Science and Information Engineering, National Taipei University, Taipei County, Taiwan. His research interests are in distributed systems, web service composition, information retrieval, mobile computing and grid computing.

Ruey-Kai Sheu received his MS and PhD degres from National Chiao-Tung University in 1998 and 2001 respectively. Sheu joined the W&Jsoft Inc. as the R&D leader from year 1999 to 2007. He has been an Assistant Professor at the Department of Computer Science, Tunghai University, Taichung, Taiwan from 2007 till now. His current

research interests include Distributed Objects, Internet Technologies, and Software System Integration, and Data Leak Protection.

Shyan-Ming Yuan received his BSEE degree from National Taiwan University in 1981, his MS degree in Computer Science from University of Maryland, Baltimore County in 1985, and his PhD degree in Computer Science from the University of Maryland College Park in 1989. Dr. Yuan joined the Electronics Research and Service Organization, Industrial Technology Research Institute as a Research Member in October 1989. Since September 1990, he has been an Associate Professor at the Department of Computer and Information Science, National Chiao Tung University, Hsinchu, Taiwan. He became the Professor in June 1995. His current research interests include Distributed Objects, Internet Technologies, and Software System Integration. Dr. Yuan is a member of ACM and IEEE.

Jyn-Jie Hsu received his MS degree from National Chiao-Tung University (NCTU), Hsinchu, Taiwan. He is current a graduate student of NCTU, and is interested in developing GUP-related applications.