2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum

Task Scheduling for GPU Accelerated Hybrid OLAP Systems with Multi-core Support and Text-to-Integer Translation

Maria Malik, Lubomir Riha, Colin Shea, Tarek El-Ghazawi
Department of Electrical and Computer Engineering
The George Washington University
Washington, DC, USA
{maria_my, lubomir, cshea05, tarek}@gwu.edu

Abstract—OLAP (On-Line Analytical Processing) is a powerful method for analyzing the excessive amount of data related to business intelligence applications. OLAP utilizes the efficient multidimensional data structure referred to as the OLAP cube to answer multi-faceted analytical queries. As queries become more complex and the dimensionality and size of the cube grows, the processing time required to aggregate queries increases.

In this paper, we are proposing: (1) a parallel implementation of MOLAP cube using OpenMP; (2) a text-to-integer translation method to allow effective string processing on GPU; and (3) a new scheduling algorithm that support these new features.

To be able to process string queries on the GPU, we are introducing a text-to-integer translation method which works with multiple dictionaries. The translation is necessary only for the GPU side of the system. To support the translation and parallel CPU implementation, a new scheduling algorithm is proposed. The scheduler divides multi-core processor(s) of a shared memory system into a processing partition and a preprocessing (or translation) partition.

The performance of the new system is evaluated. The text-to-integer translation adds a new vital functionality to our system; however it also slows down the GPU processing by 7% when compare to original implementation without string support.

The performance measurements indicate that due to the parallel implementation, the processing rate of the CPU partition improves from 12 to 110 queries per second. Moreover, the CPU partition is now able to process OLAP cubes of size 32 GB at rate of 11 queries per second. The total performance of the entire hybrid system (CPU + GPU) increased from 102 to 228 queries per second.

Keywords: OLAP, GPU acceleration, OpenMP, scheduling

I. INTRODUCTION

The rapid growth in the amount of text data that needs to be processed demands the creation of fast and effective data analysis techniques. The ability to efficiently analyze and quickly view this massive amount of data exceeds the processing power of today's fastest homogenous computers. Even with advanced multi-node architectures, the response time and cost are prohibitive. The application of node heterogeneous tasking and its application to OLAP is still a rich area for research. On-line Analytical Processing (OLAP), has become a crucial part of many organizations, as it contributes to business-oriented decision-making by identifying, extracting, and analyzing data. OLAP tasking is highly interactive and response time is critical. The fundamental structure of OLAP is a data cube. The data cube allows views of data from multiple perspectives and at a variety of levels. The individual cells of a cube contain data related to elements along each of its dimensions. This data is based on aggregations from fact tables. Each cell may be of a different granularity and/or based on different dimensions. The number of aggregations, count, sum, average, maximum etc., are required is determined by every possible combination of the dimension granularity.

Recently, there has been a growing interest in GPUs because of their high performance processing capabilities for many scientific, business intelligence and engineering applications. The latest NVIDIA GPUs provide concurrent thread execution, which enables unified system and GPU memory pointer access and arithmetic.

The current paper highlights two key issues to analyze online data effectively. First, we are proposing a system that provides string support for the GPU. In this algorithm we have introduced a string dictionary system that translates literal data into integers to take advantage of the enhanced GPU performance when working with the integers. Fact tables from renowned TPC-DS benchmark have been used for evaluation of the translation performance.

Second, we are proposing a parallel version for CPU based OLAP cube processing. This version is using a shared memory model and is implemented in OpenMP. We have also built a performance model of the parallel implementation that is used for its performance estimation. The performance estimation is an essential functionality required by the scheduling algorithm that allows the scheduler to decide which resource (CPU or GPU) to use for answering different queries.

Third, we are proposing a new scheduling algorithm that now supports text-to-integer translation and multiple CPU and GPU partitions. The CPU is virtually partitioned into two partitions. The first partition uses a parallel OpenMP implementation for OLAP cube processing and the second partition executes the string-to-integer translations for queries that are scheduled to the GPU.

978-0-7695-4676-6/12 $26.00 © 2012 IEEE
DOI 10.1109/IPDPSW.2012.259

This paper describes task scheduling of OLAP queries utilizing NVIDIA's Fermi GPU architecture with concurrent kernel execution and multicore CPUs with shared memory. The work builds on previously establish work in aggregation [19, 13], GPU processing [9], and scheduling [2]. Section 2 presents related work. Section 3 describes our approach to accelerating OLAP on the GPU. Section 4 provides the performance evaluation. Finally, Section 5 provides our conclusions.

II. Related Work

In this section, we highlight related research in OLAP algorithms, parallel aggregation, heterogeneous processing, task scheduling, and string dictionary.

A. OLAP Cube Algorithms

There is a large amount of research on sequential computation of the OLAP data cube. The algorithms can be classified into three categories: top down, bottom up, and array-based.

The PipeSort algorithm [1] is an example of a top down approach. This algorithm works from fine granularity views to coarse granularity. It was first developed for standard relational tables. Its end goal is to establish the minimum cost spanning tree from the lattice. The PipeSort can be described as seeing the sort orders for group-bys at successive levels, starting from the bottom of the tree and working up.

The bottom up computation algorithm was developed to address the cost of additional cube dimensions. When the data cube doubles in size, the data often becomes sparse, thus yielding longer sort times. The bottom up algorithm aggregates and sorts based on a single dimension. It recursively partitions the current results to aggregate at successively finer degrees of granularity.

Array-based algorithms [20] arrange data into consecutive vectors of a given size. The algorithm uses the structure of the cube to support aggregation operations, eliminating sorting. This leads to cuboid aggregation in sections without complete transversal of the data set. The end product of the algorithm is a minimum memory-spanning tree.

B. Parallel Cube Aggregation

Zhao, Deshpande and Naughton [20] describe the array-based algorithm for computing the Cube for Multidimensional OLAP systems. They utilize the work first proposed by Sarawagi and Stonebraker [13]. An n-dimensional array is chunked into smaller n-dimensional chunks and each is stored on disk. The array chunk should correspond to the blocking size of the disk and all blocks should be the same size. Zhao and Deshpande take the blocks one step further and compress arrays that have less than 40% of their cells filled. They use a chunk-offset compression. They design an algorithm that first constructs the minimum size-spanning tree for the group-bys of a cube and then it can compute any group-by of a cube from its parent assuming it has a minimum size. The advanced algorithm overlaps computations of different group-bys avoiding the multiple scans required by a naïve algorithm.

Liang and Orlowska [10] propose four different algorithms for parallelizing aggregations of multi-dimensional cube creation. The first is a naïve parallel algorithm that consists of two phases. In the first phase each processor computes a fragment of the relation table having n attributes and tuples. The second phase merges the data cube fragments to form the original data cube through rounds of parallel merging. The second algorithm is a parallelization and expansion of "the smallest parent method" [5] through scheduling for the cube computation by establishing an estimated cost tree. The third algorithm, a matching-based algorithm, yields accurate size estimation at each node [14]. Preceding layer by layer it looks for the best method of computing the nodes at the next lower level from the current layer. This results in an expansion of the original algorithm that accounts for processor/node allocation costs for multiple trees or subset of trees used to in the computation. The final algorithm is a hybrid parallel algorithm that combines the smallest parent method and the matching-base algorithm. This algorithm finds a directed minimum spanning forest of a weighted graph, and allocates tasking according to their weights.

C. Heterogeneous Computing

General-purpose computing on graphics processing units has gained in popularity. This is driven by the extreme parallel computational access at extremely low cost.

Using a GPU as a co-processing unit for database operations was first examined by Govindaraju, Lloyd, Wang, Lin, and Manocha [6]. Govindaraju, et al, presented a series of instruction optimizations that at the time exploited the native texture based units in GPUs. As a general SIMD instruction programming model had yet to be developed. Their results have been surpassed by the release of subsequent hardware advances and CUDA SDK advances. Govindaraju, Lloyd, Wang, Lin and Manocha do present a series of still relevant methods for dealing with SQL queries, predicates, and Boolean combinations. They assume that simple predicates can be evaluated via a depth test and a stencil test. More complex predicates are evaluated by transformation into semi-linear queries. They exploit the stencil buffer and texture used repeatedly for evaluating operations.

Kaczamrski [8] presents two different parallel algorithms for CPU and GPU Cube creation. The most prevalent point to this work is the clear understanding of the removal of memory bank conflicts when accessing cube intersections. In the worst case it is seen that a naïve GPU solutions is 20x slower than a CPU solution. Simplifying data access points per thread to a single location will alleviate conflicts. An algorithm is introduced in which a

single thread reads two data records, n and n-1, and calculates the cube intersections for them. The algorithm exploits the localization of data by having the beginning and end points of all the sequences for a data set. One down side to this algorithm is the waste of global memory bandwidth due to redundant reads of neighboring data input records. The author suggests using a thread block's shared memory as a short-term cache for global memory reads.

Lauer, Datta, Khadikov, and Anselm [9] present the latest OLAP and GPU research to date. Here, the authors using aggregation with multiple weighted aggregations. They combine a parallel "brute-force" scan of the fact table and parallel techniques of aggregations. This is broken into a series of 4 steps: preprocessing on the CPU, Parallel Table Scan on GPU, parallel reduction on the GPU, and Final Aggregation and Post processing on CPU. The preprocessing on the CPU begins by breaking down the coordinates of the requested aggregated cell. In the second step, the parallel table scan on the GPU occurs after the query parameters produced by the CPU preprocessor, have been copied to the GPU memory. The thread checks to see if the tuple contains a value in the given range. After checking all dimensions, the thread reads the value from global memory multiplies it with combined weight and adds the product to a variable storing the thread's intermediate value. Step 3 is parallel reduction on the GPU that will provide one aggregated value written to a pre-allocated array in global memory. The final step is a final aggregation and post processing on the CPU. Due to the small number of intermediate values the reduction will not efficiently utilize the GPU so it is performed on the CPU.

D. Task Scheduling

Task scheduling as it pertains to this body of work can be encompassed as a reconfigurable computing task. In this area the scheduler must account with resource utilization, throughput, computation time, memory access time, IO bottlenecks and processor speed. The technique used to schedule may be static or dynamic.

The linear programming technique presented by Praskash [12] and the iterative search algorithm by Yen [17] yield quality solutions at the cost of increased solution search time. Yen's algorithm utilizes a sensitivity-based search and reduces the number of steps over Prakash's. Yen's algorithm generates a displacement vector from the target solution using weighting functions. These functions measure the amount of changes caused by the move. The process is executed iteratively across all processing elements, PE. The PE with the highest sensitivity is chosen for execution.

Critical path reduction based schedulers, algorithms utilizing genetic algorithms [4] and clustering techniques [7], provide speedups over exhaustive searches without sacrificing quality of solution. Hou's [7] clustering technique looks to quickly discover an optimal schedule. The computation cost and communication cost had been accomplished through clustering of tasks to the same PE to reduce communications overhead. Scheduling is based on delay sensitivity after borders of cluster tasking has been adjusted to minimize delay.

Oh's bias-based scheduling algorithm [11] looks to execute iterative searches to minimize the penalty in scheduling tasks. The best possible execution time of a task is calculated on a given PE when optimally scheduled or the best imaginary level (BIL). Oh's algorithm uses the BIM and BIL to restrict the number of PEs the schedule must iterative over, reducing time-to-schedule.

Fast heuristic co-scheduling algorithms, from the heterogeneous computing arena, such as minimal execution time and minimal completion time are utilized for the fast scheduling. Minimal execution time (MET) [15] algorithms use iterative search methods to discover and assign tasks that result in minimal execution time. It ignores the load, communication overhead and other constraints and schedules tasking. This works well on systems with small workloads or with larger intervals between task submissions. The Minimal completion time (MCT) [2] utilizes an interactive search to sort data independent tasks according to their minimal completion time.

E. String Dictionary

Brodal and Fagerberg [21] present static cache-oblivious dictionary structures for strings which provide analogues of tries and suffix trees in the cache-oblivious model. Their approach takes as input, either a set of strings to store, a compressed tree, or a suffix tree, and creates a cache-oblivious data structure.

Aho and Corasick [22] discuss that finite state machine pattern matching scheme for occurrences of large numbers of keywords in text strings. The proposed algorithm is comprised of two parts. First, construct a finite state pattern matching machine from the keywords. Second, use the pattern matching machine to process the text string in a single pass. Whenever the machine finds a match for a keyword, it signals. The construction of the pattern matching machine is time proportional to the sum of the lengths of the keywords. However, the presented algorithm is using finite state machines in pattern matching applications. The accustomed and traditional algorithms provide challenges for constructing finite automation from regular expressions and programming complexities.

Scarpazza, Villa and Petrini [23] explore a class of high-performance exact string searching solutions using the Aho-Corasick algorithm that has been optimized for IBM Cell/B.E. processor. The results of this paper demonstrate that the Cell/B.E. processor has a potential to compete with both general-purpose based solutions and FPGA-based solutions for performance, dictionary search and flexibility. However, the trade-off between dictionary size and search performance is present. With the small dictionary size that fits in the local memories of processing cores, the throughput reaches 40 Gbps per processor. When dictionaries are large,

the typical throughput is between 1.6 and 2.2 Gbps per processor. Moreover, this paper approach is specifically optimized for IBM Cell/B.E. processor.

III. GPU Acceleration of Hybrid OLAP System with Multicore CPUs

A. System Overview

In our previous work [16] we have proposed a heterogeneous system that uses two independent resources to accelerate a hybrid OLAP system. A Hybrid OLAP system uses both a multidimensional database in form of multiple OLAP cubes and a relational database tables to answer queries.

In our system, two resources are used. The first resource is a CPU with a large main memory that works with OLAP cubes. In our new implementation the CPU also performs text-to-integer translation. The second resource is a GPU accelerator and with a smaller but faster global memory. The GPU performs two main tasks: (1) building the cube from relational tables stored in GPU memory and (2) executing queries that are costly for CPU.

The aggregation process in the CPU is limited by two factors: memory size and memory bandwidth. The main memory is expensive and its size is limited, therefore it is not able to store an extremely large OLAP cubes. Only a data cube with limited resolution and therefore smaller size can be pre-calculated and stored. This resolution level is shown in Figure 1 as level *M*.

The second factor is a performance limitation caused by memory bandwidth, which affects the speed of the CPU aggregation. As the size of the cube grows, the time required to search the cube grows as well. At a given cube resolution, the time required for processing the partial cube by a CPU and the time to process raw data in GPU memory reach equilibrium. Then GPU can answer the query as fast as CPU. This level is shown in Figure 1 as level G.

B. Performance Modeling of OLAP Cube Proccessing for System with Multicore CPUs

In a hybrid OLAP system CPUs answer the queries based on data stored in multidimensional database structures such as OLAP cubes. The processing of an OLAP cube is always constrained by memory bandwidth and not by the performance of the CPU. Therefore we can estimate a query processing time based on:

- amount of data that needs to be transferred from memory (it is equal to the size of the sub-cube that needs to be processed)
- memory bandwidth that can be achieved for data structure used to store the OLAP cubes.

In the next two sections we will elaborate both bullets in more details.

C. Sub-cube Size Estimation

Figure 1 demonstrates that one OLAP system can have multiple pre-calculated cubes with different resolutions. For example if the cube has a time dimension, the resolutions in this dimension can be: years (low resolution), months, days, hours (high resolution). As the resolution grows, the size of the cube grows as well. If the size of the cube is larger than the memory size (Figure 1 – level M), the CPU can pre-calculate only part of the cube. Moreover, it is always desirable to respond to the query using a cube with lowest possible resolution to minimize memory accesses and query processing time.

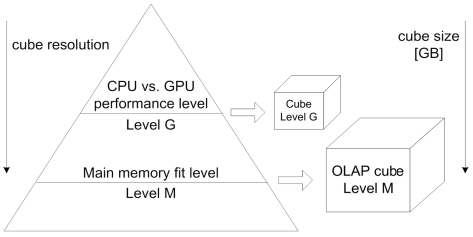

Figure 1. Cube resolution and cube size

The size of the sub-cube and the resolution of the cube can be extracted directly from the query. Suppose the system works with N dimensional cube, then the query Q that exploits all the dimensions is formulated as:

$$Q\big(C_1(f_1,t_1,r_1),\dots,C_L(f_L,t_L,r_L),\dots,C_N(f_N,t_N,r_N)\big), \tag{1}$$

where $C_L(f,t,r)$ is the condition in L-th dimension,
f and t is a range of the condition (f – from, t – to),
r is resolution of the cube needed by the condition.

The resolution R of the cube that needs to be processed to answer the query is defined as the highest resolution that is required by all conditions in query Q:

$$R = \max(r_1, r_2, \dots, r_L, \dots, r_N). \tag{2}$$

If the resolution R is too high and cube is not pre-calculated, the query must be answered by GPU. However, if the cube of resolution R is in the CPU memory then the sub-cube size can be calculated. The total size of the sub-cube is based on parameters f and t in all dimensions and the size of a cell in the cube.

$$SC_{size} = 1024^2 \cdot E_{size} \prod_{i=1}^{N} (f_i - t_i), \tag{3}$$

where SC_{size} is estimated size of the sub-cube in MB,
E_{size} is size of one cell in the OLAP cube in bytes.

The sub-cube size estimation is also shown in Figure 2.

D. Memory Performance Modeling for CPU based OLAP Cube Processing on Multicore Processors

Previously, we developed a single threaded implementation of OLAP cube aggregation algorithm. This implementation achieved a maximum memory bandwidth of 1 GB per second. By developing a more efficient implementation, we were able to increase the memory bandwidth for one thread to 5 GB per second. More importantly, we have developed a parallel OpenMP version that can achieve processing rates from 15 to 20 GB per second for cube sized 128 MB and more. The bandwidth measurements for 1, 4 and 8 threads on our test system are shown in Figure 3.

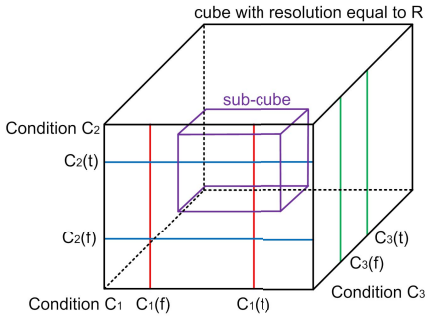

Figure 2. Area of limited search in the case of a three-dimensional cube

Our test system configuration is as follows: the main memory is configured with triple channel 94 GB of DDR3; the main processors are two Intel® Xeons® X5667. The theoretical memory bandwidth of X5667 is 32 GB/s.

Our memory performance model is based on performance measurements of the OLAP cube aggregation algorithm. We have developed an OpenMP benchmark that measures the processing time for different sub-cube sizes. The performance characteristics derived from the results of the benchmark are presented in Figure 4 for 4 threads and Figure 5 for 8 threads.

The processing time is measured for sub-cube sizes from 1MB to 32 GB. For higher precision processing time estimation, the full range is divided into Range A (1 MB to 512 MB) and Range B (512MB to 32 GB) where each range uses a different estimation function f_A and f_B. The estimation functions f_A and f_B are chosen based on best fit for a particular range. The final estimation function for entire range is defined as:

$$P_{CPU}(SC_{size}) = f(SC_{size}) = \begin{cases} f_A(SC_{size}) \; if \; SC_{size} \in Range\; A \\ f_B(SC_{size}) \; if \; SC_{size} \in Range\; B \end{cases}, \quad (4)$$

where SC_{size} is the estimated size of the sub-cube in MB.

For our test system with two Intel Xeon X5667 processors the estimation functions for implementation running 4 threads are:

$$f_{A|4T}(SC_{size}) = 0.0001 \cdot {SC_{size}}^{0.9341}, \quad (5)$$

$$f_{B|4T}(SC_{size}) = 5 \cdot 10^{-5} \cdot SC_{size} + 0.0096. \quad (6)$$

Therefore the processing time of query Q that has to access SC_{size} MB of data from OLAP cube stored in system memory can be estimated as:

$$T_{CPU|4T} = \begin{cases} 0.0001 \cdot {SC_{size}}^{0.9341} \; if \; SC_{size} < 512\; MB \\ 5 \cdot 10^{-5} \cdot SC_{size} + 0.0096 \; if \; SC_{size} > 512\; MB \end{cases}. \quad (7)$$

Similarly, from Figure 5 the estimation functions for the implementation running 8 threads on our test system is defined as:

$$f_{A|8T}(SC_{size}) = 6 \cdot 10^{-5} \cdot {SC_{size}}^{0.984}, \quad (8)$$

$$f_{B|8T}(SC_{size}) = 4 \cdot 10^{-5} \cdot SC_{size} + 0.0146. \quad (9)$$

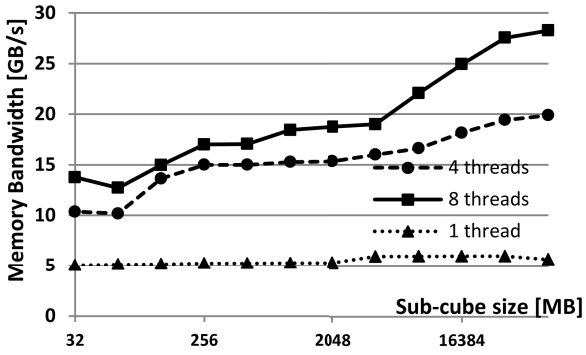

Figure 3. Memory bandwidth for multithreaded OLAP cube processing by CPU

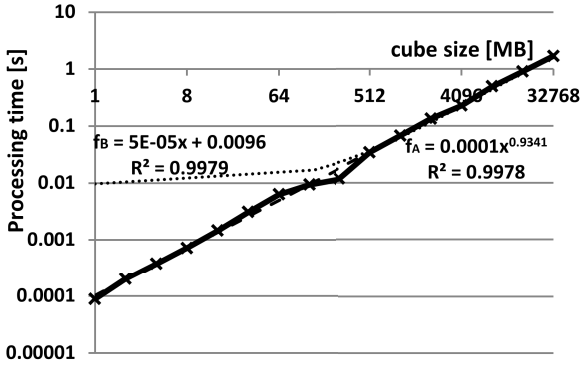

Figure 4. Performance characteristics of OLAP cube processing. Performance measured for multithreaded OpenMP implementation running 4 threads on dual CPU quad core system with Intel® Xeons® X5667.

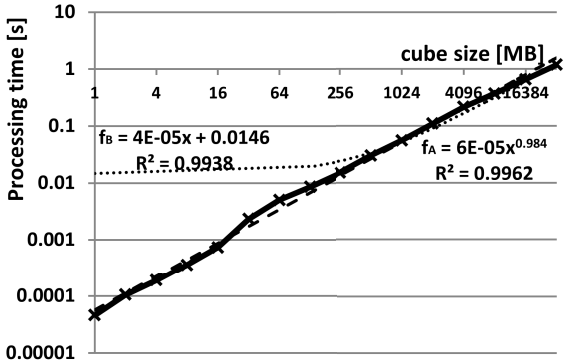

Figure 5. Performance characteristics of OLAP cube processing. Performance measured for multithreaded OpenMP implementation running 8 threads on dual CPU quad core system with Intel® Xeons®

And the time required to process query Q by 8 threaded parallel version is:

$$T_{CPU|8T} = \begin{cases} 6 \cdot 10^{-5} \cdot {SC_{size}}^{0.984} \; if \; SC_{size} < 512\; MB \\ 4 \cdot 10^{-5} \cdot SC_{size} + 0.0146 \; if \; SC_{size} > 512\; MB \end{cases}. \quad (10)$$

This version is using all physical CPU cores of our test system.

E. Performance Modeling and Estimation for GPU Processing with Text-to-Integer Translation

Figure 6 illustrates that the GPU answers queries based on data stored in the form of fact table in the GPU global memory. This memory has a very high bandwidth (up to 144 GB/s for Tesla® C2070 [17]) when using column based access pattern. For the current application, a 1D array memory structure is employed as this data structure provides maximum performance by placing all columns of the table one after another.

The processing performance of the GPU is again limited by amount of data fetched from global memory. The fact table contains two types of columns: dimension columns and data columns. The dimension columns are equivalent to dimensions of an OLAP cube and are used for filtration during query processing. The data columns store the data that are stored in the OLAP cube and are used for aggregation. Processing time of a query therefore depends on how many columns need to be accessed. Note that if the query reads a column it always reads the entire column and not just part of it.

To estimate the processing time of a query Q we propose following decomposition into query Q_D:

$$Q_D = \begin{Bmatrix} C_1(f_1,t_1,l_1),\dots,C_1(f_L,t_L,l_L),\dots,C_1(f_{M1},t_{M1},l_{M1}) \\ C_2(f_1,t_1,l_1),\dots,C_2(f_L,t_L,l_L),\dots,C_2(f_{M2},t_{M2},l_{M2}) \\ \dots \\ C_L(f_1,t_1,l_1),\dots,C_L(f_L,t_L,l_L),\dots,C_L(f_{ML},t_{ML},l_{ML}) \\ \dots \\ C_N(f_1,t_1,l_1),\dots,C_N(f_{I.},t_{I.},l_{I.}),\dots,C_N(f_{MN},t_{MN},l_{MN}) \end{Bmatrix} \quad (11)$$

where $C_L(f_i,t_j,l_K)$ is a condition for dimension L and level K.

The combination of L and K in the condition $C_L(f_i,t_j,l_K)$ points to a particular column in the table (see in Figure 6). It is not essential for every query to contain all conditions. The number of conditions in the query refers to how many columns must be read from global memory. The lower the condition number is, the faster the query processing. The model calculates number of columns C accessed by query Q_D as:

$$C_{Q_D} = \begin{matrix} \#\ of\ filtration\ conditions\ in\ Q_D \\ + \\ \#of\ data\ columns\ processed\ by\ Q_D \end{matrix} \quad (12)$$

The model estimates the GPU processing time of the query Q_D as:

$$T_{GPU} = P_{GPU}\left(\frac{C_{Q_D}}{C_{TOTAL}}, n_B\right), \quad (13)$$

where P_{GPU} is a GPU search performance function,
n_B is a number of multiprocessor per kernel used to process the query.

Since the latest GPU accelerators from NVIDIA based on the Fermi architecture [19] support concurrent kernel execution, our system can divide a GPU unit into independent partitions and process multiple queries in parallel. Each query is then processed by one partition. The performance of a partition is based on number of streaming multiprocessor (SM) allocated to the partition.

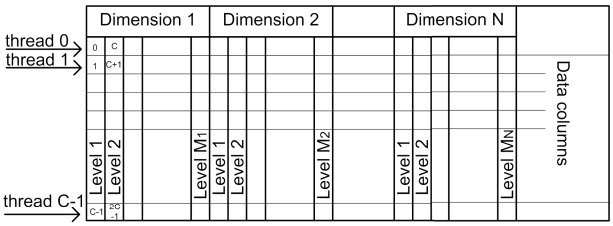

Figure 6. Data organization in the GPU memory

The performance function P_{GPU} is estimated from performance measurements on the database by our benchmark. Performance functions are measured for different GPU partition sizes. The derivation process of performance functions for Tesla C2070 is shown in Figure 8. The partition sizes in this case are 1 SM, 2 SMs and 4 SMs and performance estimation functions are:

$$\begin{aligned} P_{GPU|1SM} &= 0.003\left(C/C_{TOT}\right) + 0.0258, \\ P_{GPU|2SM} &= 0.0015\left(C/C_{TOT}\right) + 0.013, \\ P_{GPU|4SM} &= 0.0008\left(C/C_{TOT}\right) + 0.0065. \end{aligned} \quad (14)$$

Tesla C2070 accelerator has 14 active SMs. If the GPU is not partitioned and therefore only a single query can be processed at a time the performance function is:

$$P_{GPU|14SM} = 0.00021\left(C/C_{TOT}\right) + 0.0020. \quad (15)$$

F. Text Translation Using Multiple Dictioinaries

In addition to the OLAP cube processing, the CPU is also responsible for converting the text parameters of incoming queries to integer parameters. To maximize the use of the GPU memory our system does not store text fields (such as street names, city names or person names) in the GPU memory. Instead the text is translated into integers using dictionaries when the database is built. Therefore every text reference in an incoming query must be translated into integer form before the query is submitted to the GPU for processing.

The implementation uses a smaller dictionary for each text column in the table rather than having one large dictionary for all text columns. This approach allows more precise time estimation of the dictionary search for every incoming query, as smaller dictionaries have smaller time variation of search as well.

Query decomposition for GPU processing shown in (11) is used for identification of columns that are processed by a query. From the same decomposition we can identify which columns are text columns and need to be translated. The translation is then executed before the query is submitted to GPU for processing.

The upper bound of text translation time can be estimated if following parameters are known: CDT_{Q_D} (16) is number of query parameters that need to be translated to integers; length of dictionaries for all columns D_L (17); and dictionary search translation function P_{DICT}. P_{DICT} of our test system is shown in Figure 9 and (17).

$$CDT_{Q_D} = \#of\ filtration\ conditions\ in\ Q_D\ with\ text\ parameters \quad (16)$$

$$D_{L|i} = dictionary\ length\ for\ column\ i$$
$$P_{DICT}(D_L) = 0.0138 \cdot 10^{-6} \cdot D_{L|i} \quad (17)$$

The estimation of the upper bound time of text translation for query Q_D is:

$$\lceil T_{TRANS} \rceil = \sum_{i \in CDT_Q} P_{DICT}(D_{L|i}) \quad (18)$$

G. Scheduling Algorithm with Support for Multicore CPUs and Text-to-Integer Translation

In [16] we have proposed a scheduling algorithm that divides the GPU into several partitions. An example of a GPU partitioning is shown in Figure 7. The old system had a single CPU partition that was using only one core of multicore CPU. This had a significant impact on the performance. In addition to this performance limitation our original implementation was not able to work with literal data on the GPU side. The new proposed version of our system addresses both of these limitations, therefore a new scheduling algorithm that supports these new features has to be developed as well.

All GPU partitions are used to answer queries using the relational database tables stored in GPU memory. All partitions have access to the entire GPU memory and to all fact tables. Therefore any partition can answer any query. The CPU partition that is dedicated for query processing has access to all OLAP cubes that are pre-calculated and stored in the main memory and answers the queries using these cubes.

The new scheduling algorithm supports the new parallel implementation of OLAP cube processing and text-to-integer translation by dividing the CPU into two virtual partitions: the processing partition and the preprocessing or translation partition. The translation is necessary for queries that are processed by the GPU. Therefore every query scheduled for the GPU must be preprocessed by CPU first. This preprocessing is initiated by the scheduler and uses the preprocessing CPU partition for the translation. The preprocessing partition returns translated queries back to the scheduler that forwards the queries to selected GPU partition.

The scheduling algorithm is designed to schedule the tasks (here by task we mean query processing) between the CPU and the GPU so that every task meets the time constraints. The time constraint is a relative time, as the perceived response time is an essential property of OLAP systems. For each query submitted to the system the scheduling algorithm must estimate the CPU processing time T_{CPU}, the GPU processing time T_{GPU} for different partitions

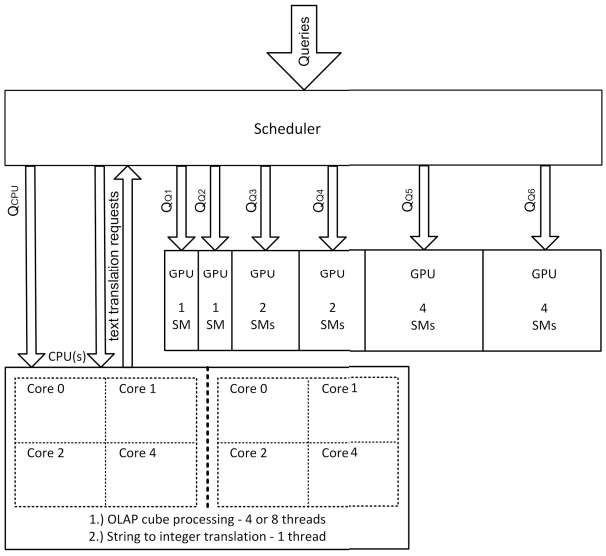

Figure 7. The block diagram of the GPU and CPU paritions. GPU has six different partitions all are used to process the queries from fact tables. CPU has two partitoins one for OLAP cube processing and one for text to integer translation.

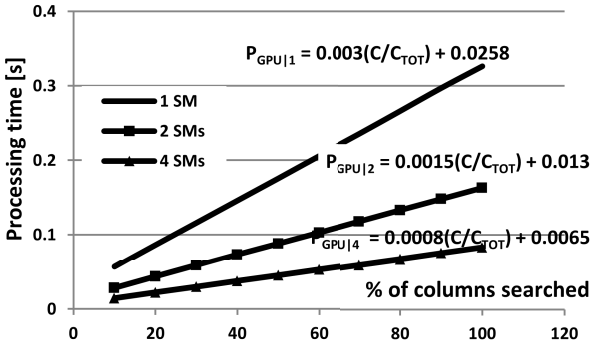

Figure 8. Tesla C2070 performance for query processing for 1, 2 and 4 SMs and for different number of searched columns. The table size in GPU memory is 4 GB.

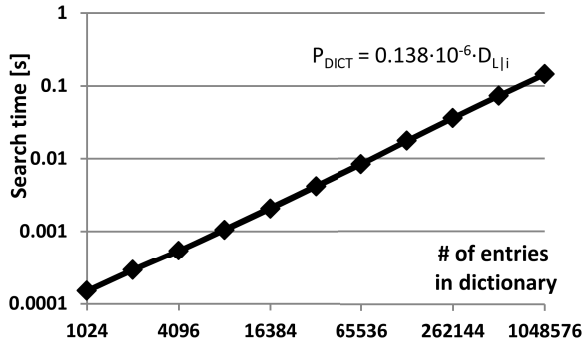

Figure 9. Dictionary search performance function for different sizes of dictionaries.

and the translation time T_{TRANS}. This estimation is based on methods presented in Sections III-B, III-E and III-F. The

system performance variables $P_{GPU}\left(\frac{c_{Q_D}}{c_{TOTAL}}, n_B\right)$ for GPU processing, $P_{CPU}(SC_{size})$ for CPU processing and $P_{DICT}(D_L)$ for text-to-integer are measured by benchmarks and stored inside the scheduler. The performance estimation functions of our test systems are shown in (7) and (10) for CPU partition, (14) for GPU partitions and for text translation.

The scheduler divides the GPU into 6 partitions: 2 partitions have 1 SM each, 2 partitions have 2 SMs each, and last two partitions have 4 SMs each. This functional partitioning has been optimized for the Tesla C2070 GPU with its 14 SM units. Each partition has its own queue into which the scheduler submits queries for processing. The CPU is divided into two virtual partitions where each of them has its own queue. The first partition, used for OLAP cube processing, has a queue called Q_{CPU}. The second partition, used for string to integer translation, has a queue called Q_{TRANS}.

Each queue is aware of how many jobs are outstanding and when all its jobs will be finished. This parameter is $T_{Q|C}$ for CPU OLAP queue, $T_{Q|TRANS}$ for CPU translation queue and $T_{Q|G1}$, $T_{Q|G2}$, …, $T_{Q|G6}$ for GPU queues.

When the scheduling algorithm receives a query Q at time T_Q it examines the query and estimates the CPU processing time T_{CPU} using (7) or (10). In the case of the GPU, three different processing times T_{GPU1}, T_{GPU2} and T_{GPU3} need to be estimated for partitions with 1, 2 and 4 SMs. The estimate functions are listed in (14). In addition a translation time T_{TRANS} is also estimated using (17).

Every query has to be answered within a predefined time period T_C after submitted to the system. This deadline time T_D for query Q is calculated as $T_D = T_Q + T_C$.

In step 3, Figure 10, response times for all system partitions are estimated. The response time of a partition is defined as a time when all queries submitted to this partition are processed plus the processing time of new query Q. In other words the response time tells the scheduler when the partition returns the answer of the query Q to the user. The response time includes the query processing time and text-to-integer translation time if query requires pre-processing of text columns.

In step 4, the scheduler identifies all partitions that can deliver the result before deadline and creates a set P_{BD} which contains these partitions. If P_{BD} is not empty then system is able to process the query in time and scheduler goes to step 5.

In step 5, if the P_{BD} set contains a CPU partition and this partition can process the query Q faster than the fastest GPU partition (partition with 4 SMs) then the query is submitted to Q_{CPU} queue. Every time the query is submitted to a queue, a value of parameter T_Q is updated. In this case of the CPU, the partition $T_{Q|C}$ is updated to $T_{Q|C} = T_{Q|C} + T_{CPU}$.

In the case that the CPU partition is slower than the GPU partition or it is not in the P_{BD} set, the query is scheduled to one of the GPU queues Q_{G1} - Q_{G5}. The scheduler algorithm goes from the slow queues Q_{G1} and Q_{G2} to the fast queues Q_{G4} and Q_{G5} and selects the slowest queue that is in the set P_{BD}. The query is submitted to the selected queue and $T_{Q|Gi}$ parameter is updated to $T_{Q|Gi} = T_{Q|Gi} + T_{GPUj}$. In addition, if

Maximum processing time per query - T_C – system parameter

1.) New query Q submitted at time T_Q
- Deadline to finish processing of query Q: $T_D = T_Q + T_C$

2.) Estimate processing times of Q for all system partitions using the performance model:
- CPU OLAP cube processing time: T_{CPU}
- GPU processing times: T_{GPU1}, T_{GPU2}, T_{GPU3}
- text to integer translation time: T_{TRANS}

3.) Estimate the response time T_R for query Q of all system partitions (for GPU partitions the translation time must be included in the estimation):
- $T_{R|CPU} = T_{Q|C} + T_{CPU}$
- $T_{R|GPUi} = \begin{cases} \max(T_{Q|Gi},\ T_{Q|TRANS} + T_{TRANS}) + T_{GPUi} & \text{- with translation} \\ T_{Q|Gi} + T_{GPUj} & \text{- without translation} \end{cases}$
  for $i = 1,2,..,6$, where $j = \lceil i/2 \rceil$

4.) Find all system partitions that answers the query Q before deadline
- find set of all partitions where $(T_D - T_R) > 0$ and save it as a set P_{BD}

5.) **IF** (set P_{BD} is not empty) **THEN**
IF (CPU partition is in the set P_{BD}) **AND** (CPU processing time T_{CPU} of query Q is shorter than processing time of the fastest GPU partition T_{GPU3}) **THEN**
- submit query Q to queue Q_{CPU}
- update $T_{Q|C} = T_{Q|C} + T_{CPU}$

ELSE
- find slowest GPU partition in set P_{BD} that can answer the query and submit the query to its queue:

FOR ($i = 0$, $i < 6$, i++)
IF (partition i is in set P_{BD}) {
IF translation is necessary {
- submit query Q to translation queue Q_{TRANS}
- update $T_{Q|TRANS} = T_{Q|TRANS} + T_{TRANS}$ }
- submit query Q to queue Q_{Gi}
- update $T_{Q|Gi} = T_{Q|Gi} + T_{GPUj}$ where $j = \lceil i/2 \rceil$
- break the FOR loop }

6.) **IF** (set P_{BD} is empty) **THEN** - find system partition with the fastest response time and submit the query Q into this partition
- find index i of min($|T_D - T_{R|CPU}|$, $|T_D - T_{R|GPU1}|$, … , $|T_D - T_{R|GPU6}|$)
IF translation is necessary {
- submit query Q to translation queue Q_{TRANS}
- update $T_{Q|TRANS} = T_{Q|TRANS} + T_{TRANS}$ }
- submit query Q to the selected partition queue
- update the $T_{Q|i}$ time parameter of the selected queue
- **IF** i points to CPU partition: $T_{Q|C} = T_{Q|C} + T_{CPU}$
- **IF** i points to GPU partition: $T_{Q|Gi} = T_{Q|Gi} + T_{GPUj}$, $j = \lceil i/2 \rceil$

T_{CPU} – estimated CPU processing time of Q,
T_{GPU1}, T_{GPU2}, T_{GPU3} – estimated GPU processing time of Q for different number of SM (1 for queues Q_{G1}, Q_{G2}; 2 for Q_{G3}, Q_{G4}; and 4 for Q_{G5}, Q_{G6}),
T_{TRANS} – estimated text to string translation time,

Q_{CPU} – queue for CPU processing,
$Q_{G1} – Q_{G6}$ – queues for GPU processing,
Q_{TRANS} – queue for text to string translation,

$T_{Q|C}$ – time when CPU finishes processing of all submitted queries in queue Q_{CPU},
$T_{Q|G1\text{-}6}$ – time when GPU partition 1-6 finishes processing of all queries submitted to queue their respective queues,
$T_{Q|TRANS}$ – time when translation partition finishes transition of all queries that requires translation

T_R – response time - time when selected partition finish processing of all queries in its queue plus processing time of new query Q
$T_{R|CPU}$ – response time of CPU partition
$T_{R|GPU1\text{-}6}$ – response time of GPU partitions 1 to 6,

P_{BD} – set of partitions that can answer the query before deadline,

Figure 10. Scheduling algorithm for CPU and GPU co-scheduling for OLAP queries with text to string translatin support.

translation is required the request is submitted to Q_{TRANS} and $T_{Q|TRANS}$ is updated to $T_{Q|TRANS} = T_{Q|TRANS} + T_{TRANS}$.

For the GPU scheduling, the main strategy is to task the slower queues first so that GPU has resources available for the computationally expensive queries that might be submitted later.

If system is not able to process the query in time (P_{BD} is empty) then the scheduler goes to step 6 and searches for the partition with the shortest response time T_R. So even though the system will not meet the time constraints, it will try to deliver the answer as soon as possible.

The real processing time of query Q is also measured by the system. When the query processing is finished, the real processing time is compared with estimated processing time. The difference of these two times then used to update the value T_Q of the queue that was processing the query. This way the errors in the estimation do not significantly affect the scheduling algorithm.

IV. RESULTS

To test the efficiency of the proposed hybrid OLAP solution with CPU and GPU co-scheduling we have developed a system model. The setup of the model is done based on characteristics extracted from performance measurements. These are the sample characteristics that describe the performance of our test system with two Intel Xeon X5667 CPUs and Tesla C2070 GPU accelerator:

Performance functions for parallel OpenMP implementation (running 4 and 8 OpenMP threads) of OLAP cube processing is:

$$T_{CPU|4T} = \begin{cases} 0.0001 \cdot {SC_{size}}^{0.9341} \; if \; SC_{size} < 512\ MB \\ 5 \cdot 10^{-5} \cdot SC_{size} \; + \; 0.0096 \; if \; SC_{size} > 512\ MB \end{cases}$$

$$T_{CPU|8T} = \begin{cases} 6 \cdot 10^{-5} \cdot {SC_{size}}^{0.984} \; if \; SC_{size} < 512\ MB \\ 4 \cdot 10^{-5} \cdot SC_{size} \; + \; 0.0146 \; if \; SC_{size} > 512\ MB \end{cases}$$

Performance function for table processing by Tesla C2070 is:

$$P_{GPU|1SM} = \; 0.003\left({C}/{C_{TOT}}\right) + 0.0258,$$

$$P_{GPU|2SM} = \; 0.0015\left({C}/{C_{TOT}}\right) + 0.013,$$

$$P_{GPU|4SM} = \; 0.0008\left({C}/{C_{TOT}}\right) + 0.0065.$$

Performance function for single threaded text to integer translation is:

$$P_{DICT}(D_L) = \; 0.0138 \cdot 10^{-6} \cdot D_{L|i} \; .$$

The model used for evaluation of the system has the following configuration: the GPU has fact table of size ~4GB which contains 3 dimensions, 4 levels in each dimension; the CPU has 4 pre-calculated OLAP cubes of sizes ~32 GB, ~500MB, ~500KB and ~4KB that are used for CPU aggregation.

The performance of the system is evaluated in terms of queries processed per second. The total number of processed queries that meet the time constraints is recorded as well as number of queries that did not.

Our previous single threaded CPU implementation of OLAP cube search [16] was able to achieve a processing rate ~ 12 queries per second if only ~500MB, ~500KB and ~4KB OLAP cubes were used for processing. This was a significant limitation of our system. By parallelizing our code using OpenMP we were able to achieve a significantly higher processing rate on the CPU side. The CPU processing rate have increased from 12 queries per second to 87 and 110 queries per second for 4 and 8 OpenMP threads, respectively. These results are shown in Figure 3.

In addition, the CPU partition itself is now able to process a set of OLAP cubes of sizes ~32 GB, ~500MB, ~500KB and ~4KB at rate 11 queries per second. Previously we had to use GPU partitions to be able to work with this large database.

We have also measured the performance utilizing the GPU accelerator Tesla C2070 only with disabled CPU processing. In this case the system processing rate is ~64 queries per second. Our previous implementation without text-to-integer translation achieved higher processing rate ~69 per second, therefore we can observe that the translation typically slows down the system by approximately 7%. We want to minimize this effect by using more sophisticated translation algorithm in our future implementation.

To see the advantage of fast parallel implementation of CPU based OLAP cube processing as a part heterogeneous system with advanced scheduling that supports text-to-integer translation we have measured the performance of the entire system. Even though the translation slows down the GPU processing by 7% the entire system is more than 2.3 times faster. The query processing rate has increased from 102 queries per second (sequential CPU implementation) to 206 or 228 queries per second for OpenMP implementation running 4 or 8 threads, respectively.

TABLE 1. PROCESSING RATE OF CPU BASED OLAP CUBE PROCESSING FOR SET OF CUBES OF SIZES ~500MB, ~500KB AND ~4KB

Implementation type	Sequential	**Parallel - OpenMP**	
Number of threads [-]	1	**4**	**8**
Processing rate [Q/s]	12	**87**	**110**

TABLE 2. PROCESSING RATE OF CPU BASED OLAP CUBE PROCESSING FOR SET OF CUBES OF SIZES ~32 GB, ~500MB, ~500KB AND ~4KB

Implementation type	**Parallel - OpenMP**	
Number of threads [-]	**4**	**8**
Processing rate [Q/s]	**9**	**11**

TABLE 3. PROCESSING RATE OF GPU ACCELERATED OLAP SYSTEM FOR SET OF CUBES OF SIZES ~ 32GB, ~500MB, ~500KB AND ~4KB

Implementation type	Sequential	**Parallel - OpenMP**	
Number of threads [-]	1	**4**	**8**
Processing rate [Q/s]	102	**206**	**228**

V. CONCLUSIONS

Our previous work presented in [16] had proposed a task-scheduling algorithm for GPU acceleration of OLAP systems. The usage of our proposed system reduces the memory requirements for the CPU memory. Since large cubes, which are costly to store and aggregate, do not have to be pre-computed. The proposed task scheduler will automatically forward these jobs to GPU.

Two main drawbacks of original solution had been identified. The first limitation dealt with performance on the CPU side of the system because only a single threaded implementation of OLAP cube processing was implemented. Therefore in this paper we present a parallel OpenMP version. The performance of the new version was measured and a new performance model was created. This performance model was implemented into the scheduling algorithm. The performance measurements show that processing rate of CPU partition was improved from 12 to 110 queries per second when compared to original implementation. In addition, the CPU partition is now able to process OLAP cubes of sizes up to 32 GB at rate 11 queries per second. The total performance of entire hybrid system (CPU + GPU) was increased from 102 to 228 queries per second.

The second limitation of the original solution is absence of text support on the GPU side. Therefore previously we were not able to work with the tables that contain string columns. This problem was now solved by text-to-integer translation so every text column is converted to integer column using dictionaries. The translation is unique to the GPU side of the system and therefore new scheduling algorithm is proposed. This algorithm estimates the translation time for every query that requires translation and uses this time for more effective scheduling. The translation slows down the GPU processing by 7% when compared to original version without translation, but adds an important functionality. In our future work we minimize this effect by using advanced translation mechanism.

REFERENCES

[1] K. Beyer ; R. Ramakrishnan ; Bottom-up Computation of Sparse and Iceberg Cubes. *Proceedings of the 1999 ACM SIGMOD Conference,* pages 359-370, 1999.

[2] T. Braun; H. Siegel; N. Beck; L. Boloni; M. Maheswaran; A. Reuther; J. Robertson; M. Theys; B. Yao; D, Hensgen; R. Freund; A Comparison Study of Static Mapping Heuristics for a Class of Meta-Tasks on Heterogeneous Computing Systems, *Proceedings of the Heterogeneous Computing Workshop*, April 1999 Page(s):15 – 29

[3] F. Dehne, T. Eavis, A. Rau-Chaplin Top-Down Computation of Partial ROLAP Data Cubes. *Proceedings of the 37th Hawaii International Conference on System Sciences.* Big Island Hawaii, 2004

[4] R.P. Dick; N. K. Jha, MOGAC: a Multiobjective Genetic Algorithm for Hardware/Software Co-Synthesis of Distributed Embedded Systems, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Volume 17, Issue 10, Oct. 1998 Page(s):920 - 935

[5] J. Gray, A. Bosworth, A. Layman and H. Prahesh. Data Cube: a Relational Aggregation Operator Generalizing Group-by, Cross-Tab, and Sub-Total. Microsoft Technical Report, MSR-TR-95-22, 1995.

[6] N.K. Govindaraju, B. Lloyd, W. Wang, M. Lin, D. Manocha. Fast Computation of Database Operations using Graphics Processors. *SIGMOD 2004: Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*. pages 215-226, Paris, France, 2004.

[7] H. Junwei; W. Wolf; Process Partitioning for Distributed Embedded Systems, *Proceedings* of Fourth International Workshop on *Hardware/Software Co-Design, 1996. (Codes/CASHE '96),* page(s): 70-76, Mar 1996

[8] K. Kaczmarski, Comparing GPU and CPU in OLAP Cube Creation. *Conference on Current Trends in Theory and Practice of Informatics. SOFSEM 2011*, pages 308-319, Novy Smokovec, Slovakia, 2011.

[9] T. Lauer, A. Datta, Z. Khadikov, C. Anselm, Exploring Graphics Processing Units as Parallel Coprocessors for Online Aggregation. *Proceedings of the ACM 13th International workshop on Data warehousing and OLAP* Toronto, Ontario, 2010.

[10] W. Liang, M. Orlowska, M. Computing Multidimensional Aggregates in Parallel. *Proceedings of Parallel and Distributed System*, pages 92-99, Tainan, Taiwan, 1998.

[11] H. Oh, S. Ha; A Hardware-Software Co-Synthesis Technique Based on Heterogeneous Multiprocessor Scheduling, *Proceedings of the Seventh International Workshop on Hardware/Software Co-Design*, 1999, May 1999 Page(s):183 – 187

[12] S. Prakash ; C. Parker ; A Design Method for Optimal Synthesis of Application-Specific Heterogeneous Multiprocessor Systems", *Proceedings on Heterogeneous Processing* Beverly Hills, California. 1992

[13] S. Sarawagi ; M. Stonebraker ; Efficient Ogranization of Large Multidimensional Arrays. *Proceedings of the Tenth International Conference on Data Engineering.* pages 328-336, Washington DC, 1994

[14] S. Sarawagi ; R. Agrawal ; A. Gupta ; On Computing the Data Cube. Technical Report RJ10026, IBM Almaden Research Center, San Jose, California, 1996

[15] H. Siegel; S. Ali; Techniques for mapping tasks to machines in heterogeneous computing systems, *Journal of Systems Architecture Volume 46* Page(s): 627-639.

[16] Lubomir Riha, Colin Shea, Maria Malik and Tarek El-Ghazawi, "Task Scheduling for GPU Accelerated OLAP Systems" *in Proc. of CASCON 2011*, Toronto, Canada.

[17] "Tesla C2050/C2070 GPU Computing Processor." NVIDIA, Santa Clara, CA

[18] T. Yen ; W. Wolf ; Sensitivity-Drive Co-Synthesis of Distributed Embedded Systems. *Proceedings of the Eighth International Symposium on System Synthesis*. pages 4-9, Cannes, France. 1995

[19] Whitepaper: NVIDIA's Next Generation CUDA Compute Architecture: Fermi. *NVIDIAFermiComputeArchitectureWhitepaper.pdf,* NVIDIA Corporation, 2009

[20] A Y. Zhao, P. M. Deshpande and J. F. Naughton. An Array-Based Algorithm for Simultaneous Multi-Dimensional Aggregates. *Proceedings of the 1997 ACM- SIGMOD Conference*, pages 159-170, Tucson, Arizona, 1997.

[21] G. S. Brodal, R. Fagerberg. Cache-Oblivious String Dictionaries. *SODA '06 Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm, 2006.*

[22] A. V. Aho, M.J. Corasick. Efficient String Matching: an Aid to Bibliographic Search. *Communications of the ACM, Volume 18, Issue 6*, June 1975.

[23] D.P. Scarpazza, O. Villa, F. Petrini. *CF '08 Proceedings of the 5th conference on Computing frontiers.* 2008