Parallel H-Tree Based Data Cubing on Graphics Processors

Baoyuan Wang * Yizhou Yu †

Abstract

Graphics processing units (GPUs) have a SIMD architecture and have been widely used recently as powerful general-purpose co-processors for the CPU. In this paper, we investigate efficient GPU-based data cubing because the most frequent operation in data cube computation is aggregation, which is an expensive operation well suited for SIMD parallel processors. H-tree is a hyper-linked tree structure used in both top-k H-cubing [21] and the stream cube [20]. Fast H-tree construction, update and real-time query response are crucial in many OLAP applications. We design highly efficient GPU-based parallel algorithms for these H-tree based data cube operations. This has been made possible by taking effective methods, such as parallel primitives for segmented data and efficient memory access patterns, to achieve load balance on the GPU while hiding memory access latency. As a result, our GPU algorithms can often achieve more than an order of magnitude speedup when compared with their sequential counterparts on a single CPU. To the best of our knowledge, this is the first attempt to develop parallel data cubing algorithms on graphics processors.

1 Introduction

Graphics processing units (GPUs) are routine components of personal computers and were traditionally designed for displaying visual information on the computer screen. Over the years, the rapid progress of modern GPU architectural design has significantly shifted their role in computing. GPUs have been widely used recently as powerful general-purpose co-processors for the CPU. Similar to CPUs, in particular multi-core CPUs, GPUs consist of multiple processors. However, GPUs provide parallel lower-clocked execution capabilities on over a hundred SIMD (Single Instruction Multiple Data) processors whereas current multi-core CPUs typically offer out-of-order execution capabilities on a much smaller number of cores. Moreover, the majority of GPU transistors are devoted to computation units rather than caches, and GPU cahe size are an order of magnitude smaller than CPU cache sizes. These hardware design choices make GPUs particularly well suited for high-bandwidth computationally intensive tasks with relatively simple logic.

Data cubes are a common data mining technique for abstracting and summarizing relational databases [18]. Cuboids in a data cube store preprocessing results that enable efficient on-line analytical processing (OLAP) [8]. Computing data cubes is a time-consuming and computationally intensive process. Parallel algorithms [30, 10, 9, 12] have been developed for systems with multiple CPUs. We investigate efficient GPU-based data cubing algorithms in this paper because the most frequent operation in data cube computation is aggregation, which is an expensive operation well suited for SIMD parallel processors.

For high-dimensional datasets, a fully materialized data cube may be several hundred times larger than the original data set. It is thus only practical to precompute a subset of the cuboids. Partial data cubes are also very useful for real-time streaming data as computing and updating a full data cube would take too long a time to keep pace with the arrival rate of a real-time data stream, and therefore, preclude the possibility of real-time data analysis. H-tree [21, 20] is an efficient data structure for computing partially materialized data cubes as well as for incremental and online cubing of streaming data. It is also closely related to efficient mining of frequent patterns [22]. In this paper, we focus on the development of H-tree based data cubing algorithms on the GPU. Specifically, we design highly efficient GPU-based parallel algorithms for H-tree construction, incremental H-tree update as well as online H-tree based data cube query processing. These GPU algorithms can achieve more than an order of magnitude speedup when compared with their sequential counterparts on a single CPU.

Although there has been much recent success on GPU-based data structures, including k-d trees [40], developing GPU-based parallel algorithms for the H-tree data structure still imposes a great challenge. The primary reason lies in the fact that unlike binary trees and k-d trees, different H-tree nodes may have drastically different number of children nodes, which makes it very hard to achieve load balance among different threads and thread blocks. We adopt three important strategies to overcome this difficulty. First, use parallel primitives designed for segmented data. By using a flag vector to indicate distinct data segments each of which may correspond to a different parent node, data across an entire level of an H-tree can be processed in parallel. For this purpose, we have developed the first segmented sorting algorithm for GPUs. It is based on a parallel algorithm for radix sort. Second, adaptively divide the workload from each data segment among multiple thread blocks. Since the workload from different data segments may differ significantly, to achieve load balance, each data segment needs to be divided into a variable number of chunks which may be assigned to different thread blocks. Third, use scatter and gather primitives as well as coalesced memory access

*Baoyuan Wang is with College of Computer Science and Technology, Zhejiang University, Hangzhou, China. email: zjuwby@gmail.com

†Yizhou Yu is with Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL 61801. email: yizhouy@acm.org

to efficiently redistribute data in parallel. This is necessary because data vectors often need to be reorganized to expose parallelism at different stages of an algorithm.

In summary, this paper has the following three contributions. First, we identify technical challenges in performing parallel H-tree based data cubing tasks on GPUs and provide general solutions to address these challenges. Our GPU-based data-parallel primitives are applicable to not only H-tree operations but also other GPU-based parallel data structures. Second, we design and implement GPU-based parallel algorithms for representative H-tree operations, and empirically evaluate these algorithms in comparison with optimized versions of their CPU counterparts. To the best of our knowledge, this is the first attempt to develop data cubing algorithms on graphics processors and the first attempt to develop parallel algorithms for H-tree operations. Third, we provide insights and suggestions on GPU programming for the tasks considered.

The remainder of this paper is organized as follows. In Section 2, we briefly introduce the GPU architecture and programming model, and review GPU-based algorithms for database operations as well as parallel data cubing algorithms on multiple CPUs. In Section 3, we describe a few parallel primitives that serve as building blocks for our algorithms for H-tree based operations. In particular, we introduce three new primitives and their implementations. We describe the H-tree structure and its GPU implementation in Section 4. In Section 5, we present our GPU algorithms for several H-tree based data-cubing tasks. We experimentally evaluate our algorithms in Section 6 and provide insights and suggestions on GPU programming as well.

2 Background and Related Work

2.1 Graphics Processors (GPUs)

In the following, we briefly introduce the architecture and programming model for both AMD and NVidia GPUs with an emphasis on NVidia G80 GPUs.

At a high level, the GPU consists of many SIMD multi-processors each of which has multiple scalar processors. At any given clock cycle, each processor of a multiprocessor executes the same instruction, but operates on different data. The GPU has a large amount of device memory, which has high bandwidth and high access latency. For example, the G80 GPU has an access latency of 400-600 cycles and a memory bandwidth of 84.6 GB/second. In addition, each multiprocessor usually has a fast on-chip local memory, which is shared by all the processors in a multi-processor. The size of this local memory is small but the access latency is low.

GPU threads are different from CPU threads in that they have low context-switch cost and low creation time as compared to their CPU counterparts. Threads are organized into thread groups, and the threads within the same group are always assigned to the same multi-processor. Threads within a thread group share computational resources such as registers on a multi-processor. They can also communicate through the fast local shared memory of the multi-processor they are assigned to. Moreover, when multiple threads in a thread group access consecutive memory addresses, these memory accesses are grouped into one access. Multi-processors allow a large number of active threads to hide the high memory access latency. While some threads are waiting for the data, the others can execute instructions. This further implies that each thread group needs to have a reasonably large number of threads. For example, each G80 multi-processor has eight scalar processors and the suggested minimum number of threads per group is 32.

GPU programming languages include graphics APIs such as OpenGL [2] and DirectX [6], and GPGPU languages such as NVidia CUDA [3], AMD CTM [1]. With these APIs, programmers write two kinds of code, the kernel code and the host code. The host code runs on the CPU to control the data transfer between the GPU and the system memory, and to start kernels on the GPU. The kernel code is executed in parallel on the GPU. In the kernel code, a computational task is divided among a number of thread groups, which are further dynamically scheduled among the multi-processors. Typically, multiple thread groups can be simultaneously scheduled on the same multi-processor.

2.2 GPU-Based Database Operations

Recently, GPUs have been used to accelerate scientific, geometric, database and imaging applications. For an overview on the state-of-the-art GPGPU techniques, we refer the reader to the recent survey by Owens et al. [31]. In the following, we focus on the techniques that use GPUs to improve the performance of database and data mining operations. Sun et al. [38] used the rendering and search capabilities of GPUs for spatial selection and join operations. Bandi et al. [5] implemented GPU-based spatial operations as external procedures to a commercial DBMS. Govindaraju et al. pioneered GPU-based algorithms for relational database operators including join [25], selections, aggregations [15] as well as sorting [14], and for data mining operations such as computing frequencies and quantiles for data streams [16]. In contrast, our algorithms are designed for data cubing [18] and online analytic processing [8] instead of regular database query processing. A few primitives supporting data-parallel programming have been developed on a similar GPU architecture our algorithms are based on. Sengupta et al. implemented GPU-based prefix sum [37] and segmented scan [36]. Popov *et al.* [32] implemented reduction and Zhou et al. [40] implemented segmented reduction on the GPU. He et al. [24] proposed a multi-pass scheme to improve the scatter and gather operations on the GPU. Our algorithms utilize these operations as primitives. Recently, Lieberman et al. [29] developed a similarity join, and Zhou et al. [40] developed an algorithm for kd-tree construction both using CUDA.

2.3 Parallel Data Cubes

Data cubes are a common data mining technique for abstracting and summarizing relational databases [17, 18]. Cuboids in

a data cube store preprocessing results that enable efficient online analytical processing (OLAP) [8, 34]. Since computing data cubes is a time-consuming process, parallel algorithms have been developed for systems with multiple CPUs. Such systems may or may not have a shared file system. When there exists a shared file system, corresponding parallel algorithms focus on evenly distributing computational load among the multiple CPUs. The methods in [30, 10] reduce communication overhead by partitioning the load into coarse-grained tasks and assigning sets of groupby computations to individual processors. When working with PC clusters without a shared file system, in addition to load balancing, minimizing data redistribution cost also becomes critical. The method in [13] is based on a spatial merge between different sub-cubes distributed over different machines. The spatial merge operation can be reduced to a parallel prefix. Effective data partitioning methods for data cubes represented as relational databases have been studied in [9, 12]. Optimal communication and memory schemes for parallel data cube construction have been presented in [27]. To the best of our knowledge, there has not been any previous work on GPU-based parallel algorithms for data cube construction and query operations.

For high-dimensional datasets, a fully materialized datacube may be several hundred times larger than the original data set. It is thus only practical to precompute a subset of the cuboids. A parallel algorithm has been developed in [11] to compute partial data cubes on multiple CPUs. However, there has not been any previous work on parallel algorithms for H-tree based partial materialization.

3 PRIMITIVES

In this section, we present parallel primitives frequently used by our GPU-based parallel algorithms. For those developed in previous work, we only provide their definitions.

3.1 Existing Primitives

Gather and Scatter Gather and scatter are parallel memory access primitives for modern GPUs. We adopt their definitions and implementations from [24]. Scatter outputs each element in an array, R_{in}, to its corresponding location in another array, R_{out}.

Primitive: scatter(R_{in}, L, R_{out})
Input: $R_{in}[1, ..., n], L[1, ..., n]$
Output:$R_{out}[1, ..., n]$
Function:$R_{out}[L[i]] = R_{in}[i], i = 1, ..., n$

Gather does the opposite of scatter.

Primitive: gather(R_{in}, L, R_{out})
Input:$R_{in}[1, ..., n], L[1, ..., n]$
Output:$R_{out}[1, ..., n]$
Function:$R_{out}[i] = R_{in}[L[i]], i = 1, ..., n$

Compact Compact only scatters the elements whose flag has been set to 1 according to a flag vector. It is originally from [Harris et al.2007].

Primitive: compact(R_{in}, F, L, R_{out})
Input: $R_{in}[1, ..., n]$,$F[1, ..., n]$,$L[1, ..., n]$
Output: $R_{out}[1, ..., m]$
Function: if $F[i] = 1$, $R_{out}[L[i]] = R_{in}[i], i = 1, ..., n$

Reduction and Segmented Reduction Summing up or finding the maximum (minimum) element of an array is a typical reduction operation. Popov *et al.* [32] was the first to use it on the GPU.

Primitive: reduction(R_{in}, $\bigoplus$, R_{out})
Input:$R_{in}[1, ..., n]$,binary operator $\bigoplus$
Output:R_{out}
Function:$R_{out} = \bigoplus R_{in}[i], i = 1, ..., n$

Gropp *et al.* [19] performs segmented reduction on arbitrary segments of the input vector by using a flag vector. [40] was the first one to perform segmented reduction in a thread block on GPUs. Our implementation extends its capability and performs segmented reduction across multiple blocks using recursion. We found that segmented reduction was perfectly suited for parallel aggregation (roll-up) operations.

Primitive: segmented_reduction(R_{in}, segID, $\bigoplus$, R_{out})
Input: $R_{in}[1, ..., n]$, $segID[1, ..., n]$, binary operator $\bigoplus$
Output: $R_{out}[1, ..., m]$
Function: $R_{out}[i] = \bigoplus R_{in}[j]$, where$segID[j] = i, i = 1, ..., m$

Scan and Segmented Scan Scan was first used on GPUs by Horn [26], then extended and carefully optimized by Sengupta *et al.* [37] using CUDA. One typical example of scan is prefix sum, with each output element computed as the sum of its preceding elements in an array. Note that there are two types of scan, excluded scan and included scan. Included scan should include the element at the current position itself. We use the prefix sum implementation from the CUDPP library [23].

Primitive: scan(R_{in}, $\bigoplus$, R_{out})
Input:$R_{in}[1, ..., n]$, binary operator $\bigoplus$
Output:$R_{out}[1, ..., n]$
Function: $R_{out}[i] = \bigoplus_{j<i} R_{in}[j], i = 1, ..., n$

Segmented scan primitives was first introduced in [35]. It permits parallel scans on an arbitrary "segment" of the input vector by using a flag vector to demarcate each segment. Sengupta *et al.* [36] is the first to introduce segmented scan on GPUs using CUDA API.

Primitive: segmented_scan(R_{in}, segID, $\bigoplus$, R_{out})
Input: $R_{in}[1, ..., n]$, $segID[1, ..., n]$, binary operator $\bigoplus$
Output: $R_{out}[1, ..., n]$
Function: $R_{out}[i] = \bigoplus_{j<i, segID[j]=segID[i]} R_{in}[j], i = 1, ..., n$

Sort Sort is a key operation in our H-tree construction algorithm. Much work has been performed on various GPU-based sorting algorithms by other researchers. This includes bitonic sort [36], radix sort [36] and quicksort [36, 7, 25]. In particular, [25] provides a quicksort implementation based on radix sort. It can handle additional data types, such as strings. Note that it is sufficient to represent the value of a database dimension using an integer or a string. Since we only need to sort values of dimensions in this paper, our sorting primitives, including radix sort and segmented (radix) sort (Section 3.2), only support these two data types. Our radix sort is based on [25].

Primitive: radix_sort(R_{in}, pos)
Input: $R_{in}[1, ..., n]$
Output: sorted $R_{in}[1, ..., n]$ and $pos[1, ..., n]$
Function: histogram based radix sort, pos stores the new position for each element.

3.2 Our Primitives

In addition to the above primitives, we have developed three new primitives needed by our GPU algorithms.

3.2.1 Split and Segmented Split

The split primitive we develop here is completely different from previously developed versions used by [36, 25]. The difference is that their split is mainly used for sorting while our split and segmented split primitives just demarcate the spans of a vector, where a span consists of a few consecutive elements that are identical to each other. Split and segmented split set the first element of each span to 1 and leave the rest to 0. These two primitives are primarily used to obtain the flag vector for segmented scan, reduction and sort.

Primitive: split(R_{in}, R_{out})
Input: $R_{in}[1, \ldots, n]$
Output: $R_{out}[1, \ldots, n]$
begin
  for each element i in **parallel**
    if $R_{in}[i] = R_{in}[i-1]$ **then**
      $R_{out}[i] \leftarrow 0$
    else
      $R_{out}[i] \leftarrow 1$
end

Primitive: segmented_split(R_{in}, $segID$, R_{out})
Input: $R_{in}[1, \ldots, n]$, $segID[1, \ldots, n]$
Output:$R_{out}[1, \ldots, n]$
begin
  for each element i in **parallel**
    if $segID[i] = segID[i-1]$ **then**
      if $R_{in}[i] = R_{in}[i-1]$ **then**
        $R_{out}[i] \leftarrow 0$
      else
        $R_{out}[i] \leftarrow 1$
    else
      $R_{out}[i] \leftarrow 1$
end

3.2.2 Segmented Sort

Our segmented sort is histogram-based. It is inspired by the parallel radix sort algorithm proposed by [39] and [25]. However there are two major differences. The first is that segmented sort performs sorting on a number of segments in parallel while the aforementioned work can be considered a special example in that it performs on one large segment only. The second difference is that each processor (thread) can own more than one histograms if its assigned data belongs to multiple segments.

Primitive: segmented_sort(R_{in}, $segID$, pos, R_{out})
Input: $R_{in}[1, ..., n]$, $segID[1, ..., n]$
Output: $pos[1, ..., n]$, $R_{out}[1, ..., n]$
Variables
T, the total number of threads
$P = n/T$, the number of elements assigned to each thread
$HNum$, 1D array holding the number of histograms per segment
$preHis$, 1D array for the results of prefix sum on $HNum$
H, the total number of histograms
Key, the number of bins in a histogram
L, a large 1D array holding all the histograms and scatter locations
begin
(*Phase 1. Histogram allocation*)
1. for each segment i in **parallel**
2. $HNum[i]$ = # of threads that fully or partially process the i-th segment
3. reduction($HNum$, $+$, H)
4. scan($HNum$, $+$, $preHis$)
5. $L \leftarrow$ new Array,$size = H * Key$
(*Phase 2. Building histograms*)
6. each thread builds its histogram(s) in L such that all histograms in segment i are all stored before those in segment $i+1$; but for all histograms within each segment, lower valued keys are all stored before higher valued ones.
7. scan L, and store results into L
(*Phase 3. Scatter elements to their sorted locations*)
8. each thread scatters its assigned elements to R_{out} according to the corresponding location in L, store that location in pos
end

Our overall segmented sorting algorithm basically performs radix sort on individual segments at the same time trying to achieve load balance. For strings, we repeatedly run the above algorithm every time using the characters at a specific position in the strings as the sorting keys. It starts from the least significant (rightmost) character of every string. Once we have reached the leftmost character, all strings become sorted. Each time we use the scatter result of the previous run as its input. To achieve load balance, we always assign an equal number of elements to each thread no matter whether these elements

belong to the same segment or not. Step 2 in phase 1 shows that each segment could be processed by several threads and a thread could process several segments. For example, thread tid is responsible for the subset $R_{in}[t, t+1, ..., t+P-1]$ and the first $k(1 < k < P)$ elements belong to i-th segment and the rest of the elements belong to the $i+1$-th segment, so two histograms should be allocated and built by thread tid with the first storing the counting results of the first k elements and the second for the rest.

The whole algorithm is divided into three phases. The first phase is the pre-computation phase for histogram allocation and thread assignment. Unlike unsegmented sort primitive proposed in [25] which allocates thread level histograms for each thread and then scatter them to a large array, our segmented sort primitive directly allocates one large array L to hold all histograms and each histogram has its designated locations in L. As step 6 shows, all the histograms for segments $1, ..., i-1$ should be stored before the histograms for segment i. However, within each segment, lower valued keys of all the histograms should be written before the higher valued keys. By doing this, relative positions of the elements within the same segment in the final sorted array are implied by the locations of their sorting keys in L. Note that each thread processes its assigned elements sequentially. Therefore, in step 8, every time after a thread deposits an element into the final sorted array, its destination index stored in L should be incremented by 1. This is to avoid write conflicts when another element with the same key comes the next time.

To the best of our knowledge, we are the first to propose a parallel algorithm for segmented sorting. We have compared our algorithm with the straightforward "segmented sort" implementation, which sequentially sorts one segment after another. Our experiments show that our parallel algorithm can achieve a three to ten fold speedup. The efficiency is due to the variance of segment lengths. The smaller the variance of segment lengths, the better the efficiency. Intuitively, a larger variance of segment lengths may evoke more histograms and thus incur more memory access latency.

4 GPU Data Structures for H-Trees

H-tree is a hyper-linked tree structure originally presented in [21] for efficient computation of iceberg cubes. It was later deployed again in [20] as the primary data structure for stream cubes. In the following, we briefly review the definition of an H-tree before introducing its actual implementation on GPUs.

1. An H-tree HT is a compact representation of all tuples in a relational database. HT has a root node "null". From the second level, each level in HT corresponds to a distinct dimension of the database. The order of the levels in HT follows a predefined order of the dimensions. Note that by default the dimensions are sorted in a cardinality ascending order. For stream data cubing [20], this order is set to suit the specific needs of a data cubing task and is called a popular path.

2. A tuple from the relational database is represented as a path from the root to a leaf. If two tuples share the same values in the first L dimensions in the predefined order, their corresponding paths in HT also share the first L segments. The two different values in the $L+1$-th dimension are stored in two children nodes of the node holding the shared value of their L-th dimension.

3. There is a header table for each level of the tree. It holds all distinct values of the corresponding dimension and the number of repetitions of each distinct value. All nodes sharing the same value are linked together by introducing an additional side link in each node. The header table also holds a pointer to the first node in each linked list.

4. All measures within a tuple are stored at the leaf node corresponding to that tuple. Intermediate nodes of HT hold aggregated measures resulting from data cube computation. An intermediate node saves the aggregated measures over the subset of tuples represented by the subtree rooted at the node. Thus, an H-tree is equivalent to a partially materialized data cube.

Each node in an H-tree needs to record at least such information as an attribute value of the corresponding dimension, parent index, the index of the leftmost child, the number of children as well as one or more aggregated measures. It can be summarized by the following structure.

```
struct HNode{
    char* atributeValue;
    HNode* parentPointer;
    HNode* leftMostChildPointer;
    HNode* sideLink;
    int childNum;
    int aggregatedMeasure;
}
```

Let us now describe our GPU data structures for an H-tree. For optimal GPU memory access performance, we use a structure of arrays (SoA) instead of an array of structures (AoS). Each level of the H-tree has six arrays with the same size. Each array holds the values of one field in the above data structure for all nodes at that level. There are various indices used for linking these arrays.

- key: an array holding attribute values of a dimension
- p: an array for parent indices
- m: an array holding values of an aggregated measure
- lmc: an array for the indices of the leftmost children
- cn: an array for the number of children for each node
- sL: an array of indices representing sidelink pointers.

The header table for each level is defined by three arrays with the same size.

- $hKey$: an array of distinct values in key
- hSL: an array holding the first indices of the sidelink lists

- hN: an array holding the number of repetitions of each distinct value

The header table holds values and indices facilitating query processing. The size of the header table for each level is equal to the cardinality of its corresponding dimension.

5 GPU Algorithms for H-Tree Based Data Cubing

5.1 GPU-Based H-Tree Construction

Parallel kd-tree construction [40] and parallel top-down bounding volume hierarchy construction [28] have already been widely used in computer graphics. Different from [28], our H-tree construction algorithm first takes a top-down approach to build the tree structure and then follows a bottom-up manner to aggregate measures at every tree node in parallel. In addition, our method heavily utilizes the proposed GPU primitives, such as segmented sort and segmented reduction, to achieve better efficiency.

In the following, we present the detailed algorithm for GPU-based H-tree construction. Since an H-tree is constructed from a collection of tuples in a relational table, the order of the dimensions (of this relational table) within the H-tree needs to be specified by OLAP experts. Note that a dimension may have multiple levels. Without loss of generality, we consider each distinct level as a new dimension in this paper.

Algorithm 1. H-TREE CONSTRUCTION
Input: $A_1, ..., A_m$, arrays holding values of the dimensions of a relational table, T, with n tuples, m dimensions and one measure mea without loss of generality (multiple measures can be handled similarly)
Output: $H[1, ..., m]$, an H-tree

Variables
pos, $newPos$, flg, $flg2$, $flg3$: 1D arrays of size n
$sF[1, ..., m][1, ..., n]$: 2D array
begin
1. initialize sF to 0
2. sort(A_1, $newPos$)
3. split(A_1, flg), $flg3 \leftarrow flg$
4. set $flg[1] = 0$, then scan(flg, $+$, $flg2$)
5. compact(A_1, $flg3$, $flg2$, $H[1].key$)
6. $flg \leftarrow flg3$, $sF[1] \leftarrow flg2$
7. **for** $i = 2$ to m
8. scatter(A_i, $newPos$, A_i)
9. segmented_sort(A_i, $flg2$, pos)
10. **for** each element p in $newPos$ in **parallel**
11. $newPos[p] \leftarrow pos[newPos[p]]$
12. segmented_split(A_i, $flg2$, $flg3$)
13. $flg \leftarrow flg3$
14. set $flg[1] = 0$, then scan(flg, $+$, $flg2$)
15. compact(A_i, $flg3$, $flg2$, $H[i].key$)
16. **for** each element e in $flg3$ in **parallel**
17. **if** $flg3[e] = 1$ **then**
18. $H[i].p[flg2[e]] \leftarrow sF[i-1][e]$
19. $flg \leftarrow flg3$, $sF[i] \leftarrow flg2$
20. GCHILDREN($H[i].p, H[i-1].lmc, H[i-1].cn$)
21. **if** $i = m$ **then**
22. scatter(mea, $newPos$, mea)
23. segmented_reduction(mea, $flg2$, $\bigoplus$, $H[i].m$)
end
end
24. **for** $i = m$ to 2
25. segmented_reduction($H[i].m$, $H[i].p$, $\bigoplus$, $H[i-1].m$)
26. HTABLE($H[i].key$)
end
end

The above pseudo code first follows a top-down approach when computing most of the properties at each node. At the beginning, we sort all tuples using the first dimension A_1 as the sorting key. Instead of updating the positions of all the remaining dimensions immediately, we only update the next dimension A_i according to $newPos$. Note that $newPos$ always traces the updated positions of the attribute values of the previously processed dimension A_{i-1}. Steps 10-11 show how to trace the updated positions.

After sorting in step 2, tuples with the same attribute value in the first dimension become contiguous in each segment. The split primitive is performed to label the first element of each segment to 1 and others to 0. In order to remove duplicated values in the sorted array, the compact primitive is performed in step 5 by only scattering the nodes whose corresponding label is 1. Within each segment in dimension A_i, all the elements share the same parent node. The segmented sort primitive in step 9 can make duplicate elements within each segment clustered in consecutive locations. Then the split and compact primitives can remove repetitions and obtain the final set of nodes for the level corresponding to dimension A_i.

The number of nodes at each level of an H-tree is dynamically determined by $flg2[n] + flg[n]$, where flg is a flag array and $flg2$ is the result of running the scan primitive on flg (step 10). $flg2$ also serves as the scatter location for each valid node. Steps 6 and 19 store the array $flg2$ into sF in order to retrieve the parent indices of the next level's nodes in steps 16-18. Note that we always use the array of parent indices at the current level to compute the array of children indices at the parent level.

Steps 23 and 25 perform roll-up operations to compute the aggregated measure for the internal nodes of the constructed H-tree using a bottom-up approach. Segmented reduction is perfectly suited for carrying out these roll-up operations, and our GPU data structures facilitate this process since the parent index of each node naturally serves as the $segmentID$.

Two steps, GCHILDREN and HTABLE, in the above pseudo code require further elaboration. GCHILDREN is responsible for computing the number of children and the index of the leftmost child for each node. HTABLE is responsible for constructing the header tables.

Procedure GCHILDREN
Input: $p[1, ..., n]$
Output: $lmc[1, ..., k], num[1, ..., k]$

begin
1. split(p, flg)
2. set $flg[1] = 0$, scan(flg, $+$, sf)
3. **for** each element i in flg in **parallel**
4. **if** $flg[i] = 1$ **then**
5. $lmc[sf[i]] \leftarrow i$
6. $T \leftarrow$ new Array, size = n; initialize all to 1
7. segmented_reduction(T, sf, $+$, num)

end

In the above procedure, we perform the split primitive on the parent index array. The result is the array of indices for the leftmost children of the nodes at the next higher level. Steps 6-7 show how to obtain the number of children for each node at the next higher level. Since children with the same parent are stored in contiguous locations, we can easily access one particular child using these two previously computed parameters.

Procedure HTABLE
Input: $R_{in}[1, ..., n]$
Output: $R_{out}[1, ..., k], Num[1, ..., k]$, $hSL[1, ..., k], sL[1, ..., n]$

begin
1. sort(R_{in}, pos)
2. **for** each element i in pos **parallel**
3. $sL[pos[i]] \leftarrow i$
4. split(R_{in}, flg), $flg2 \leftarrow flg$
5. set $flg[1] = 0$, scan(flg, $+$, sf)
6. compact(R_{in}, $flg2$, sf, R_{out})
7. compact(sL, $flg2$, sf, hSL)
8. $T \leftarrow$ new Array, $size = k$; initialize all to 1
9. segmented_reduction(T, sf, $+$, Num)

end

Note that the sidelink lists for each level are computed in procedure HTABLE, and side links in our algorithm are different from the sequential version which stores a pointer to the next node holding the same value. In our parallel version, by sorting the entire array of attribute values at each level, all the duplicate values are clustered in consecutive locations. After sorting, pos holds the new locations for all elements. The sidelink list is then computed by saving the original location for each element in pos. Steps 2-3 perform this task. Similar to procedure GCHILDREN, the number of repetitions for each distinct value of a dimension is computed by segmented reduction in steps 8-9. In the header table, we only store the first location of these repetitions and the number of repetitions for each distinct value.

5.2 GPU-Based H-Tree Update

In this section, we introduce a parallel algorithm for updating an H-tree. Such an operation would be very useful for streaming data. For example, a stream data cube should be continuously and incrementally updated with respect to a potentially infinite incoming data stream [20]. Instead of inserting one tuple at a time which would be a waste of the massive parallelism on the GPU, we propose to accumulate N incoming tuples and insert them simultaneously every time. This is in the same spirit as the algorithm presented in [33] for bulk incremental updates.

The key idea is that every time we need to update the H-tree, we parallelize over the N new tuples, and let each thread perform the insertion of one tuple in a purely sequential way. To some extent, we can simply treat each new tuple as one small H-tree which has only one branch from the root to the leaf. Therefore, updating the H-tree becomes equivalent to simultaneously merging N small H-trees with one existing large H-tree. Similar to the construction algorithm, the update algorithm also follows a top-down approach. Instead of immediately inserting each new node which could cause many write conflicts, we try to scatter two arrays, one for the nodes in the existing H-tree and the other for the new nodes constructed for the incoming tuples, simultaneously into a temporary third array. By performing the split and compact primitives, we can merge these arrays of nodes for each level in a way much similar to the H-tree construction algorithm. A key challenge in this update operation is to identify for each node in the second array the target location of the scatter operation. Once we have scattered these new nodes, the remaining slots in the temporary array are only left for nodes from the existing H-tree.

Algorithm 2. H-TREE UPDATE
Input : $oldTree[0, ..., m]$, an existing H-tree;
$A_1, ..., A_m$, arrays holding values of the dimensions of n incoming m-dimensional tuples
Output: updated H-tree

Variables
$newPos, pos, flg, sf, IsFound, preIndex$: 1D arrays of size n
$insertIndex[1, \ldots, m][1, \ldots, n]$: 2D array

begin
1. sort(A_1, $newPos$)
2. split(A_1, flg) 3. $flg[1] \leftarrow 0$, scan(flg, $+$, sf)
4. **for** $i = 1$ to m
5. **if** $i > 1$ **then**
6. scatter(A_i, $newPos$, A_i)
7. segmented_sort(A_i, sf, pos)
8. **for** each element p in pos **parallel**
9. $newPos[p] = pos[newPos[p]]$
10. segmented_split(A_i, sf, flg)
11. scan(flg, $+$, sf)

end
12. **for** each element e in A_i **parallel**
13. $left \leftarrow oldTree[i-1][preIndex[e]].lmc$
14. $right \leftarrow left + oldTree[i-1][preIndex[e]].cn$
15. **if** $IsFound[e]$ **then**
16. binary search $A_i[e]$ in the $oldTree[i].key$ from $[left, right)$, set the search stop index to ss
17. $insertIndex[i][e] \leftarrow ss$
18. **if** $A_i[e] != oldTree[i].key[ss]$ **then**
19. $IsFound[e] \leftarrow false$

end
20. **else**
21. $insertIndex[i][e] \leftarrow right$

22. $preIndex[e] \leftarrow insertIndex[i][e]$
23. $insertIndex[i][e] += e + 1$
end
24. $middleData \leftarrow$ new Array, $size = oldTree[i].size + n$
25. scatter(A_i, $insertIndex[i]$, $middleData$)
26. scatter $oldTree[i]$ to the empty positions of $middleData$
27. split and compact $middleData$ to get final nodes
28. compute parent(children) indices using split results
end
29. roll up to get each new node's aggregated measure
30. re-compute header table for each level
end

In the above pseudo code for parallel H-tree update, some of the ideas are similar to the construction algorithm. Steps 12-28 show the differences. For every small H-tree(one tuple), we search for each of its nodes in a set of candidate nodes of the existing H-tree. Since we maintain the nodes in a sorted order, binary search can be applied to perform the search efficiently. There are two observations. First, if one node finds a match in the existing H-tree, its child should only be searched among the children of the matched node. Second, if one node cannot find a match in the candidate set, descendants of the candidates can be safely pruned. The incoming node should be inserted as the rightmost child of the matched node at the higher level. Steps 14 and 19 show the search process.

After step 23, $insertIndex[i][e]$ holds the scatter location for each incoming new node. We allocate one temporary array, *valid*, and initialize all its elements to 1. Then we reset all its elements corresponding to the locations recorded in $insertIndex[i]$ to 0. Those elements remaining to be 1 indicate the locations for the nodes from the existing H-tree. We use $oldPos[i]$ to represent the scatter locations for nodes at level i of the existing H-tree. $oldPos[i]$ can be set using the result from an excluded scan of $valid$.

The attribute values of the dimensions and parent indices of the nodes in the updated H-tree are both computed by steps 24-28. We must update the parent indices of the nodes in the existing H-tree before scattering them to the *middleData* array because at this time, their parent nodes have already been scattered. Parent indices of the new nodes can be directly fetched from the scatter locations of their parent nodes in the previous step. Steps 29 and 30 compute the aggregated measure array and header tables in the same way as the construction algorithm.

5.3 GPU-Based H-Tree Query Processing

In this section, we describe GPU-based algorithms supporting data cube queries using the constructed H-tree. Since an H-tree represents a popular path in the corresponding data cube, it only materializes the cuboids along this popular path. When a query can be answered by simply fetching an aggregated measure from an H-tree node, there is not much computation involved except for the lookup operation to locate the node. Otherwise, online cubing (aggregation) needs to be performed to answer the query using partially materialized results in the H-tree. According to [20], the precomputed results in the H-tree can significantly accelerate online cubing. Unlike a sequential algorithm which processes queries one by one, our GPU-based algorithm can process N queries in parallel. We assign one thread to each query that only requires the lookup operation, but parallelize any query that requires online cubing over multiple threads to achieve better load balance. Each thread then processes the data related to a portion of a sidelink list.

Without loss of generality, our query processing algorithms only address point queries, which only have instantiated dimensions but no inquired dimensions. Subset (or subcube) queries with inquired dimensions can always be decomposed into multiple point queries which can then be processed in parallel on the GPU.

5.3.1 Simple Lookup

We first present the algorithm for queries that can be directly answered by H-tree lookups. For stream cubes, this is called on-popular-path queries [20].

Algorithm 3: ON-POPULAR-PATH_QUERY
Input: $H[0, ..., m]$, an H-tree;
$q[1, ..., n][1, ..., m]$, n queries each with at most m instantiated dimensions
Output: $R_{out}[1, ..., n]$: results of the n queries

begin
1. $left \leftarrow H[0].lmc$
2. $right \leftarrow left + H[0].cn$
3. $j \leftarrow 1$
4. **for** each query $q[i]$ in **parallel**
5. binary search $q[i][j]$ in $H[j].key$ between $[left, right)$
6. **if** found **then**
7. let $fIndex$ be the found index
8. **if** $q[i][j]$ is the last instantiated dimension of $q[i]$ **then**
9. $R_{out}[i] \leftarrow H[j].m[fIndex]$
10. **return**
11. $lmChild \leftarrow H[j].key[fIndex]$
12. $childNum \leftarrow H[j].cn[findex]$
end
13. **else**
14. **return** not found
end
15. $j \leftarrow j + 1$
16. **goto** step 4
end

In the above algorithm, $H[i](i = 1, ..., m)$ represents the pointer to the i-th level ($H[0]$ is the root). $q[i][j]$ represents the value of the j-th instantiated dimension of query i. For an on-popular-path query, instantiated dimensions must match attribute values of dimensions along a contiguous segment of a path in the H-tree and this contiguous segment must start from the root. Thus, all the instantiated dimensions in the query must be checked, if they can all be found in the corresponding levels of the H-tree, we simply return the aggregated measure

stored at the last instantiated dimension in the tree. Otherwise, return zero or not defined. If the fan-out factor of every node in the H-tree is C, at most $TotalVisit = \frac{1}{2} C\, Num_r$ nodes would be visited during the lookup, where Num_r represents the number of instantiated dimensions. Since all children of a node are maintained as an ordered list, we can use binary search to speed up the lookup operation.

5.3.2 Online Cubing

The more general case involves queries not on the popular path. Such queries require online cubing to compute any aggregated measure. That means the final result may be answered by aggregating partial results at multiple nodes. Such aggregations are performed on the fly.

Algorithm 4: ONLINE_CUBING
Input: $H[1, ..., m]$, an H-tree;
$q[1, ..., n][1, ..., m]$, n queries each with at most m instantiated dimensions
Output: $R_{out}[1, ..., n]$, results of the n queries

begin
1. set up n thread blocks with block size S on the GPU
2. $Hist \leftarrow$ new Array[n*S], initialize all to 0
3. **for** each query $q[i]$ in **parallel**
4. assign $q[i]$ to the i-th thread block whose ID is bid
5. find the lowest level whose corresponding dimension is instantiated in query $q[i]$, let the found level be $fLevel$
6. search $q[i][fLevel]$ in $H[fLevel].hKey$
7. **if** found **then**
Let t be the found index in $H[fLevel].hKey$
8. $num \leftarrow H[fLevel].hN[t]$
9. $st \leftarrow H[fLevel].hSL[t]$
end
10. **else**
11. **return** not found
12. divide the elements in $H[fLevel].sL[st, ..., st + num]$ into chunks, the size per chunk is $chs = num/S$
13. **for** each thread tid in block bid in **parallel**
14. **for** $k = 1$ to chs
15. $old \leftarrow H[fLevel].sL[tid * chs + k + st]$
16. $pi = H[fLevel].p[old]$
17. $j \leftarrow fLevel - 1$
18. $flg \leftarrow true$
19. **while**($j >= 0$) **do**
20. **if** $q[i][j]$ is instantiated in query $q[i]$ **then**
21. **if** $q[i][j] = H[j].key[pi]$ **then**
22. $pi \leftarrow H[j].p[pi]$
23. **else**
24. $flg \leftarrow false$, break
25. $j \leftarrow j - 1$
end
26. **if** $flg = true$ **then**
27. $Hist[bid*S+tid] \bigoplus = H[fLevel].m[old]$
end
end
end

28. $segID \leftarrow$new Array[n*S]
29. **for** each element e in $segID$ in **parallel**
30. $segID[e] \leftarrow e/S$
31. segmented_reduction($Hist$, $segID$, $\bigoplus$, R_{out})
end

In the above algorithm, if the size of the header table is sufficiently large, step 6 can also be parallelized. Both the starting index of a sidelink list and the number of nodes in the sidelink list, can be directly retrieved as in steps 8 and 9. Then we parallelize over all elements in the sidelink list. Since the number of elements in a sidelink list can be large, we use one thread block to process them and the elements are evenly distributed among all threads within the block. Each element in the sidelink list has an index into the array holding the attribute values of a dimension. We can use this same index to retrieve the parent index and move up the tree, checking all instantiated dimensions on the way up. Steps 19-27 show this process. Step 31 performs segmented reduction on $Hist$ to obtain the final aggregated result of the inquired measure for each query. Since one thread block is responsible for one query, we can use the block ID as the segment ID in the segmented reduction primitive.

Please note that for a single query, our method achieves load balance very well thanks to parallel segmented reduction. However, for multiple queries, load balance would become an issue, and our current online cubing algorithm has not explicitly considered this issue. To further improve the query response time, we need an explicit cache model to store the top-k most popular queries. Nevertheless, this is a challenging problem out of the scope of this paper.

5.3.3 Hybrid Queries

In a real scenario, there is usually a combination of both types of queries. We can split the batch of queries into two separate groups, first invoke the GPU kernel for on-popular-path queries, and then invoke the online cubing kernel for the second group. Finally, we merge the query results together.

6 EXPERIMENTAL RESULTS

In this section, we evaluate the performance of our proposed GPU algorithms, including H-tree construction, update and query processing algorithms, and compare them with the corresponding algorithms on the CPU.

6.1 Experimental Configuration

The described algorithms have been implemented and tested on an Intel Core 2 Duo quad-core processor with an NVidia GeForce 8800 GTX GPU. The details of the experimental setup are shown in Table 1. The GPU uses PCI-E 2.0 bus to transfer data between the host (CPU memory) and the device with a theoretical bandwidth of 4GB/s. The 128 processors on the GPU are organized into 16 SMs (stream multiprocessor) with each SM executing multiple thread blocks concurrently. There are 16KB shared memory and 8192 registers in each

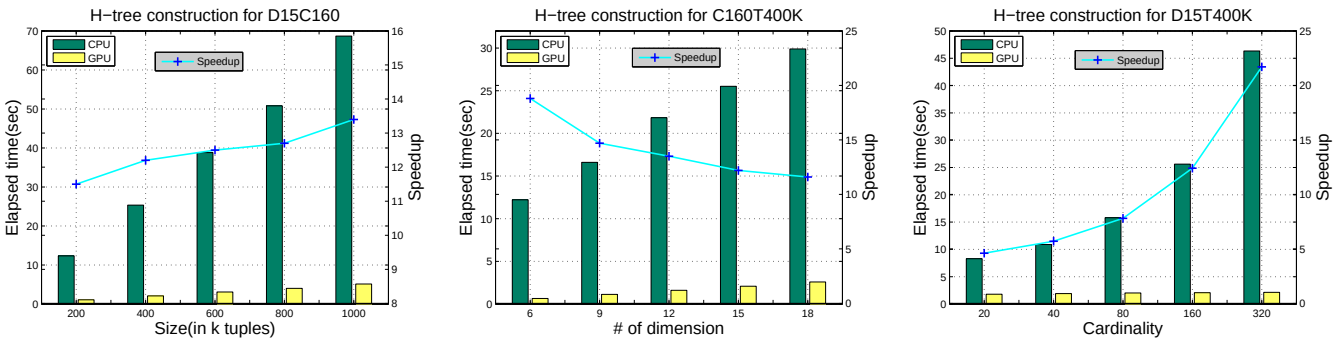

Figure 1: Time and GPU speedups for H-tree construction. (*left*) speedup and time vs. the number of tuples; (*middle*) speedup and time vs. # of dimensions; (*right*) speedup and time vs. cardinality of each dimension.

Table 1: Hardware configuration

	CPU	GPU
processor	2.4GHz x 4	1.35GHz x 128
data cache	L1:32KB x 4, L2:4096KB x 2	16KB x 8
cache latency (cycles)	L1:2, L2:8	1
DRAM(MB)	4096	768
DRAM latency (cycles)	300	400-600
bus width (bits)	64	384

SM. Note that, shared memory is pure CUDA implementation. It is used as data exchange cache within a thread block, and is as fast as registers if there are no bank conflicts. More transistors on the GPU have been devoted to data processing rather than data caching and flow control. In order to fully keep all the ALUs busy, thousands of threads should be spawn to hide GPU memory access latency.

Note that even though the GPU has much more processors, it has higher memory access latency and less amount of data cache, and its processors have a lower clock rate. Therefore, the peak performance of the GeForce 8800 GTX GPU is 346 GFlops which is only 9x the peak performance of a single core on the Intel Core 2 Duo processor.

6.2 Data Generator

We have tested our algorithms with synthetic data sets, which were produced by a data generator similar to the IBM data generator [4] designed for testing data mining algorithms. We use the dataset convention as defined in [20], e.g., D6C20T800K means there are 6 dimensions with the cardinality of each dimension set to 20, and there are a total of 800k tuples.

6.3 Implementation on GPU

Our proposed algorithms are implemented using CUDA [3], which is the only C language environment for GPUs. The parallel task is encoded as kernels invoked by the host but run on the device. At any point, there is only one kernel being executed on the GPU. Global synchronization is implemented through the GPU memory. That means the current running kernel uses the output of the previous kernel as its input. When setting up a kernel, the number of thread blocks and the number of threads in each block have to be specified. Issuing more threads per block results in less allocated resource (registers, etc.) per thread, but can potentially improve the overall performance because more memory access latency could be masked. Our experiments show that the number of threads per block should be between 128 and 256 for H-tree construction and 32 for online cubing.

No atomic operations are supported for GPUs within CUDA 1.1 or below. Since NVidia GeForce 8800GTX is only compatible with CUDA 1.0, software strategies are used to avoid write conflicts. For example, in the sorting and segmented sorting primitives, each thread owns its assigned histogram(s) and only counts the elements it is responsible for. Afterwards, the scan primitive is run over all the histograms to generate write locations for all elements.

Note that the number of nodes at each level of an H-tree is dynamically determined by the scan result of a flag vector for the sorted attribute values of a dimension. Thus the memory for the H-tree is also dynamically allocated. Since we process relational datasets, values of a dimension are usually specified as character strings. Our string type data is implemented using $char4$ pointer array in order to improve the efficiency of memory access and alignment. The pointer is declared on the host and GPU memory for that array is allocated by the $cudaMalloc$ API. We then transfer the data from the host to the GPU allocated array by $cudaMemcpy$ API.

6.4 Optimization Considerations

6.4.1 Minimize memory overhead

First, low data transfer rate over the PCI-Express bus negatively affects the overall performance. Frequent data transfer between GPU and CPU memories should be avoided. To achieve this goal, the entire constructed H-tree always resides in the GPU memory instead of being sent back to the system memory. We also suggest to batch many small transfers into a large one due to the overhead associated with each transfer. Second, shared memory within the SMs should be utilized as much as possible to improve data locality and avoid GPU memory access latency. For example, in the online cubing al-

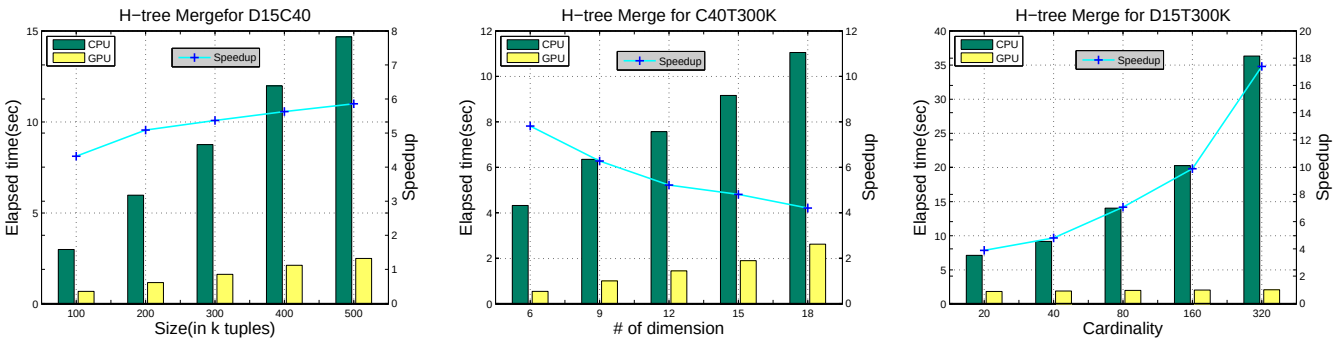

Figure 2: Time and GPU speedups for H-tree update. (*left*) speedup and time vs. the number of new tuples; (*middle*) speedup and time vs. # of dimensions; (*right*) speedup and time vs. cardinality of each dimension.

gorithm, we use one block to process each query so that we can load the instantiated dimension values into the shared memory to improve locality since they are accessed multiple times. Third, coalesced GPU memory access patterns should be exploited to improve performance. Coalescing means if threads in a half-warp access consecutive memory words, these memory accesses can be coalesced into one memory transaction. Otherwise, a separate memory transaction is issued for each thread and throughput is significantly reduced. Our results show that coalesced access can achieve 1.5 to 3 times speedup when compared with noncoalesced access.

6.4.2 Avoid bank conflicts

Shared memory is on-chip. It is as fast as registers as long as there are no bank conflicts between threads. So much attention should be taken by developers when using shared memory. To achieve high memory bandwidth, shared memory is divided into equally-sized modules, called banks, which can be accessed simultaneously. Bank conflicts occur when multiple memory requests fall into the same bank, in which condition the access has to be serialized, which significantly decreases the shared memory bandwidth. Shared memory padding is one possible solution to avoid it. Our proposed structures of arrays (SoA) can avoid bank conflicts to some extent by design.

6.4.3 Minimize branch divergence

Recent GPUs include control flow instructions in the GPU instruction set but programs using these instructions may experience reduced performance due to branch divergence. Branch divergence occurs only within a warp. Different warps execute independently regardless of whether they are following the common code path or not. Once divergence occurs, the warp serially executes each path taken, disabling the threads not on the current running path. Therefore, the use of control flow instructions should be minimized when possible.

6.5 Analysis

Let us now present the time and space complexity of the proposed parallel H-tree algorithms. Let P be the number of processors ($P = 128$ in G80), N be the number of incoming tuples for H-tree construction or update, M be the number of dimensions, C be the maximum cardinality of any dimension, L be the maximum length of the strings used as values of the dimensions, T be the number of tuples in an existing H-tree, and Key be the number of distinct keys in the histograms used in the sorting (sort and segmented sort) primitives.

6.5.1 Time Complexity

The time complexity for H-tree construction and update are $O(NML/P+M\log N)$ and $O(NML/P+M\log(N+T)+M\log C/P+TM/P)$, respectively. And the time complexity for an on-popular-path query and online cubing query are $O(M\log C/P)$ and $O(NM/(CP))$ respectively. So from a theoretical perspective, on-line cubing is more complicated than on-popular-path queries.

6.5.2 Space Complexity

The total amount of device memory used by H-tree construction in addition to the memory holding input data is $O(NM+F(N))$, where $F(N)$ represents the memory cost of sorting primitives. In the worse case, $F(N) = (Key+1)N$ (each element is a distinct segment). Similarly, the total amount of additional device memory used by H-tree update is $O(NM + S_{old} + F(N))$, where S_{old} represents the size of the existing H-tree. In an H-tree update, we need to first put the existing tree and incoming tuples together before eliminating duplicate nodes, which incurs additional memory cost compared with H-tree construction. The memory cost by online cubing is $O(S)$ (where S is the block size) per query, since it needs to summarize all the partial values computed by each thread. An on-popular-path query costs the least amount of memory which is only $O(1)$ per query.

6.6 Results

In this section, we analyze the performance of our proposed GPU algorithms over their CPU counterparts. In our experiments, we investigated how the total number of tuples, the total number of dimensions and the cardinality of each dimension affect the performance of our algorithms. There are two main reasons giving rise to the reported performance. One

lies in the differences in the hardware architecture, including memory stalls, cache amount, the type of processors, memory bandwidths and so on. The other lies in the differences in algorithm design choices. We will analyze which one contributes more. As mentioned in Section 6.1, the peak performance of the GPU we use is only 9x the peak performance of a single core on the CPU. This peak performance ratio should be used as a reference number when we look at the speedups of our algorithms. In all experiments, the performance of the GPU algorithms was compared against the performance of an optimized sequential algorithm running on a single core of the CPU.

Figure 1 demonstrates the performance of our parallel H-tree construction algorithm, which typically achieves a speedup above 10. Figure 1(a) shows the elapsed time on both CPU and GPU as well as the speedups for H-tree construction on datasets with an increasing number of tuples. The number of dimensions is set to 15 and the cardinality of each dimension is set to 160. We found that our GPU-based construction algorithm scaled better with larger datasets compared with the CPU-based sequential algorithm. This is probably because it is easier to achieve load balance with larger datasets. Figure 1(b) shows the elapsed time and speedups for H-tree construction on datasets with an increasing number of dimensions. The cardinality of each dimension is set to 160 and there are 400k tuples in each dataset. The results show that the GPU algorithm has more advantage when the number of dimensions is relatively small. Figure 1(c) shows the elapsed time and speedups for H-tree construction on datasets with the cardinality of each dimension increasing from 20 to 320. There are 15 dimensions and 400k tuples in each dataset. The running time of both CPU and GPU algorithms increases with the cardinality of each dimension. However, the running time of our GPU algorithm increases at a much slower rate than that of the CPU algorithm.

Figure 2 shows the performance of our parallel H-tree update algorithm. Figure 2(a) shows the elapsed time and speedups for updating an existing H-tree constructed from $D15C40T300K$ using an increasing number of incoming tuples. The speedup improves with the number of incoming tuples. For stream cubes, this parameter can be tuned to suit the data arrival rate. In figure 2(b), the number of dimensions increases from 6 to 18. The existing H-tree was constructed from 300K tuples and the cardinality was set to 40. We update the existing H-tree using 300K incoming tuples. The results show that our algorithm is more advantageous when the number of dimensions is relatively small. Figure 2(c) shows that when the cardinality of each dimension varies from 20 to 320, the running time of both CPU and GPU algorithms are increasing, but our GPU-based algorithm increases at a much slower rate. This is because the CPU algorithm sequentially builds the header table and children pointers. Before every node insertion, it needs to check its existence in the header table and sequentially find the correct location to insert. On the other hand, our GPU algorithm processes nodes in parallel, and maintains the children nodes of the same parent in a sorted order so binary search can be performed to accelerate

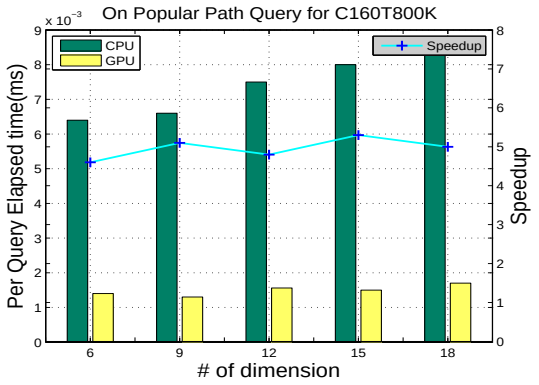

Figure 3: GPU speedup and average time vs. # of dimensions for on-popular-path queries

the process.

Figures 3-6 show the performance of our data cube query processing algorithms. To test the on-popular-path query algorithm, we randomly generate 10K on-popular-path queries with each one containing a random number of instantiated dimensions. To test the online cubing algorithm, we also randomly generate 10K queries none of which is on the popular path. Then we analyze the average elapsed time per query in each type. Since our H-tree resides in the GPU device memory, we need to transfer the queries from the host to the device memory and then transfer the results back to the host again. We have included these transfer times in the reported average elapsed time per query. Note that our online cubing algorithm can typically achieve a speedup much larger than 10.

Figures 3 and 5 show the performance of both query processing algorithms with an increasing number of dimensions. Note that varying the number of dimensions has little impact on on-popular-path queries. This is because we let each thread process one query in a purely sequential way, and the overhead of parallelization becomes minimal. However, for the online cubing, our GPU algorithm has less advantage with a larger number of dimensions. Figures 4 and 6 show the performance of both query algorithms with an increasing cardinality of the dimensions. Similarly, varying cardinality has little impact on on-popular-path queries. Compared with CPU query algorithms, both our GPU query algorithms have more advantage when the cardinality is relatively small. This is mainly because a larger cardinality of the dimensions gives rise to larger H-trees which require more memory accesses to search for instantiated dimensions of a query. Note that we also tested hybrid queries and the performance is similar to online cubing since on-popular-path queries only needs a small portion of the overall running time.

In the reported performance comparisons, we can find that the speedup always decreases with an increasing number of dimensions for all GPU algorithms. This is because with a larger number of dimensions, more global arrays need to be allocated to store values of additional dimensions, parent(child) indices, side links, header tables and so on. As a result, more scatter operations are needed to update these arrays, and compact the H-tree nodes and header table elements. Since the scatter

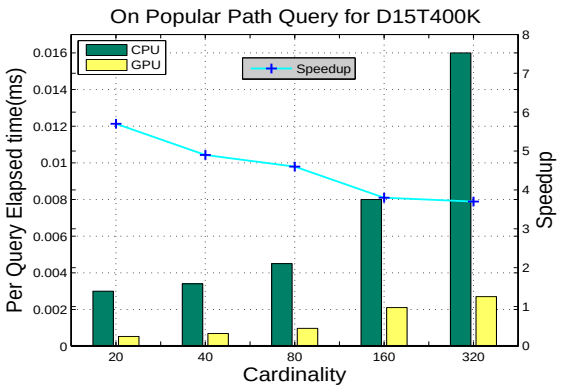

Figure 4: GPU speedup and average time vs. cardinality of each dimension for on-popular-path queries

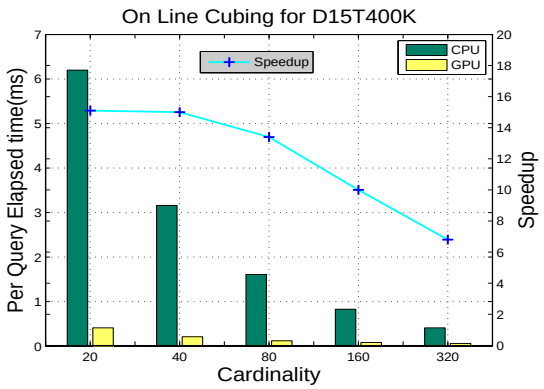

Figure 6: GPU speedup and average time vs. cardinality of each dimension for online cubing

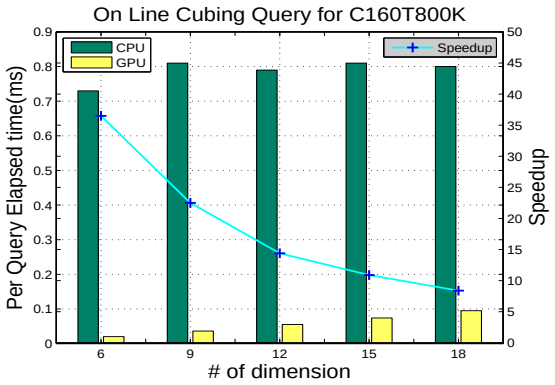

Figure 5: GPU speedup and average time vs. # of dimensions for online cubing

Table 2: Breakdown of GPU Running Times (ms)

	Search	I/O	Key	Index	Htable	Rollup
H-tree construction for D15C40T600K	N/A	23	1948	118	559	36
H-tree update D15C40T300K	25	14	1058	257	530	38
10K on popular path queries	7.4	0.68	N/A	N/A	N/A	N/A
10K online cubing queries	3758	0.72	N/A	N/A	N/A	1.83

operation cannot guarantee coalesced memory access to the GPU memory and branch divergence is hard to avoid within the warps, the overall performance is negatively affected. Another important reason is that GPU memory access latency is about 400-600 cycles which is longer than CPU DRAM access latency.

Table 2 shows the breakdown of running times for the four GPU algorithms. Note that in the H-tree construction and update algorithms, the I/O item only includes the data transfer time from the host to the device since we maintain the H-tree on the device. But in the query processing algorithms, we include the transfer time from the device back to the host. From the table we can find out that I/O operations only occupy 8% of the total execution time of the on-popular-path query algorithm. Compared with on-popular-path queries, online cubing takes much more time. Although search pays an important role in the update algorithm, its cost can be ignored when compared with other components. Note that in both H-tree construction and update algorithms, computation related to values of dimensions, often used as sorting keys, is dominant. This is mainly due to the segmented sort primitive. The "Index" column in the table includes the time to compute parent and children indices. Compared with H-tree construction, H-tree update costs more in this item because it has to first put the incoming nodes and the existing nodes together, and then eliminate the duplicated ones. Header table related computation takes the second largest portion of the running time because before compacting the header table we have to sort the array of nodes at each level. Finally, we found that the roll-up operation cost very little because our segmented reduction primitive was well optimized with padding to avoid bank conflicts. Note that we ran the query processing algorithms on an H-tree constructed for D15C40T600K.

7 Conclusions

In this paper, we have presented efficient GPU-based parallel algorithms for H-tree based data cube operations. This has been made possible by parallel primitives supporting segmented data and efficient memory access patterns. As a result, our GPU algorithms can achieve a speedup comparable to the peak performance ratio between the GPU and a single core on the CPU most of the time. Our H-tree construction and online cubing algorithms can often achieve a speedup much larger than 10. We have analyzed various factors that can either positively or negatively affect the performance. To the best of our knowledge, this is the first attempt to develop parallel data cubing algorithms on graphics processors.

Acknowledgments

This work was partially supported by National Natural Science Foundation of China (60728204/F020404) and National Science Foundation (IIS 09-14631).

References

[1] Amd ctm. http://ati.amd.com/products/streamprocessor.

[2] Opengl. http://www.opengl.org.

[3] Nvidia cuda (compute unified device architecture) programming guide 2.0. http://developer.nvidia.com/object/cuda.html, 2008.

[4] R. Agrawal and R. Srikant. Mining sequential patterns. In *Proc. Int. Conf. Data Engineering (ICDE)*, 1995.

[5] N. Bandi, C. Su, D. Agrawal, and A. El Abbadi. Hardware acceleration in commercial databases. In *VLDB*, 2004.

[6] D. Blythe. The direct3d 10 system. In *SIGGRAPH*, 2006.

[7] D. Cederman and P. Tsigas. A practical quicksort algorithm for graphics processors. Technical report, No.2008-01, 2008.

[8] S. Chaudhuri and U. Dayal. An overview of data warehousing and OLAP technology. *SIGMOD Record*, 26:65–74, 1997.

[9] Y. Chen, F. Dehne, T. Eavis, and A. Rau-Chaplin. Parallel rolap data cube construction on shared-nothing multiprocessors. *Distributed and Parallel Databases*, 15:219–236, 2004.

[10] F. Dehne, T. Eavis, S. Hambrusch, and A. Rau-Chaplin. Parallelizing the data cube. *Distributed and Parallel Databases*, 11:181–201, 2002.

[11] F. Dehne, T. Eavis, and A. Rau-Chaplin. Computing partial data cubes for parallel data warehousing applications. In *Euro PVM/MPI*, 2001.

[12] F. Dehne, T. Eavis, and A. Rau-Chaplin. The cgmcube project: Optimizing parallel data cube generation for rolap. *Distributed and Parallel Databases*, 19:29–62, 2006.

[13] S. Goil and A. Choudhary. High performance olap and data mining on parallel computers. *Data Mining and Knowledge Discovery*, 1:391–417, 1997.

[14] N. Govindaraju, J. Gray, R. Kumar, and D. Manocha. Gputerasort: high performance graphics co-processor sorting for large database managment. In *SIGMOD*, 2006.

[15] N. Govindaraju, B. Lloyd, W. Wang, M. Lin, and D. Manocha. Fast computation of database operations using graphics processors. In *SIGMOD*, 2004.

[16] N. Govindaraju, N. Raghuvanshi, and D. Manocha. Fast and approximate stream mining of quantiles and frequencies using graphics processors. In *SIGMOD*, 2005.

[17] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data cube: A relational operator generalizing group-by, cross-tab and sub-totals. In *Proc. 1996 Int. Conf. Data Engineering (ICDE'96)*, pages 152–159, New Orleans, LA, Feb. 1996.

[18] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab and sub-totals. *Data Mining and Knowledge Discovery*, 1:29–54, 1997.

[19] W. Gropp, E. Lusk, and A. Skjullum. *Using MPI: Portable Parallel Programming with the Message Passing Interface*. MIT Press, 1994.

[20] J. Han, Y. Chen, G. Dong, J. Pei, B.W. Wah, J. Wang, and Y.D. Cai. Stream cube: An architecture for multidimensional analysis of data streams. *Distributed and Parallel Databases*, 18(2):173–197, 2005.

[21] J. Han, J. Pei, G. Dong, and K. Wang. Efficient computation of iceberg cubes with complex measures. In *SIGMOD*, 2001.

[22] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *SIGMOD*, 2000.

[23] M. Harris, J.D. Owens, S. Sengupta, Y. Zhang, and A. Davidson. Cudpp library. CUDPP homepage. http://www.gpgpu.org/developer/cudapp, 2007.

[24] B. He, N. Govindaraju, Q. Luo, and B. Smith. Efficient gather and scatter operations on graphics processors. In *ACM/IEEE Supercomputing*, 2007.

[25] B. He, K. Yang, R. Fang, M. Lu, N.K. Govindaraju, Q. Luo, and P.V. Sander. Relational joins on graphics processors. In *SIGMOD*, 2008.

[26] D. Horn. Stream reduction operations for gpgpu applications. In M. Pharr, editor, *GPU Gems 2*, pages 573–589. Addision Wesley, 2005.

[27] R. Jin, K. Vaidyanathan, G. Yang, and G. Agrawal. Communication and memory optimal parallel data cube construction. *IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS*, 16(12):1105–1119, 2005.

[28] Mo Q. Lauterbach, C. and D. Manocha.

[29] M.D. Lieberman, J. Sankaranarayanan, and H. Samet. A fast similarity join algorithm using graphics processing units. In *ICDE*, 2008.

[30] H. Lu, X. Huang, and Z. Li. Computing data cubes using massively parallel processors. In *Proc. 7th Parallel Computing Workshop*, 1997.

[31] J.D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A.E. Lefohn, and T.J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26, 2007.

[32] S. Popov, J.M. Gunther, H.-P. Seidel, and P. Slusallek. Stackless kd-tree traversal for high performance gpu ray tracing. In *Eurographics*, pages 415–424, 2007.

[33] N. Roussopoulos, Y. Kotidis, and M. Roussopoulos. Cubetree: Organization of and bulk incremental updates on the data cube. In *SIGMOD*, 1997.

[34] S. Sarawagi, R. Agrawal, and N. Megiddo. Discovery-driven exploration of OLAP data cubes. In *Proc. Int. Conf. of Extending Database Technology (EDBT'98)*, pages 168–182, Valencia, Spain, Mar. 1998.

[35] J. Schwartz. Ultracomputers. *ACM Transactions on Programming Languages and Systems*, 2(4):484–521, 1980.

[36] S. Sengupta, M. Harris, Y. Zhang, and J.D. Owens. Scan primitives for gpu computing. In *ACM SIGGRAPH/EUROGRAPHICS conference on Graphics Hardware*, 2007.

[37] S. Sengupta, A.E. Lefohn, and J.D. Owens. A work-efficient step-efficient prefix sum algorithm. In *Proceedings of the Workshop on Edge Computing Using New Commodity Architectures*, pages D–26–27, 2006.

[38] C. Sun, D. Agrawal, and A. El Abbadi. Hardware acceleration for spatial selections and joins. In *SIGMOD*, 2003.

[39] M. Zagha and G.E. Blelloch. Radix sort for vector multiprocessors. In *ACM/IEEE Supercomputing*, 1991.

[40] K. Zhou, Q. Hou, R. Wang, and B. Guo. Real-time kd-tree construction on graphics hardware. In *SIGGRAPH*, 2008.