Fast Computation of Database Operations using Graphics Processors

Naga K. Govindaraju Brandon Lloyd Wei Wang Ming Lin Dinesh Manocha

University of North Carolina at Chapel Hill

{naga, blloyd, weiwang, lin, dm}@cs.unc.edu

http://gamma.cs.unc.edu/DataBase

ABSTRACT

We present new algorithms for performing fast computation of several common database operations on commodity graphics processors. Specifically, we consider operations such as conjunctive selections, aggregations, and semi-linear queries, which are essential computational components of typical database, data warehousing, and data mining applications. While graphics processing units (GPUs) have been designed for fast display of geometric primitives, we utilize the inherent pipelining and parallelism, single instruction and multiple data (SIMD) capabilities, and vector processing functionality of GPUs, for evaluating boolean predicate combinations and semi-linear queries on attributes and executing database operations efficiently. Our algorithms take into account some of the limitations of the programming model of current GPUs and perform no data rearrangements. Our algorithms have been implemented on a programmable GPU (e.g. NVIDIA's GeForce FX 5900) and applied to databases consisting of up to a million records. We have compared their performance with an optimized implementation of CPU-based algorithms. Our experiments indicate that the graphics processor available on commodity computer systems is an effective co-processor for performing database operations.

Keywords: graphics processor, query optimization, selection query, aggregation, selectivity analysis, semi-linear query.

1. INTRODUCTION

As database technology becomes pervasive, Database Management Systems (DBMSs) have been deployed in a wide variety of applications. The rapid growth of data volume for the past decades has intensified the need for high-speed database management systems. Most database queries and, more recently, data warehousing and data mining applications, are very data- and computation-intensive and therefore demand high processing power. Researchers have actively sought to design and develop architectures and algorithms for faster query execution. Special attention has been given to increase the performance of selection, aggregation, and join operations on large databases. These operations are widely used as fundamental primitives for building complex database queries and for supporting on-line analytic processing (OLAP) and data mining procedures. The efficiency of these operations has a significant impact on the performance of a database system.

As the current trend of database architecture moves from disk-based system towards main-memory databases, applications have become increasingly computation- and memory-bound. Recent work [3, 21] investigating the processor and memory behaviors of current DBMSs has demonstrated a significant increase in the query execution time due to memory stalls (on account of data and instruction misses), branch mispredictions, and resource stalls (due to instruction dependencies and hardware specific characteristics). Increased attention has been given on redesigning traditional database algorithms for fully utilizing the available architectural features and for exploiting parallel execution possibilities, minimizing memory and resource stalls, and reducing branch mispredictions [2, 5, 20, 24, 31, 32, 34, 37].

1.1 Graphics Processing Units

In this paper, we exploit the computational power of graphics processing units (GPUs) for database operations. In the last decade, high-performance 3D graphics hardware has become as ubiquitous as floating-point hardware. Graphics processors are now a part of almost every personal computer, game console, or workstation. In fact, the two major computational components of a desktop computer system are its main central processing unit (CPU) and its (GPU). While CPUs are used for general purpose computation, GPUs have been primarily designed for transforming, rendering, and texturing geometric primitives, such as triangles. The driving application of GPUs has been fast rendering for visual simulation, virtual reality, and computer gaming.

GPUs are increasingly being used as co-processors to CPUs. GPUs are extremely fast and are capable of processing tens of millions of geometric primitives per second. The peak performance of GPUs has been increasing at the rate of $2.5 - 3.0$ times a year, much faster than the Moore's law for CPUs. At this rate, the GPU's peak performance may move into the teraflop range by 2006 [19]. Most of this performance arises from multiple processing units and stream processing. The GPU treats the vertices and pixels constituting graphics primitives as streams. Multiple *vertex* and

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SIGMOD 2004 June 13-18, 2004, Paris, France.

Copyright 2004 ACM 1-58113-859-8/04/06 . . . $5.00.

pixel processing engines on a GPU are connected via data flows. These processing engines perform simple operations in parallel.

Recently, GPUs have become programmable, allowing a user to write *fragment programs* that are executed on pixel processing engines. The pixel processing engines have direct access to the texture memory and can perform vector operations with floating point arithmetic. These capabilities have been successfully exploited for many geometric and scientific applications. As graphics hardware becomes increasingly programmable and powerful, the roles of CPUs and GPUs in computing are being redefined.

1.2 Main Contributions

In this paper, we present novel algorithms for fast computation of database operations on GPUs. The operations include predicates, boolean combinations, and aggregations. We utilize the SIMD capabilities of pixel processing engines within a GPU to perform these operations efficiently. We have used these algorithms for selection queries on one or more attributes and generic aggregation queries including selectivity analysis on large databases.

Our algorithms take into account some of the limitations of the current programming model of GPUs which make it difficult to perform data rearrangement. We present novel algorithms for performing multi-attribute comparisons, semi-linear queries, range queries, computing the kth largest number, and other aggregates. These algorithms have been implemented using fragment programs and have been applied to large databases composed of up to a million records. The performance of these algorithms depends on the instruction sets available for fragment programs, the number of fragment processors, and the underlying clock rate of the GPU. We also perform a preliminary comparison between GPU-based algorithms running on a NVIDIA GeForceFX 5900 Ultra graphics processor and optimized CPU-based algorithms running on dual 2.8 GHz Intel Xeon processors.

We show that algorithms for semi-linear and selection queries map very well to GPUs and we are able to obtain significant performance improvement over CPU-based implementations. The algorithms for aggregates obtain a modest gain of $2-4$ times speedup over CPU-based implementations. Overall, the GPU can be used as an effective co-processor for many database operations.

1.3 Organization

The rest of the paper is organized as follows. We briefly survey related work on database operations and use of GPUs for geometric and scientific computing in Section 2. We give an overview of the graphics architectural pipeline in Section 3. We present algorithms for database operations including predicates, boolean combinations, and aggregations in Section 4. We describe their implementation in Section 5 and compare their performance with optimized CPU-based implementations. We analyze the performance in Section 6 and outline the cases where GPU-based algorithms can offer considerable gain over CPU-based algorithms.

2. RELATED WORK

In this section, we highlight the related research in main-memory database operations and general purpose computation using GPUs.

2.1 Hardware Accelerated Database Operations

Many acceleration techniques have been proposed for database operations. Ailamaki et al. [3] analyzed the execution time of commercial DBMSs and observed that almost half of the time is spent in stalls. This indicates that the performance of a DBMS can be significantly improved by reducing stalls.

Meki and Kambayashi used a vector processor for accelerating the execution of relational database operations including selection, projection, and join [24]. To utilize the efficiency of pipelining and parallelism that a vector processor provides, the implementation of each operation was redesigned for increasing the vectorization rate and the vector length. The limitation of using a vector processor is that the load-store instruction can have high latency [37].

Modern CPUs have SIMD instructions that allow a single basic operation to be performed on multiple data elements in parallel. Zhu and Ross described SIMD implementation of many important database operations including sequential scans, aggregation, indexed searches, and joins [37]. Considerable performance gains were achieved by exploiting the inherent parallelism of SIMD instructions and reducing branch mispredictions.

Recently, Sun et al. present the use of graphics processors for spatial selections and joins [35]. They use color blending capabilities available on graphics processors to test if two polygons intersect in screen-space. Their experiments on graphics processors indicate a speedup of nearly 5 times on intersection joins and within-distance joins when compared against their software implementation. The technique focuses on pruning intersections between triangles based on their 2D overlap and is quite conservative.

2.2 General-Purpose Computing Using GPUs

In theory, GPUs are capable of performing any computation that can be mapped to the stream-computing model. This model has been exploited for ray-tracing [29], global illumination [30] and geometric computations [22].

The programming model of GPUs is somewhat limited, mainly due to the lack of random access writes. This limitation makes it more difficult to implement many data structures and common algorithms such as sorting. Purcell et al. [30] present an implementation of bitonic merge sort, where the output routing from one step to another is known in advance. The algorithm is implemented as a fragment program and each stage of the sorting algorithm is performed as one rendering pass. However, the algorithm can be quite slow for database operations on large databases.

GPUs have been used for performing many discretized geometric computations [22]. These include using stencil buffer hardware for interference computations [33], using depth-buffer hardware to perform distance field and proximity computations [15], and visibility queries for interactive walkthroughs and shadow generation [12].

High throughput and direct access to texture memory makes fragment processors powerful computation engines for certain numerical algorithms, including dense matrix-matrix multiplication [18], general purpose vector processing [36], visual simulation based on coupled-map lattices [13], linear algebra operations [17], sparse matrix solvers for conjugate gradient and multigrid [4], a multigrid solver for boundary value problems [11], geometric computations [1, 16], etc.

3. OVERVIEW

In this section, we introduce the basic functionality available on GPUs and give an overview of the architectural pipeline. More details are given in [9].

3.1 Graphics Pipeline

A GPU is designed to rapidly transform the geometric description of a scene into the pixels on the screen that constitute a final image. Pixels are stored on the graphics card in a *frame-buffer*. The frame buffer is conceptually divided into three buffers according to the different values stored at each pixel:

- **Color Buffer:** Stores the color components of each pixel in the frame-buffer. Color is typically divided into red, green, and blue channels with an alpha channel that is used for blending effects.
- **Depth Buffer:** Stores a depth value associated with each pixel. The depth is used to determine surface visibility.
- **Stencil Buffer:** Stores a stencil value for each pixel. It is called the stencil buffer because it is typically used for enabling/disabling writes to portions of the frame-buffer.

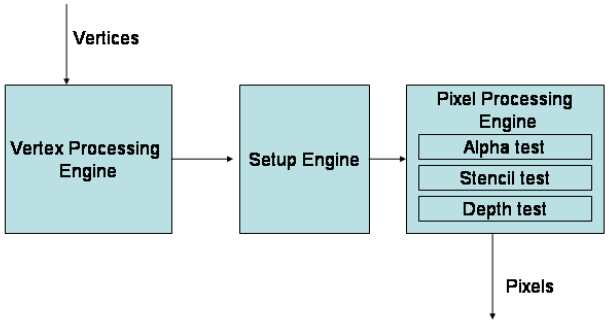

Figure 1: ***Graphics architectural pipeline overview: This figure shows the various units of a modern GPU. Each unit is designed for performing a specific operation efficiently.***

The transformation of geometric primitives (points, lines, triangles, etc.) to pixels is performed by the graphics pipeline, consisting of several functional units, each optimized for performing a specific operation. Fig 1 shows the various stages involved in rendering a primitive.

- **Vertex Processing Engine:** This unit receives vertices as input and transforms them to points on the screen.
- **Setup Engine:** Transformed vertex data is streamed to the setup engine which generates slope and initial value information for color, depth, and other parameters associated with the primitive vertices. This information is used during rasterization for constructing *fragments* at each pixel location covered by the primitive.
- **Pixel Processing Engines:** Before the fragments are written as pixels to the frame buffer, they pass through the pixel processing engines or *fragment processors*. A series of tests can be used for discarding a fragment before it is written to the frame buffer. Each test performs a comparison using a user-specified relational operator and discards the fragment if the test fails.
  - **Alpha test:** Compares a fragment's alpha value to a user-specified reference value.
  - **Stencil test:** Compares the stencil value of a fragment's corresponding pixel with a user-specified reference value.
  - **Depth test:** Compares the depth value of a fragment to the depth value of the corresponding pixel in the frame buffer.

The relational operator can be any of the following : $=$, $<$, $>$, $\leq$, $\geq$, and $\neq$. In addition, there are two operators, *never* and *always*, that do not require a reference value.

Current generations of GPUs have a pixel processing engine that is programmable. The user can supply a custom *fragment program* to be executed on each fragment. For example, a fragment program can compute the alpha value of a fragment as a complex function of the fragment's other color components or its depth.

3.2 Visibility and Occlusion Queries

Current GPUs can perform visibility and occlusion queries [27]. When a primitive is rasterized, it is converted to fragments. Some of these fragments may or may not be written to pixels in the frame buffer depending on whether they pass the alpha, stencil and depth tests. An *occlusion query* returns the *pixel pass count*, the number of fragments that pass the different tests. We use these queries for performing aggregation computations (see Section 4).

3.3 Data Representation on the GPUs

Our goal is to utilize the inherent parallelism and vector processing capabilities of the GPUs for database operations. A key aspect is the underlying data representation.

Data is stored on the GPU as *textures*. Textures are 2D arrays of values. They are usually used for applying images to rendered surfaces. They may contain multiple channels. For example, an RGBA texture has four color channels - red, blue, green and alpha. A number of different data formats can be used for textures including 8-bit bytes, 16-bit integers, and floating point. We store data in textures in the floating-point format. This format can precisely represent integers up to 24 bits.

To perform computations on the values stored in a texture, we render a single quadrilateral that covers the window. The texture is applied to the quadrilateral such that the individual elements of the texture, *texels*, line up with the pixels in the frame-buffer. Rendering the textured quadrilateral causes a fragment to be generated for every data value in the texture. Fragment programs are used for performing computations using the data value from the texture. Then the alpha, stencil, and depth tests can be used to perform comparisons.

3.4 Stencil Tests

Graphics processors use stencil tests for restricting computations to a portion of the frame-buffer based on the value in the stencil buffer. Abstractly, we can consider the stencil buffer as a mask on the screen. Each fragment that enters

the pixel processing engine corresponds to a pixel in the frame-buffer. The stencil test compares the stencil value of a fragment's corresponding pixel against a reference value. Fragments that fail the comparison operation are rejected from the rasterization pipeline.

Stencil operations can modify the stencil value of a fragment's corresponding pixel. Examples of such stencil operations include

- **KEEP:** Keep the stencil value in stencil buffer. We use this operation if we do not want to modify the stencil value.
- **INCR:** Increment the stencil value by one.
- **DECR:** Decrement the stencil value by one.
- **ZERO:** Set the stencil value to zero.
- **REPLACE:** Set the stencil value to the reference value.
- **INVERT:** Bitwise invert the stencil value.

For each fragment there are three possible outcomes based on the stencil and depth tests. Based on the outcome of the tests, the corresponding stencil operation is performed:

- **Op1**: when a fragment fails the stencil test,
- **Op2**: when a fragment passes the stencil test and fails the depth test,
- **Op3**: when the fragment passes the stencil and depth tests.

We illustrate these operations with the following pseudo-code for the StencilOp routine:

StencilOp(Op1, Op2, Op3)

```
if (stencil test passed) /* perform stencil test */
  /*  fragment passed stencil test */
   if(depth test passed) /* perform depth test */
       /* fragment passed stencil and depth test */
       perform Op3 on stencil value
   else
       /* fragment passed stencil test */
       /* but failed depth test */
       perform Op2 on stencil value
   end if
else
  /* fragment failed stencil test */
  perform Op1 on stencil value
end if
```

4. BASIC DATABASE OPERATIONS USING GPUS

In this section, we give a brief overview of basic database operations that are performed efficiently on a GPU. Given a relational table `T` of m attributes $(a_1, a_2, ..., a_m)$, a basic SQL query is in the form of

```
SELECT A
FROM T
WHERE C
```

where `A` may be a list of attributes or aggregations (SUM, COUNT, AVG, MIN, MAX) defined on individual attributes, and `C` is a boolean combination (using AND, OR, EXIST, NOT EXIST) of *predicates* that have the form a_i **op** a_j or a_i **op** *constant*. The operator **op** may be any of the following: $=, \neq, >, \geq, <, \leq$. In essence, queries specified in this form involve three categories of basic operations: predicates, boolean combinations, and aggregations. Our goal is to design efficient algorithms for performing these operations using graphics processors.

- *Predicates*: Predicates in the form of a_i **op** *constant* can be evaluated via the depth test and stencil test. The comparison between two attributes, a_i **op** a_j, can be transformed into a semi-linear query $a_i - a_j$ **op** 0, which can be executed on the GPUs.
- *Boolean combinations*: A boolean combination of predicates can always be rewritten in a conjunctive normal form (CNF). The stencil test can be used repeatedly for evaluating a series of logical operators with the intermediate results stored in the stencil buffer.
- *Aggregations*: This category includes simple operations such as COUNT, SUM, AVG, MIN, MAX, all of which can be implemented using the counting capability of the occlusion queries on GPUs.

To perform these operations on a relational table using GPUs, we store the attributes of each record in multiple channels of a single texel, or the same texel location in multiple textures.

4.1 Predicate Evaluation

In this section, we present novel GPU-based algorithms for performing comparisons as well as the semi-linear queries.

4.1.1 Comparison between an Attribute and a Constant

We can implement a comparison between an attribute "*tex*" and a constant "*d*" by using the depth test functionality of graphics hardware. The stencil buffer can be configured to store the result of the depth test. This is important not only for evaluating a single comparison but also for constructing more complex boolean combinations of multiple predicates.

To use the depth test for performing comparisons, attribute values need to be stored in the depth buffer. We use a simple fragment program for copying the attribute values from the texture memory to the depth buffer.

A comparison operation against a depth value d is implemented by rendering a screen filling quadrilateral with depth d. In this operation, the rasterization hardware uses the comparison function for testing each attribute value stored in the depth buffer against d. The comparison function is specified using the depth function. Routine 4.1 describes the pseudo-code for our implementation.

4.1.2 Comparison between Two Attributes

The comparison between two attributes, a_i **op** a_j, can be transformed into a special semi-linear query ($a_i - a_j$ **op** 0), which can be performed very efficiently using the vector processors on the GPUs. Here, we propose a fast algorithm that can perform any general semi-linear query on GPUs.

```
COMPARE( tex, op, d )
1  CopyToDepth( tex )
2  set depth test function to op
3  RENDERQUAD( d )

CopyToDepth( tex )
1  set up fragment program
2  RenderTexturedQuad( tex )
```

ROUTINE 4.1: COMPARE *compares the attribute values stored in texture* `tex` *against* `d` *using the comparison function* `op`. `CopyToDepth` *called on line* 1 *copies the attribute values in* `tex` *into the depth buffer.* `CopyToDepth` *uses a simple fragment program on each pixel of the screen for performing the copy operation. On line* 2*, the depth test is configured to use the comparison operator* `op`*. The function* RENDERQUAD*(*`d`*) called on line* 3 *generates a fragment at a specified depth* `d` *for each pixel on the screen. Rasterization hardware compares the fragment depth* `d` *against the attribute values in depth buffer using the operation* `op`*.*

Semi-linear Queries on GPUs

Applications encountered in Geographical Information Systems (GIS), geometric modeling, and spatial databases define geometric data objects as linear inequalities of the attributes in a relational database [28]. Such geometric data objects are called semi-linear sets. GPUs are capable of fast computation on semi-linear sets. A linear combination of m attributes is represented as:

$$\sum_{i=1}^{i=m} s_i \cdot a_i$$

where each s_i is a scalar multiplier and each a_i is an attribute of a record in the database. The above expression can be considered as a dot product of two vectors s and a where $s = (s_1, s_2, ..., s_m)$ and $a = (a_1, a_2, ..., a_m)$.

```
SEMILINEAR( tex, s, op, b )
1  enable fragment program SEMILINEARFP(s, b)
2  RenderTexturedQuad( tex )

SEMILINEARFP( s, op, b)
1  a = value from tex
2  if dot( s, a) op b
3    discard fragment
```

ROUTINE 4.2: SEMILINEAR *computes the semi-linear query by performing linear combination of attribute values in tex and scalar constants in s. Using the operator op, it compares the the scalar value due to linear combination with b. To perform this operation, we render a screen filling quad and generate fragments on which the semi-linear query is executed. For each fragment, a fragment program* SEMILINEARFP *discards fragments that fail the query.*

SEMILINEAR computes the semi-linear query:

$$(s \cdot a)\ op\ b$$

where *op* is a comparison operator and b is a scalar constant. The attributes a_i are stored in separate channels in the texture `tex`. There is a limit of four channels per texture. Longer vectors can be split into multiple textures, each with four components. The fragment program SEMILINEARFP() performs the dot product of a texel from `tex` with `s` and compares the result to `b`. It discards the fragment if the comparison fails. Line 2 renders a textured quadrilateral using the fragment program. SEMILINEAR maps very well to the parallel pixel processing as well as vector processing capabilities available on the GPUs. This algorithm can also be extended for evaluating polynomial queries.

```
EVALCNF( A )
1   Clear Stencil to 1.
2   For each of A_i, i = 1, .., k
3     do
4        if ( mod(i, 2) ) /* valid stencil value is 1 */
5             Stencil Test to pass if stencil value is equal to 1
6             StencilOp(KEEP,KEEP,INCR)
7        else /* valid stencil value is 2 */
8             Stencil Test to pass if stencil value is equal to 2
9             StencilOp(KEEP,KEEP,DECR)
10       endif
11       For each B_j^i, j = 1, .., m_i
12       do
13            Perform B_j^i using COMPARE
14       end for
15       if ( mod(i, 2)) /* valid stencil value is 2 */
16            if a stencil value on screen is 1, replace it with 0
17       else /* valid stencil value is 1 */
18            if a stencil value on screen is 2, replace it with 0
19       endif
20  end for
```

ROUTINE 4.3: EVALCNF *is used to evaluate a CNF expression. Initially, the stencil is initialized to* 1*. This is used for performing* $TRUE\ AND\ A_1$*. While evaluating each formula* A_i*, Line* 4 *sets the appropriate stencil test and stencil operations based on whether i is even or odd. If i is even, valid portions on screen have stencil value 2. Otherwise, valid portions have stencil value 1. Lines* $11 - 14$ *invalidate portions on screen that satisfy* $(A_1 \wedge A_2 \wedge ... \wedge A_{i-1})$ *and fail* $(A_1 \wedge A_2 \wedge ... \wedge A_i)$*. Lines* $15 - 19$ *compute the disjunction of* B_j^i *for each predicate* A_i*. At the end of line* 19*, valid portions on screen have stencil value* 2 *if i is odd and* 1*, otherwise. At the end of the line* 20*, records corresponding to non-zero stencil values satisfy A.*

4.2 Boolean Combination

Complex boolean combinations are often formed by combining simple predicates with the logical operators *AND, OR, NOT*. In these cases, the stencil operation is specified to store the result of a predicate. We use the function STENCILOP (as defined in Section 3.4) to initialize the appropriate stencil operation for storing the result in stencil buffer.

Our algorithm evaluates a boolean expression represented as a CNF expression. We assume that the CNF expression has no *NOT* operators. If a simple predicate in this expression has a *NOT* operator, we can invert the comparison operation and eliminate the *NOT* operator. A CNF expression C_k is represented as $A_1 \wedge A_2 \wedge ... \wedge A_k$ where each A_i is represented as $B_1^i \vee B_2^i \vee ... \vee B_{m_i}^i$. Each $B_j^i, j = 1, 2, .., m_i$ is a simple predicate.

The CNF C_k can be evaluated using the recursion $C_k = C_{k-1} \wedge A_k$. C_0 is considered as $TRUE$. We use the pseudocode in routine 4.3 for evaluating C_k. Our approach uses three stencil values $0, 1, 2$ for validating data. Data values corresponding to the stencil value 0 are always invalid. Initially, the stencil values are initialized to 1. If i is the iteration value for the loop in line 2, lines $3 - 19$ evaluate C_i. The valid stencil value is 1 or 2 depending on whether i is even or odd respectively. At the end of line 19, portions on the screen with non-zero stencil value satisfy the CNF C_k. We can easily modify our algorithm for handling a boolean expression represented as a DNF.

Range Queries

A range query is a common database query expressed as a boolean combination of two simple predicates. If $[low, high]$ is the range for which an attribute x is queried, we can evaluate the expression ($x \geq$ low) AND ($x \leq$ high) using EVALCNF. Recent GPUs provide a feature *GL_EXT_Depth_bounds_test* [8], useful in accelerating shadow algorithms. Our algorithm uses this feature for evaluating a range query efficiently. The pseudo-code for our algorithm RANGE is given in Routine 4.4. Although a range query requires the evaluation of two simple predicates, the computational time for our algorithm in evaluating RANGE is comparable to the time required in evaluating a single predicate.

```
Range( tex, low, high )
1  SetupStencil()
2  CopyToDepth( tex )
3  Set depth bounds based on [low, high]
4  Enable depth bounds test
5  RenderQuad(low)
6  Disable depth bounds test
```

ROUTINE 4.4: *`SetupStencil` is called on line 1 to enable selection using the stencil buffer. `CopyToDepth` called on line* 2 *copies the attribute values in `tex` into the depth buffer. Line* 3 *sets the depth bounds based on [low, high]. The attribute values copied into the depth buffer and falling within the depth bounds pass the depth bounds test. Lines* $4-6$ *perform the depth bounds test. The stencil is set to 1 for the attributes passing the range query and 0 for the other.*

4.3 Aggregations

Several database operations aggregate attribute values that satisfy a condition. On GPUs, we can perform these operations using *occlusion queries* to return the count of records satisfying some condition.

4.3.1 COUNT

Using an occlusion query for counting the number of records satisfying some condition involves three steps:

1. Initialize the occlusion query
2. Perform the boolean query
3. Read back the result of the occlusion query into COUNT

4.3.2 MIN and MAX

The query to find the minimum or maximum value of an attribute is a special case of the kth largest number. Here, we present an algorithm to generate the kth largest number.

k-th Largest Number

Computing the k-th largest number occurs frequently in several applications. We can utilize expected linear time selection algorithms such as QUICKSELECT [14] to compute the k-th largest number. Most of these algorithms require data rearrangement, which is extremely expensive on current GPUs because there is no functionality for data writes to arbitrary locations. Also, these algorithms require evaluation of conditionals and may lead to branch mispredictions on the CPU. We present a GPU-based algorithm that does not require data rearrangement. In addition, our algorithm exhibits SIMD characteristics that exploit the inherent parallelism available on the GPUs.

Our algorithm utilizes the binary data representation for computing the k-th largest value in time that is linear in the number of bits.

```
KthLargest( tex, k )
1  b_max = maximum number of bits in the values in tex
2  x = 0
3  for i = b_max-1 down to 0
4      count = Compare( tex, ≥, x +2^i )
5      if count > k - 1
6      x = x +2^i
7  return x
```

ROUTINE 4.5: KTHLARGEST *computes the k-th largest attribute value in texture tex. It uses b_max passes starting from the MSB to compute the k-th largest number. During a pass i, it determines the i-th bit of the k-th largest number. At the end of b_max passes, it computes the k-th largest number in x.*

The pseudocode for our algorithm KTHLARGEST is shown in routine 4.5. KTHLARGEST constructs in `x` the value of the k-th largest number one bit at a time starting with the most significant bit (MSB), `b_max`-1. As an invariant, the value of `x` is maintained less than or equal to the k-th largest value. Line 4 counts the number of values that are greater than or equal to $x + 2^i$, the tentative value of `x` with the ith bit set. This count is used for deciding whether to set the bit in `x` according to the following lemma:

Lemma 1: *Let v_k be the k-th largest number in a set of values. Let count be the number of values greater than or equal to a given value m.*

- if $count > k-1 : m \leq v_k$
- if $count \leq (k-1) : m > v_k$

Proof: Trivial.

If count $> k-1$ then the tentative value of `x` is smaller than the k-th largest number. In this case, we set `x` to the tentative value on line 6. Otherwise the tentative value is too large so we leave `x` unchanged. At the end of line 6, if the loop iteration is i, the first i bits from MSB of `x` and v_k are the same. After the last iteration of the loop, `x` has the value of the k-th largest number. The algorithm for the k-th smallest number is the same, except that the comparison in line 5 is inverted.

4.3.3 SUM and AVG

An accumulator is used to sum a set of data values. One way of implementing an accumulator on current GPUs is using a *mipmap* of a floating point texture. Mipmaps are multi-resolution textures consisting of multiple levels. The highest level of the mipmap contains the average of all the values in the lowest level, from which it is possible to recover the sum by multiplying the average with the number of values. A fragment program must be used to create a floating-point mipmap. Computing a floating-point mipmap on current GPUs tends to be problematic for three reasons. Firstly, reading and writing floating-point textures can be slow. Secondly, if we are interested in the sum of only a subset of values, e.g. those that are greater than a given number, then introduce conditionals in the fragment program. Finally, the floating point representation may not have enough precision to give an exact sum.

Our accumulator algorithm avoids some of the problems of the mipmap method. We perform only texture reads

which are more efficient than texture writes. Moreover, we calculate the precise sum to arbitrary precision and avoid conditionals in the fragment program. One limitation of the algorithm is that it works only on integer datasets, although it can easily be extended to handle fixed-point datasets.

```
Accumulator( tex )
1  alpha test = pass with alpha ≥ 0.5
2  sum = 0
3  for i = 0 to b_max do
4      enable fragment program TestBit(i)
5      initialize occlusion query
6      RenderTexturedQuad( tex )
7      count = pixel count from occlusion query
8      sum + = count ∗2^i
9  return sum

TestBit(i)
1  v = value from tex
2  fragment alpha = frac(v /2^(i+1))
```

ROUTINE 4.6: *Accumulator computes the sum of attribute values in texture tex. It performs b_max passes to compute the sum. Each pass computes the number of values with i-th bit set and stores it in count. This count is multiplied with 2^i and added to sum. At the end of the b_max passes, the variable sum aggregates all the data values in the texture.*

ACCUMULATOR sums the values stored in the texture `tex` utilizing the binary data representation. The sum of the values x_j in a set X can be written as:

$$\sum_{j=0}^{|X|} x_j = \sum_{j=0}^{|X|} \sum_{i=0}^{k} a_{ij} 2^i$$

where $a_{ij} \in \{0, 1\}$ are the binary digits of x_j and k is the maximum number of bits used to represent the values in X. Currently, no efficient algorithms are known for summing the texels on current GPUs. We can, however, quickly determine the number of texels for which a particular bit i is set. If we reverse the order of the summations, we get an expression that is more amenable to GPU computation:

$$\sum_{i=0}^{k} 2^i \left(\sum_{j=0}^{|X|} a_{ij} \right)$$

The inner summation is simply the number of x_j that have the ith bit set. This summation is the value of `count` calculated on lines 4-6 where we render a quad textured with `tex`.

The fragment program TESTBIT ensures that only fragments corresponding to texels with the ith bit set pass the alpha test. Determining whether a particular bit is set is trivial with bit-masking operations. Since current GPUs do not support bit-masking operations in fragment programs, we use an alternate approach. We observe that an integer x has its ith bit equal to 1 if and only if the fractional part of $x/2^{i+1}$ is at least 0.5. In `TestBit`, we divide each value by 2^{i+1} and put the fractional part of the result into the alpha channel. We use the alpha test for rejecting fragments with alpha less than 0.5. It is possible to perform the comparison and reject fragments directly in the fragment program, but it is faster in practice to use the alpha test. Pseudocode for our algorithm is shown in the routine 4.6.

ACCUMULATOR can be used for summing only a subset of the records in `tex` that have been selected using the stencil buffer. Attributes that are not selected fail the stencil test and thus make no contribution to the final sum. We use the ACCUMULATOR algorithm to obtain SUM. AVG is obtained by computing SUM and COUNT, and computed as AVG = SUM/COUNT.

5. IMPLEMENTATION & PERFORMANCE

We have implemented and tested our algorithms on a high end Dell Precision Workstation with dual 2.8GHz Intel Xeon Processors and an NVIDIA GeForceFX 5900 Ultra graphics processor. The graphics processor has 256MB of video memory with a memory data rate of 950MHz and can process upto 8 pixels at processor clock rate of 450 MHz. This GPU can perform single-precision floating point operations in fragment programs.

5.1 Benchmarks

For our benchmarks, we have used a database consisting of TCP/IP data for monitoring traffic patterns in local area network and wide area network and a census database [6] consisting of monthly income information. In the TCP/IP database, there are one million records in the database. In our experiments, each record has 4 attributes, $(data\_count, data\_loss, flow\_rate, retransmissions)$.

Each attribute in the database is stored in as a floating-point number encoded in a 32 bit RGBA texture. The video memory available on the NVIDIA GeForce FX 5900 graphics processor can store more than 50 attributes, each in a texture of size 1000×1000, amounting to a total of 50 million values in the database. We transfer textures from the CPU to the graphics processor using an AGP $8X$ interface.

The census database consists of 360K records. We used four attributes for each record of this database. We have benchmarked our algorithms using the TCP/IP database. Our performance results on the census data are consistent with the results obtained on the TCP/IP database.

5.2 Optimized CPU Implementation

We implemented the algorithms described in section 4 and compared them with an optimized CPU implementation. We compiled the CPU implementation using Intel compiler 7.1 with full compiler optimizations [1]. These optimizations include

- **Vectorization:** The compiler detects sequential data scans and generates code for SIMD execution.
- **Multi-threading:** We used the compiler switch `-QParallel` to detect loops which may benefit from multi-threaded execution and generate appropriate threading calls. This option enables the CPU implementation to utilize hyper-threading technology available on Xeon processors.
- **Inter-Procedural Optimization (IPO):** The compiler performs function inlining when IPO is enabled. It reduces the function call branches, thus improving its efficiency.

For the timings, we ran each of our tests 100 times and computed the average running time for the test.

[1] `http://www.intel.com/software/products/compilers/techtopics/compiler_optimization_71.pdf`

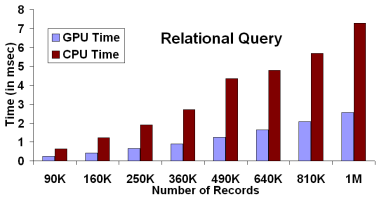

Figure 3: *Execution time of a predicate evaluation with* 60% *selectivity by a CPU-based and a GPU-based algorithm. Timings for the GPU-based algorithm include time to copy data values into the depth buffer. Considering only computation time, the GPU is nearly 20 times faster than a compiler-optimized SIMD implementation.*

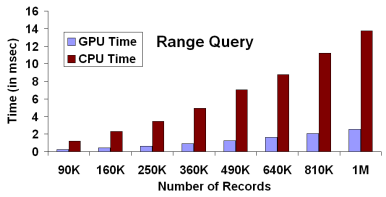

Figure 4: *Execution time of a range query with* 60% *selectivity using a GPU-based and a CPU-based algorithm. Timings for the GPU-based algorithm include time to copy data values into the depth buffer. Considering only computation time, the GPU is nearly* 40 *times faster than a compiler-optimized SIMD implementation.*

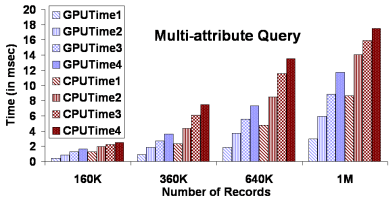

Figure 5: *Execution time of a multi-attribute query with* 60% *selectivity for each attribute and a combination of AND operator. Time$_i$ is the time to perform a query with i attributes. We show the timings for CPU and GPU-based implementations.*

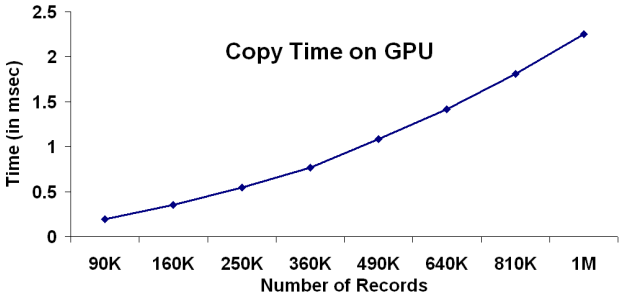

Figure 2: *Plot indicating the time taken for copying data values in a texture to the depth buffer.*

5.3 GPU Implementation

Our algorithms described in Section 4 are implemented using the OpenGL API. For generating the fragment programs, we used NVIDIA's CG compiler [23]. As the code generated by the compiler is often sub-optimal, we examined the assembly code generated by the current compiler and reduced the number of assembly instructions to perform the same operation.

For the counting operations, we chose to use *GL_NV_occlusion_query* for image-space occlusion queries. These queries can be performed asynchronously and often do not add any additional overhead.

5.4 Copy Operation

Various database operations, such as comparisons, selection, etc, require the data values of an attribute stored in the depth buffer. For these operations, we copy the corresponding texture into the depth buffer. A fragment program is used to perform the copy operation. Our copy fragment program implementation requires three instructions.

1. Texture Fetch: We fetch the texture value corresponding to a fragment.

2. Normalization: We normalize the texture value to the range of valid depth values $[0, 1]$.

3. Copy To Depth: The normalized value is copied into the fragment depth.

Figure 2 shows the time taken to copy values from textures of varying sizes into the depth buffer. The figure indicates an almost linear increase in the time taken to perform the copy operation as a function of the number of records. In the future, it may be possible to copy data values from textures directly to a depth buffer and that would reduce these timings considerably. Also, the increase in clock rates of graphics processors and improved optimizations to perform depth buffer writes [26] could help in reducing these timings.

5.5 Predicate Evaluation

Figure 3 shows a plot of the time taken to compute a single predicate for an attribute such that the selectivity is 60%. In our experiments, we performed the operation on the first attribute of each record in the database. The plot compares a compiler-generated SIMD optimized CPU code against a simple GPU implementation. The GPU timings include the computational time for evaluating the predicate, as well as the time taken to copy the attribute into the depth buffer. We observe that the GPU timings are nearly 3 times faster than the CPU timings. If we compare only the computational time on the GPU, we observe that the GPU implementation is nearly 20 times faster than the SIMD optimized CPU implementation.

5.6 Range Query

We tested the performance of RANGE by timing a range query with 60% selectivity. To ensure 60% selectivity, we set the valid range of values between the 20th percentile and 80th percentile of the data values. Again, in our tests, we used the *data_count* as our attribute. Figure 4 compares the time taken for a simple GPU implementation and a compiler-optimized SIMD implementation on CPU. In the GPU timings, we included the time taken for the copy operation. We observe that overall the GPU is nearly 5.5 times faster than the CPU implementation. If we consider only the computational time on GPU and CPU, we observe that the GPU is nearly 40 times faster than the compiler optimized CPU implementation.

5.7 Multi-Attribute Query

We have tested the performance of our hardware-based multi-attribute queries by varying the number of attributes and also the number of records in the database. We used queries with a selectivity of 60% for each attribute and applied the *AND* operator on the result for each attribute. In

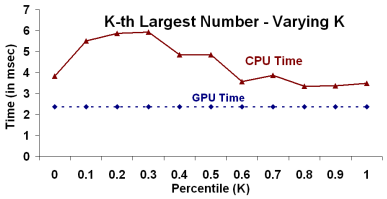

Figure 7: *Time to compute k-th largest number on the data count attribute. We used a portion of the TCP/IP database with nearly* $250K$ *records.*

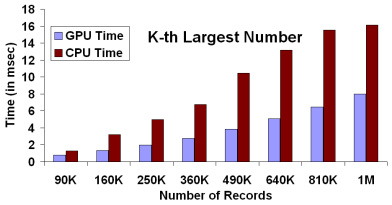

Figure 8: *Time taken to compute the median using* **KthLargest** *and* **QuickSelect** *on varying number of records.*

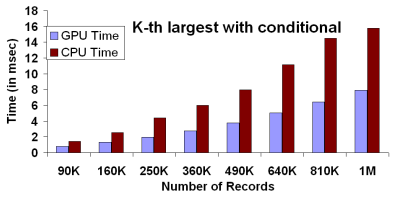

Figure 9: *Time taken to compute the K-th largest number by the two implementations.*

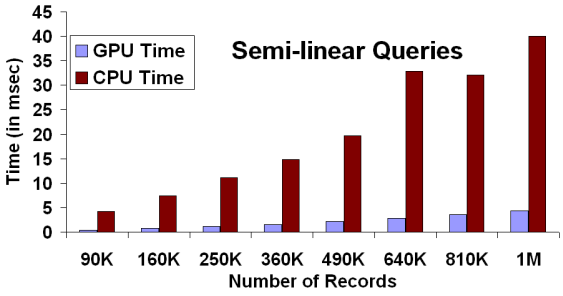

Figure 6: *Execution time of a semi-linear query using four attributes of the TCP/IP database. The GPU-based implementation is almost one order of magnitude faster than the CPU-based implementation.*

our tests, we used up to four attributes per query. For each attribute in the query, the GPU implementation copies the data values from the attribute's texture to the frame-buffer. Figure 5 shows the time taken by the GPU and CPU respectively, to perform multi-attribute queries with the varying number of attributes and records. The timings indicate that the GPU implementation is nearly 2 times faster than the CPU implementation. If we consider only the computational times on the GPU by ignoring copy times, we observe that the GPU is nearly 20 times faster than the optimized CPU implementation.

5.8 Semi-linear Query

We performed a semi-linear query on the four attributes by using a linear combination of four random floating-point values and compared it against an arbitrary value. Figure 6 summarize our timings for various number of tests on GPU and CPU. In our tests, we observe that the GPU timings are 9 times faster than an optimized CPU implementation.

5.9 K-th Largest Number

We performed three different tests to evaluate our KthLargest algorithm on GPU. In each of these tests, we compared KthLargest against a CPU implementation of the algorithm QuickSelect [14]. In our experiments, we used the *data_count* attribute. This attribute requires 19 bits to represent the largest data value and has a high variance.

Test 1: Vary k and compute the time taken for KthLargest and QuickSelect. The tests were performed on 250K records in the database. Figure 7 shows the timings obtained using each of the implementations. This plot indicates that time taken by KthLargest is constant irrespective of the value of k and is an interesting characteristic of our algorithm. On an average, the GPU timings for our algorithm are nearly twice as fast in comparison to the CPU implementation. It should be noted that the GPU timings include the time taken to copy values into the depth buffer. Comparing the computational times, we note that the average KthLargest timings are 3 times faster than QuickSelect.

Test 2: In these tests, we compared the time taken by KthLargest and QuickSelect to compute a median of a varying number of records. Figure 8 illustrates the results of our experiments. We observe that the KthLargest on the GPU is nearly twice as fast as QuickSelect on the CPU. Considering only the computational times, we observe that KthLargest is nearly 2.5 times faster than QuickSelect.

Test 3: We also compared the time taken by KthLargest and QuickSelect for computing a median with on data values with 80% selectivity. Figure 9 indicates the time taken by KthLargest and QuickSelect in computing the median as a function of the number of records. Our timings indicate that KthLargest with 80% selectivity requires exactly the same amount of time as performing KthLargest with 100% selectivity. We conclude from this observation that the use of a conditional to test for valid data has almost no effect on the running time of KthLargest. For the CPU timings, we have copied the valid data into an array and passed it as a parameter to QuickSelect. The timings indicate that the total running time is nearly the same as that of running QuickSelect with 100% selectivity.

5.10 Accumulator

Figure 10 demonstrates the performance of Accumulator on the GPU and a compiler-optimized SIMD implementation of accumulator on the CPU. Our experiments indicate that our GPU algorithm is nearly 20 times slower than the CPU implementation, when including the copy time. Note that the CPUs have a much higher clock rate as compared to the GPU.

5.11 Selectivity Analysis

Recently, several algorithms have been designed to implement join operations efficiently using selectivity estimation [7, 10]. We compute the selectivity of a query using the COUNT algorithm (Section 4.3). To obtain the selectivity count, image-space occlusion queries are used. We performed the experiments described in Sections 5.5, 5.6, 5.7, 5.8. We observed that there is no additional overhead in obtaining the count of selected queries. Given selected data values scattered over a 1000×1000 frame-buffer, we

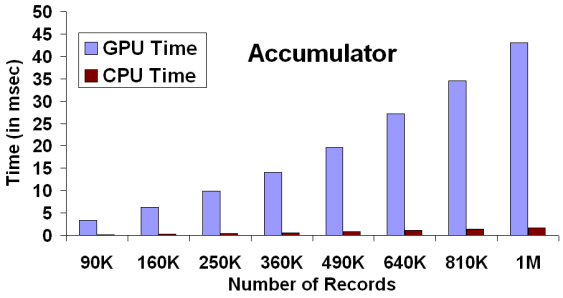

Figure 10: ***Time required to sum the values of an attribute by the CPU and by the GPU-based Accumulator algorithm***

can obtain the number of selected values within 0.25 ms.

6. ANALYSIS

In the previous section, we have highlighted the performance of our algorithms on different database operations and performed a preliminary comparison with some CPU-based algorithms. In this section, we analyze the performance of our algorithms and highlight the database operations for which the GPUs can offer considerable gain in performance.

6.1 Implementing Basic Operations on GPUs

There are many issues that govern the performance of the algorithms implemented on a GPU. Some of the upcoming features in the next generation GPUs can improve the performance of these algorithms considerably.

Precision: Current GPUs have depth buffers with a maximum of 24 bits. This limited precision can be an issue. With the increasing use of GPUs in performing scientific computing, graphics hardware developers may add support for higher-precision depth buffers.

Copy Time: Several of our algorithms require data values to be copied from the texture memory to the depth buffer. Current GPUs do not offer a mechanism to perform this operation efficiently and this operation can take a significant fraction of the overall algorithm (e.g. algorithms for relational and range queries). In the future, we can expect support for this operation on GPUs which could improve the overall performance.

Integer Arithmetic Instructions: Current GPUs do not offer integer arithmetic instructions in the pixel processing engines. In addition to database operations, several image and video compression algorithms also require the use of integer arithmetic operations. Fragment programs were introduced in just the last few years. The instruction sets for these programs are still being enhanced. The instructions for integer arithmetic would reduce the timings of our ACCUMULATOR algorithm significantly.

Depth Compare Masking: Current GPUs support a boolean depth mask that enables or disables writes to a depth buffer. It is very useful to have a comparison mask specified for the depth function, similar to that specified in the stencil function. Such a mask would make it easier to test if a number has i-th bit set.

Memory Management: Current high-end GPUs have up to 512MB of video memory and this limit is increasing every year. However, due to the limited video memory, we may not be able to copy very large databases (with tens of millions of records) into GPU memory. In such situations, we would use out-of-core techniques and swap textures in and out of video memory. Another related issue is the bus bandwidth. Current PCs use an AGP8× bus to transfer data from the CPU to the GPU and the PCI bus from the GPU to the CPU. With the announcement of PCI-EXPRESS bus, the bus bandwidth is going to improve significantly in the near future. Asynchronous data transfers would also improve the performance of these algorithms.

No Branching: Current GPUs implement branching by evaluating both portions of the conditional statement. We overcome this issue by using multi-pass algorithms and evaluating the branch operation using the alpha test or the depth test.

No Random Writes: GPUs do not support random access writes, which makes it harder to develop algorithms on GPUs because they cannot use data rearrangement on GPUs.

6.2 Relative Performance Gain

We have presented algorithms for predicates, boolean combinations and aggregations. We have also performed preliminary comparison with optimized CPU-based implementations on a workstation with dual 2.8 GHz Xeon processors. Due to different clock rates and instruction sets, it is difficult to perform explicit comparisons between CPU-based and GPU-based algorithms. However, some of our algorithms perform better than others. We classify our algorithms into three categories: high performance gain, medium performance gain and low performance gain.

6.2.1 High Performance Gain

In these algorithms, we have observed an order of magnitude speedup over CPU-based implementations. These include algorithms for semi-linear queries and selection queries. The main reason for the improved performance are:

- **Parallel Computation:** GPUs have several pixel processing engines that process multiple pixels in parallel. For example, on a GeForce FX 5900 Ultra we can process 8 pixels in parallel and reduce the computational time significantly. Also, each pixel processing engine has vector processing capabilities and can perform vector operations very efficiently. The speedup in selection queries is mainly due to the parallelism available in pixel processing engines. The semi-linear queries also exploit the vector processing capabilities.

- **Pipelining:** GPUs are designed using a pipelined architecture. As a result, they can simultaneously process multiple primitives within the pipeline. The algorithms for handling multiple-attribute queries map well to the pipelined implementation.

- **Early Depth-Culling:** GPUs have specialized hardware that early in the pipeline can reject fragments that will not pass the depth test. Since the fragments do not have to pass through the pixel processing engines, this can lead to a significant performance increase.

- **Eliminate branch mispredictions:** One of the major advantages in performing these selection queries on GPUs is that there are no branch mispredictions.

Branch mispredictions can be extremely expensive on the modern CPUs. Modern CPUs use specialized schemes for predicting the outcome of the branch instruction. Each branch mis-prediction can cost several clock cycles on current CPUs. For example, on a Pentium IV a branch misprediction can have a penalty of 17 clock cycles [25].

6.2.2 Medium Performance Gain

Several of our algorithms for database operations are only able to use a subset of the capabilities of the GPUs. In these cases, we have observed a speedup of nearly a factor of 2 to 4 times in comparison to an optimized-CPU implementation. For example, the KthLargest() routine exhibits these characteristics. The speedup in the KthLargest() is mainly due to the parallelism available in pixel processing engines. Given the GPU clock rate and the number of pixel processing engines, we can estimate the time taken to perform KthLargest() under some assumptions. We assume that there is no latency in the graphics pipeline and in transmitting the pixel pass count from the GPU to the CPU. On a GeForce FX 5900 Ultra with clock rate 450MHz and processing 8 pixels per clock cycle, we can render a single quad of size 1000×1000 in 0.278 ms. In our experiments, we render 19 such quads to compute the k-th largest number. Rendering these quads should take 5.28 ms. The observed time for this computation is 6.6 ms, which indicates that we are utilizing nearly 80% of the parallelism in the pipeline. The observed timings are slightly higher due to the latencies in transmitting the data from the GPU to the CPU and vice-versa. A key advantage of our algorithm KthLargest() in comparison with other parallel order statistic algorithms is that it does not require any data rearrangement. Data rearrangements can be expensive when combined with branching.

6.2.3 Low Performance Gain

In some cases, we did not observe any gain over a CPU-based implementation. Our GPU based Accumulator algorithm is much slower than the CPU-based implementation. There are several reasons for this performance:

- **Lack of Integer Arithmetic:** Current GPUs do not support integer arithmetic instructions. Therefore, we used a fragment program with at least 5 instructions to test if the i-th bit of a texel is 1. There are several ways to implement such a feature in the hardware. A simplest mechanism is to copy the i-th bit of the texel into the alpha value of a fragment. This can lead to significant improvement in performance.
- **Clock Rate:** Not only are we comparing two architectures with different instruction sets, but they also have different clock rates. Our CPU implementation used top-of-the-line dual Xeon processors operating at 2.8GHz. Each Xeon processor has four SIMD processors that can perform four operations in parallel. On the other hand, the current GPU clock rate (450MHz) is much lower than the CPU clock rate. It is possible that the increasing parallelism in the GPUs and development of new instruction sets for fragment programs can bridge this gap in performance.

Our preliminary analysis indicates that it is advantageous to perform selection and semi-linear queries on GPUs. In addition, GPUs can also be used effectively to perform order statistics algorithms.

7. CONCLUSIONS AND FUTURE WORK

In this paper, we have presented novel algorithms for performing database operations on the GPUs. These include algorithms for predicates, boolean combinations, and aggregation queries. We have implemented these algorithms on a state-of-the-art GPU and highlighted its performance for following queries: relational query, range query, multi-attribute query, semi-linear query, kth-largest number computation, accumulator and selectivity analysis. We have also performed preliminary comparisons with optimized implementations of CPU-based algorithms. In some cases, we observed noticeable performance gain.

We have shown that GPUs are excellent candidates for performing some of the databases operations. Some reasons for this finding include:

- **Commodity Hardware:** High-end GPUs are freely available on PCs, consoles and workstations. The cost of a high-end GPU is typically less than the cost of a high-end CPU (by a factor of two or more).
- **Higher Growth Rate:** Over the last decade the growth rate of GPU performance has been higher than that of CPUs. This trend is expected to continue for the next five years or so. The performance gap between the CPUs and GPUs will probably increase considerably and we can obtain improved performance for database queries on the GPUs.
- **Multiple Fragment Processors and Improved Programmability:** Current high-end GPUs already have 8 fragment processors. This number is expected to increase in the future. As the instruction sets of fragment programs improve, the running time of many of our algorithms will further decrease.
- **Useful Co-Processor:** Overall, the GPU can be used as an effective co-processor along with the CPUs. It is clear that GPU is an excellent candidate for some database operations, but not all. Therefore, it would be useful for database designers to utilize GPU capabilities alongside traditional CPU-based code.

There are many avenues for future work. It is possible to use new capabilities and optimizations to improve the performance of many of our algorithms. Furthermore, we would like to develop algorithms for other database operations and queries including sorting, join, and indexed search, and OLAP and data mining tasks such as data cube roll up and drill-down, classification, and clustering, which may benefit from multiple fragment processors and vector processing capabilities. We also plan to design GPU-based algorithms for queries on more complicated data types in the context of spatial and temporal databases and perform continuous queries over streams using GPUs.

Acknowledgements

This research is supported in part by ARO Contract DAAD19-99-1-0162, NSF award ACI-0118743, ONR Contracts N00014-01-1-0067 and N00014-01-1-0496, and Intel Corporation. We would like to thank NVIDIA Corporation especially Steven

Molnar, Paul Keller and Stephen Ehmann for their support. We would also like to thank Jasleen Sahni for providing access to the TCP/IP database.

8. REFERENCES

[1] Pankaj Agarwal, Shankar Krishnan, Nabil Mustafa, and Suresh Venkatasubramanian. Streaming geometric optimization using graphics hardware. In *11th European Symposium on Algorithms*, 2003.

[2] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis. Weaving relations for cache performance. In *Proceedings of the Twenty-seventh International Conference on Very Large Data Bases 2001*, pages 169–180, 2001.

[3] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. DBMSs on a modern processor: Where does time go? In *Proc. of VLDB*, pages 266–277, 1999.

[4] J. Bolz, I. Farmer, E. Grinspun, and P. Schröder. Sparse matrix solvers on the gpu: Conjugate gradients and multigrid. *ACM Trans. on Graphics (Proc. of ACM SIGGRAPH)*, 22(3), 2003.

[5] Peter A. Boncz, Stefan Manegold, and Martin L. Kersten. Database architecture optimized for the new bottleneck: Memory access. In *Proc. of VLDB*, pages 54–65, 1999.

[6] Census bureau databases. http://www.bls.census.gov/cps/.

[7] Zhiyuan Chen, Nick Koudas, Flip Korn, and S. Muthukrishnan. Selectively estimation for boolean queries. In *Proceedings of the nineteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 216–225, 2000.

[8] Ext_depth_bounds_test specification. http://www.nvidia.com/dev_content/nvopenglspecs/GL_EXT_depth_bounds_test.txt.

[9] Michael Doggett. Programmability features of graphics hardware. *ACM SIGGRAPH Course Notes # 11*, 2003.

[10] Lise Getoor, Benjamin Taskar, and Daphne Koller. Selectivity estimation using probabilistic models. In Timos Sellis and Sharad Mehrotra, editors, *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data 2001, Santa Barbara, California, United States, May 21–24, 2001*, pages 461–472, 2001. ACM order number 472010.

[11] N. Goodnight, C. Woolley, G. Lewin, D. Luebke, and G. Humphreys. A multigrid solver for boundary value problems using programmable graphics hardware. *ACM SIGGRAPH/Eurographics Conference on Graphics Hardware*, 2003.

[12] N. Govindaraju, B. Lloyd, S. Yoon, A. Sud, and D. Manocha. Interactive shadow generation in complex environments. *Proc. of ACM SIGGRAPH/ACM Trans. on Graphics*, 22(3):501–510, 2003.

[13] M. Harris, G. Coombe, G. Scheuermann, and A. Lastra. Physically-based visual simulation on graphics hardware. *SIGGRAPH/Eurographics Workshop on Graphics Hardware*, 2002.

[14] C. A. R. Hoare. Algorithm 65: find. *Commun. ACM*, 4(7):321–322, 1961.

[15] K. Hoff, T. Culver, J. Keyser, M. Lin, and D. Manocha. Fast computation of generalized voronoi diagrams using graphics hardw are. *Proceedings of ACM SIGGRAPH*, pages 277–286, 1999.

[16] S. Krishnan, N. H. Mustafa, and S. Venkatasubramanian. Hardware-assisted computation of depth contours. In *13th ACM-SIAM Symposium on Discrete Algorithms*, 2002.

[17] J. Kruger and R. Westermann. Linear algebra operators for gpu implementation of numerical algorithms. *ACM Trans. on Graphics (Proc. of ACM SIGGRAPH)*, 22(3), 2003.

[18] E. S. Larsen and D. K. McAllister. Fast matrix multiplies using graphics hardware. *Proc. of IEEE Supercomputing*, 2001.

[19] M. Macedonia. The gpu enters computing's mainstream. *Computer*, October 2003.

[20] S. Manegold, P. Boncz, and M L. Kersten. What happens during a join? Dissecting CPU and memory optimization effects. In *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, September 10–14, 2000, Cairo, Egypt*, pages 339–350, 2000.

[21] Stefan Manegold, Peter A. Boncz, and Martin L. Kersten. Generic database cost models for hierarchical memory systems. In *Proceedings of the Twenty-eighth International Conference on Very Large Data Bases 2002*, pages 191–202, 2002.

[22] D. Manocha. *Interactive Geometric and Scientific Computations using Graphics Hardware*. SIGGRAPH Course Notes # 11, 2003.

[23] William R. Mark, R. Steven Glanville, Kurt Akeley, and Mark J. Kilgard. Cg: A system for programming graphics hardware in a c-like language. In *Proc. of ACM SIGGRAPH*, 2003. http://developer.nvidia.com/page/cg_main.html.

[24] Shintaro Meki and Yahiko Kambayashi. Acceleration of relational database operations on vector processors. *Systems and Computers in Japan*, 31(8):79–88, August 2000.

[25] http://lava.cs.virgina.edu/bpred.html.

[26] Nvidia geforce fx gpus: Intellisample technology. http://www.nvidia.com/object/intellisample_tb.html.

[27] Nv_occlusion_query specification. http://www.nvidia.com/dev_content/nvopenglspecs/GL_NV_occlusion_query.txt.

[28] Niki Pissinou. Towards and infrastructure for temporal databases — A workship report. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 23(1):35–51, March 1994.

[29] T. Purcell, I. Buck, W. Mark, and P. Hanrahan. Ray tracing on programmable graphics hardware. *ACM Trans. on Graphics (Proc. of SIGGRAPH'02)*, 21(3):703–712, 2002.

[30] T. Purcell, C. Donner, M. Cammarano, H. Jensen, and P. Hanrahan. Photon mapping on programmable graphics hardware. *ACM SIGGRAPH/Eurographics Conference on Graphics Hardware*, pages 41–50, 2003.

[31] Jun Rao and Kenneth A. Ross. Cache conscious indexing for decision-support in main memory. In *Proc. of VLDB*, pages 78–89, 1999.

[32] Kenneth A. Ross. Conjunctive selection conditions in main memory. In ACM, editor, *Proceedings of the Twenty-First ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems: PODS 2002*, pages 109–120, 2002. ACM order number 475021.

[33] J. Rossignac, A. Megahed, and B.D. Schneider. Interactive inspection of solids: cross-sections and interferences. In *Proceedings of ACM Siggraph*, pages 353–60, 1992.

[34] Ambuj Shatdal, Chander Kant, and Jeffrey F. Naughton. Cache conscious algorithms for relational query processing. In *20th International Conference on Very Large Data Bases, 1994*, pages 510–521, 1994.

[35] Chengyu Sun, Divyakant Agrawal, and Amr El Abbadi. Hardware acceleration for spatial selections and joins. In ACM, editor, *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, pages 455–466, 2003.

[36] C. J. Thompson, S. Hahn, and M. Oskin. Using modern graphics architectures for general-purpose computing: A framework and analysis. *Proc. of IEEE/ACM International Symposium on Microarchitectures*, pages 306–317, 2002.

[37] Jingren Zhou and Kenneth A. Ross. Implementing database operations using simd instructions. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 145–156. ACM Press, 2002.