Efficient Relational Database Management using Graphics Processors

Naga K. Govindaraju Dinesh Manocha

University of North Carolina at Chapel Hill

{naga, dm}@cs.unc.edu
http://gamma.cs.unc.edu/DB

ABSTRACT

We present algorithms using graphics processing units (GPUs) to efficiently perform database management queries. Our algorithms use efficient data memory representations and storage models on GPUs to perform fast database computations. We present relational database algorithms that successfully exploit the high memory bandwidth and the inherent parallelism available in GPUs. We implement these algorithms on commodity GPUs and compare their performance with optimized CPU-based algorithms. We show that the GPUs can be used as a co-processor to accelerate many database and data mining queries.

1. INTRODUCTION

The challenge of real-time data analysis pushes the limits of database systems. Due to the intense data processing nature, current database management systems may not be able to fully exploit the computing power of current CPUs. A major limitation arises from accessing the data through different memory hierarchies. Furthermore, database applications such as network traffic analysis and financial markets need to analyze gigabytes of data in a short time frame. The system performance remains a key issue due to the high volume of data that needs to be processed. According to Mike Stonebraker, "Relational databases are one to two orders of magnitude too slow", especially for data streaming applications. Recently, there has been considerable work on analyzing the influence of the processor and memory hierarchies on database algorithms [2, 4, 19]. The initial results seem to indicate that there is substantial memory and computational overhead due to cache misses, branch mispredictions etc. Many algorithms have been proposed to improve the performance including data parallel algorithms, cache conscious data structures and algorithms, instruction buffering algorithms and better storage models.

In this paper, we make a case for using the commodity graphics processors (GPUs) for relational database management systems. High performance GPUs are now an integral part of every personal computer, console and handheld device, including cell phones. Due to increasing programmability, the GPUs have evolved into powerful processors that are flexible enough to perform general purpose computations [16]. The inherent parallelism and the high memory bandwidth within the GPUs has been used to accelerate many of the traditional algorithms by an order of magnitude as compared to the CPU-based implementations. For example, a high-end GPU such as NVIDIA GeForce FX 6800 Ultra has a peak performance of 48 GFlops and a peak memory bandwidth of 35.2 GBps. In contrast, a 3.4 GHz Pentium IV CPU is capable of a peak performance of 6.8 GFlops and the DDRII main memories on PCs have a peak memory bandwidth of 6.4 GBps. Furthermore, the computational power of GPUs is progressing at a rate faster than the Moore's law for the CPUs.

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 use these algorithms to perform for selection queries on one or more attributes, generic aggregation queries and join queries on large databases. We also perform a preliminary comparison between the GPU-based algorithms running on two high end GPUs: NVIDIA GeForceFX 5900 Ultra and NVIDIA GeForceFX 6800 Ultra GPU and optimized CPU-based algorithms running on a 2.8 GHz Intel Xeon processor. Our algorithms indicate a $5 - 20$ times performance improvement over the CPU-based algorithms.

Organization: The rest of the paper is organized as follows. We briefly describe the storage models on CPUs and the mapping of the storage models and data representation of attributes on GPUs in Section 2. In Section 3, we present and evaluate our algorithms for database operations including predicates, boolean combinations, aggregations, and join queries. Finally, we highlight many areas for future research in Section 4.

2. DATABASE STORAGE

In this section, we briefly describe the data representation and storage models for CPUs for performing relational database operations. We discuss the storage issues on CPUs, and present efficient storage models on GPUs. We evaluate the memory access performance on high-end GPUs and the growth rate of memory operations on GPUs in the recent years.

2.1 Storage Models on CPUs

The workload characterization studies on modern DBMS systems indicate a substantial memory overhead on CPUs and appropri-

Proceedings of the First International Workshop on Data Management on New Hardware (DaMoN 2005); June 12, 2005, Baltimore, Maryland, USA.

ate data placement is becoming increasingly important to achieve high performance on commodity PCs [1]. The memory overhead is mainly due to the delay incurred in transferring data from main memory to the cache and is observed in many memory intensive database operations such as join queries [3, 5, 17]. Recent studies also indicate that $50 - 90\%$ of the memory stalls are due to cache misses [1]. Moreover, the size of the CPU cache is small (typically $<$ 2MB), and may not be able to store large relations. Also, the data placement scheme used to store the data can be quite different from the order in which the data is accessed. Therefore, only a small fraction of the data transferred to the CPU cache is used in the query.

In order to improve the performance, different data placement schemes are used in modern database management systems [1, 4]. These can be mainly classified into three categories:

- **N-ary Storage Model (NSM):** NSM stores records contiguously from the beginning of each disk page, and uses an offset at the end of the page to locate the beginning of each record [18]. In many queries, only a small fraction of these records are used, and due to the small CPU cache sizes, it is possible to replace useful data in the CPU cache.

- **Decomposition Storage Model (DSM):** DSM decomposes an n-attribute relation into n sub-relations [6]. Due to the partitioning of the attributes, DSM stores data efficiently in the CPU cache. These algorithms are also well-suited for compression [8]. However, in multi-attribute queries, there is an additional overhead of joining the sub-relations.

- **Partition Attributes Across (PAX) Storage Model:** The PAX storage model [1] uses a combination of DSM and NSM, and improves the CPU cache performance. The storage model is similar to NSM, and stores the records in the same page as obtained using NSM. However, within each page, PAX groups the values of the attributes together into minipages and is similar to DSM.

2.2 Storage Model on GPUs

GPUs have a higher memory bandwidth than the main memories on PCs and our goal is to exploit it for database operations. However, the memory representation on a GPU is highly optimized for rasterizing geometric primitives. In order to achieve a high memory bandwidth ($\sim$ 35 GBps), the GPU memory representations such as 2D textures are significantly different from the main memory representations on the CPUs. We utilize the high memory bandwidth on GPUs to perform fast database operations by mapping the storage models on CPUs to optimized representations on GPUs. The mapping can be performed efficiently using the texture mapping hardware on the GPUs.

Given any storage model on CPUs, we represent the data values in 2D textures. 2D textures are conceptually 2D arrays of values, and 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 bytes, 16-bit integers and IEEE 32-bit floating point values. The memory bandwidth in accessing the data varies with the underlying texture format. Fig. 1 highlights the floating point memory bandwidth obtained on a high-end GPU using a GPU benchmarking utility [12]. In order to obtain maximum performance, we represent the data using a single channel IEEE 32-bit floating point texture.

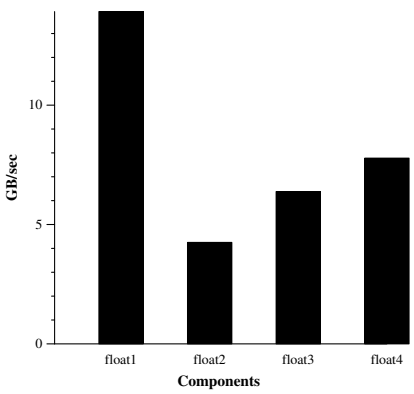

Figure 1: ***Floating point bandwidth on an NVIDIA GeForce FX 6800 Ultra GPU: This graph indicates the memory bandwidth obtained on accessing data values in a single, two, three, and four channel floating point textures using GPUBench. The graph also indicates the poor memory performance in accessing data values in textures with more than a single channel.***

To perform computations on the attributes in the texture, we render a single quadrilateral and generate a fragment for each attribute value. Given any storage format (NSM, DSM or PAX), the texture mapping hardware is used to map appropriate data values in the texture to the fragments generated during the rasterization of the quadrilateral. We then use fragment programs to perform computations using the attribute values in the texture. We use alpha test along with depth tests to perform comparisons, blending operations to evaluate conditional assignments, and stencil tests to encode the predicated output. For a detailed overview of the graphics pipeline, refer to [9, 11].

2.3 Data Representation

GPUs have specialized hardware to efficiently evaluate comparison operations on data values stored in the depth buffer. In the depth buffer, data is represented using 24-bit fixed point format, and is quite different from the IEEE 32-bit floating point format. In order to utilize the entire precision for data stored in the depth buffer, we map the mantissa of the floating point representation to a fixed point format.

The fixed point format is used to store data values in the range [0,1], and the value 0xffffff maps to 1. Therefore, in order to convert the mantissa m into 24-bit fixed point format m^{fixed}, we apply an offset and a scale as follows:

$$m^{fixed} = (m + 1) * scale,$$

where $scale = 0xffffff * \{\frac{1.0}{1 \ll 24}\}^2$. Using our mapping, we utilize the full 24-bit precision of the depth buffer.

2.4 Copy Operation

Many relational database operations such as predicates, selectivity, etc., require the data values to be placed in the depth buffer. We use the copy operation to copy the attribute values in the texture to the depth buffer. Current GPUs support a specialized instruction *unpack* to compute the mantissa, and the exponent of a floating point value in the fragment program, and the mantissa can be copied into

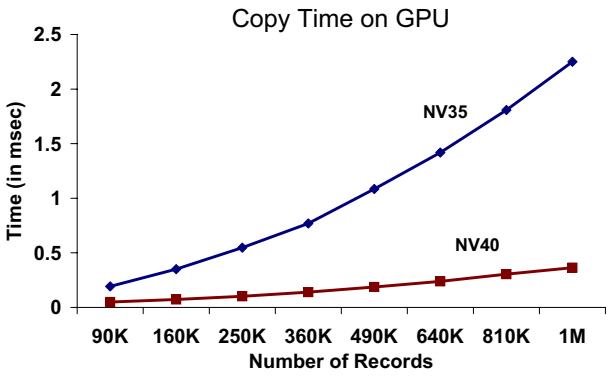

Figure 2: ***Memory Performance: This graph shows the time taken to copy attribute values into the depth buffer of a GPU. The graph indicates that the performance of the copy operation improves over a factor of 5 during the last one year.***

the depth buffer. Our copy fragment program uses 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.

```
CopyToDepth( tex )
1 set up fragment program
2 RenderTexturedQuad( tex )
Copy Fragment Program
TEX R0, f[TEX0].xyzw, TEX0, RECT;
PK4UB R0.x, R0;
MOVR o[DEPR].z, R0.x;
END
```

ROUTINE 2.1: *`CopyToDepth` uses a simple fragment program to copy the normalized attributes in the texture into the depth buffer.*

2.5 Memory Performance

Data values are stored efficiently in the depth buffer in a lossless compressed format to improve the memory bandwidth and the overall query performance. The performance of the copy operation is mainly dependent on the amount of the data copied into the depth buffer and the GPU memory clock rates. Fig. 2 highlights a linear increase in the copy time with the number of data records. The graph also highlights the improvement in the memory performance during the last one year on two high end GPUs - NVIDIA GeForce FX 5900 Ultra (NV35) and NVIDIA GeForce FX 6800 Ultra (NV40). We observe nearly a five times improvement in the performance of the copy operation between two successive generations of GPUs, and this performance improvement trend is expected to continue for the next several years.

3. DATABASE QUERIES

In this section, we describe some of the database queries that are evaluated efficiently on graphics processors.

3.1 Predicates

We evaluate simple predicates in the form of a_i **op** $constant$ using the depth test functionality of the graphics processors. The attribute values are first copied into the depth buffer, and the depth test is used to specify the comparison operation. The comparison operation is performed by rendering a screen filling quadrilateral with depth d. The results of the comparison are then stored in the stencil buffer. Similarly, we evaluate predicates in the form of a_i **op** a_j as $(a_i - a_j$ **op** $0)$, where the $constant$ is 0.

3.2 Boolean Combinations

```
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 3.1: 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.

Complex boolean combinations are often formed by combining simple predicates with the logical operators *AND, OR, NOT*. We use the stencil operation STENCILOP to initialize the appropriate stencil operation for storing the result in stencil buffer. Our algorithm evaluates a boolean expression represented as a CNF expression C_k, and is represented as $C_k = A_1 \wedge A_2 \wedge ... \wedge A_k$ where each $A_i = B_1^i \vee B_2^i \vee ... \vee B_{m_i}^i$. 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 3.1 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 loop in line 2, lines $3 - 19$ evaluate C_i. The valid stencil value is 1 or 2 based 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 also handle special CNF queries such as range queries efficiently on GPUs. A range query is a common database query expressed using a CNF of two simple predicates. If $[low, high]$ is

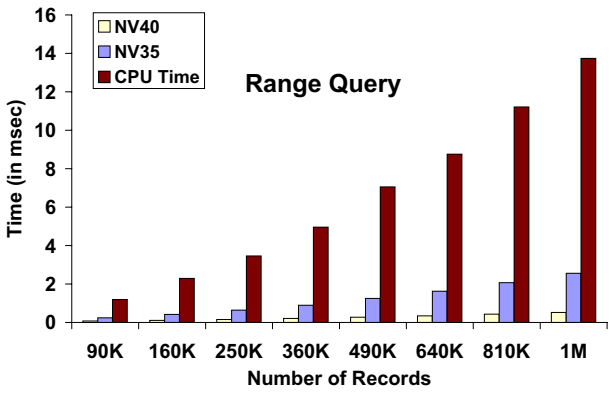

Figure 3: ***Execution time of a range query with*** 60% ***selectivity using a GPU-based and a CPU-based algorithm. Timings for the GPU-based algorithm include time to copy data values into the depth buffer. The NV40 GPU implementation is nearly*** 20 ***times faster than a compiler-optimized SIMD implementation on a 2.8 GHz Intel Xeon processor, and shows nearly 5 times performance improvement over a high end GPU (NV35) implementation in the last year.***

the range for which an attribute x is queried, we can evaluate the expression $(x \geq \text{low})\ AND\ (x \leq \text{high})$ using EVALCNF. Recent GPUs provide a feature *GL_EXT_Depth_bounds_test* [7], which is useful for accelerating shadow algorithms. Our algorithm uses this feature for evaluating a range query efficiently. Although a range query requires the evaluation of two simple predicates, the computational time for our GPU-based algorithm is comparable to the time required in evaluating a single predicate. Fig. 3 highlights the performance of GPU-based range queries and SIMD-optimized queries on a 2.8 GHz CPU. We observe that the NV40 GPU-implementation is nearly 20 times faster than the CPU-based algorithm. Also, due to the high growth rate of GPUs, we observe that our algorithm exhibits a performance improvement of nearly a factor of five between two successive generations of the GPUs, NV35 and NV40.

3.3 Aggregates

Many database operations aggregate attribute values that satisfy a condition. On GPUs, we can perform these operations using *occlusion queries* to return the count of the records satisfying some condition. We describe algorithms to perform some common aggregation queries such as QUANTILES, SUM and AVG.

3.4 QUANTILES

Quantile computations occur frequently in many applications. We can utilize expected linear time selection algorithms such as QUICKSELECT [13] 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. Our GPU-based algorithm does not require data rearrangement and exhibits SIMD characteristics to efficiently 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.

The pseudocode for our algorithm KTHLARGEST is shown in routine 3.2. KTHLARGEST constructs in x the value of the k-th largest

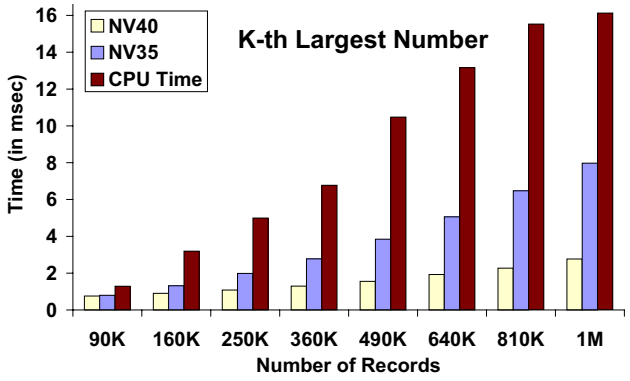

Figure 4: ***Time taken to compute the median using*** KTHLARGEST ***and*** QUICKSELECT ***on varying number of records. Our NV40 implementation is nearly 6 times faster than*** QUICKSELECT ***on a 2.8 GHz Xeon processor.***

```
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 3.2: 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.*

number one bit at a time starting with the most significant bit (MSB), `b_max`-1. Fig. 4 compares the performance of our GPU-based algorithms to compute median on two high end GPUs - NV40 and NV35, and a SIMD optimized CPU implementation of QUICKSELECT on a 2.8 GHz Xeon processor. We observe that the NV40 implementation is nearly 6 times faster than the CPU, and shows nearly 3 times performance improvement per year due to the GPU growth rate.

3.5 SUM and AVG

An accumulator is used to sum a set of data values. Our algorithm uses occlusion queries to efficiently calculate the precise sum of the data values up to double precision and avoids conditionals in the fragment program.

Our algorithm 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

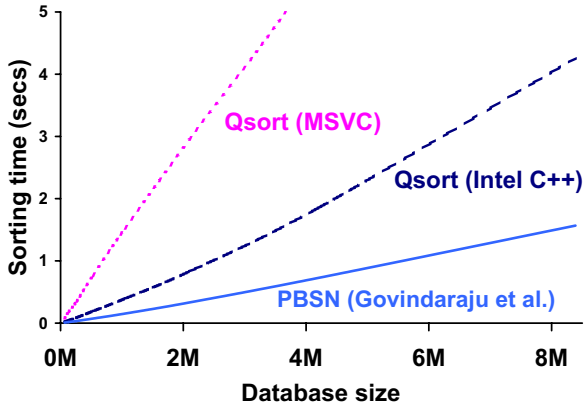

Figure 5: ***CPU vs. GPU performance: This graph compares the performance of optimized implementations of Quicksort algorithm on a 3.4 GHz Pentium IV CPU and the GPU-based PBSN algorithm on a NV40 GPU. We observe that the PBSN algorithm is nearly 3 times faster than the optimized Quicksort algorithm.***

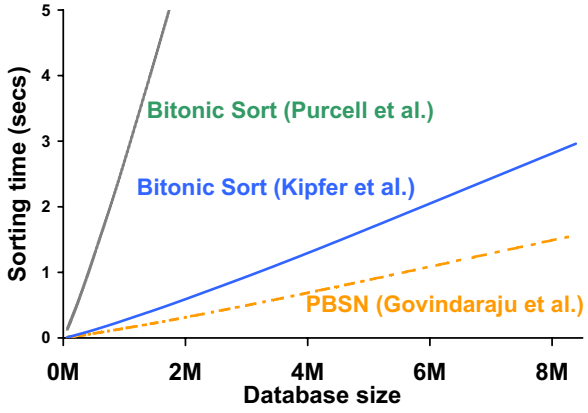

Figure 6: ***Comparison of GPU-based sorting performance: This graph compares the performance of prior GPU-based sorting algorithms against the PBSN algorithm. The graph indicates that the PBSN algorithm is at least 2 times faster than improved bitonic sorting network algorithm proposed by Kipfer et al.***

computation:

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

A simple fragment program is used to determine if the i-th bit of a value is equal to 1 or not, and an occlusion query is used to count the number of values with the i-th bit set to 1.

3.6 Join Operations

Join operations are considered computationally intensive, and can be accelerated by sorting the join keys. Bit-vectors representing the active records are computed using search algorithms such as binary search. Due to the relatively small join-based selectivities, we expect the size of bit-vectors to be small, and therefore, bit-vector computation can reduce the computational overhead.

3.6.1 Sorting

On CPUs, the sorting operation can be performed efficiently using Quicksort. However, the performance of sorting algorithms on CPUs is governed by the cache misses [15], and instruction dependencies [19]. An analytical and experimental study by LaMarca and Ladner [15] indicates that the incoherent data reads and writes in the sorting algorithms can often lead to a performance loss, and for large sequences Quicksort incurs substantially high number of misses.

We use a fast periodic balanced sorting network (PBSN) algorithm proposed by Govindaraju et al. [11] to perform sorting efficiently on GPUs. Govindaraju et al. [11] use the texture mapping and blending hardware to efficiently map sorting network algorithms on GPUs. The PBSN algorithm proceeds in multiple steps. In each step, a comparator mapping is created at each pixel on the screen and the color of the pixel is compared against exactly one other pixel on the screen. The minimum is stored in one of the pixels and the maximum is stored in the other. The comparison operations are performed as vector operations using the blending functionality of the GPU and the mapping is implemented using the texture mapping hardware. For more details, refer to [11].

The resulting algorithm exploits the high memory bandwidth and inherent parallelism on the GPUs. Fig. 5 highlights the performance of the PBSN algorithm implemented on a GPU with that of optimized CPU algorithms. We observe that our PBSN implementation on a NV40 GPU is at least three times faster than an optimized Quicksort implementation on a 3.4 GHz Pentium IV CPU. We used the optimized Quicksort implementation available in the Intel compiler for Pentium processors [1].

We have also evaluated the performance of the PBSN algorithm against other GPU-based sorting algorithms including the improved bitonic sorting network algorithm [14]. Fig. 6 indicates that our implementation of PBSN algorithm using the blending hardware is $2 - 20$ times faster than prior GPU-based sorting algorithms.

3.6.2 Bit-Vector Computation

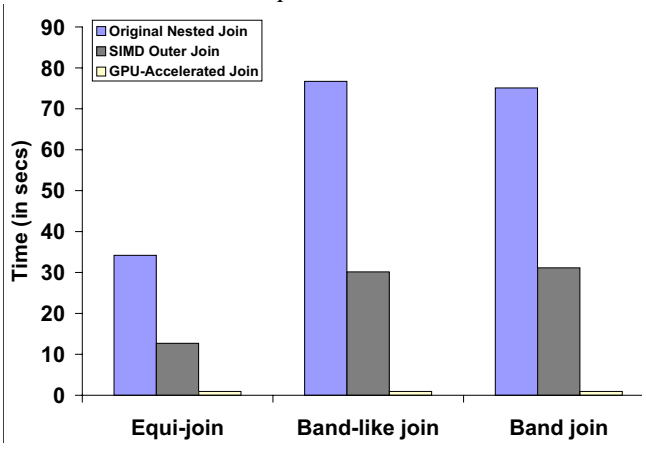

Figure 7: ***Graph highlighting the performance of GPU-Accelerated join algorithm on an NV40 vs. SIMD optimized nested join algorithms on a 3.4 GHz Pentium IV CPU***

We performed experiments on equi-join and non-equi join queries,

[1] http://www.intel.com/cd/ids/developer/asmo-na/eng/20372.htm?prn=Y

and compared the performance against SIMD optimized nested join algorithms [10]. In particular, we evaluated the performance on queries proposed by Zhou and Ross [19]. We have used a selectivity of 10^{-4} for equi-join queries and 5×10^{-3} for band-join queries, and used an outer relation consisting of a million values and an inner relation composed of 10^4 values. Our results (shown in Fig. 7) indicate a significant improvement over the SIMD optimized nested join algorithms.

4. CONCLUSIONS AND FUTURE WORK

We present efficient data representation and computational models to perform fast relational database queries on GPUs. Our algorithms utilize the high memory bandwidth and inherent parallelism within a GPU to quickly evaluate the queries. We compare the performance of our algorithms with optimized CPU-based implementations. In many cases, we observe a factor of $5 - 20$ times performance improvement in the query time by using the GPU as co-processor. Based on current technology trends, we expect the future GPUs to support a wider variety of hardware instructions ond offer higher precision, more parallelism and higher memory bandwidth. As a result, we can design efficient GPU-based algorithms for many other database and data mining algorithms such as k-NN queries, spatial database computations, etc. Finally, we would like to redesign the architecture of current database and data mining systems to exploit the capabilities of the GPUs for faster queries.

Acknowledgements

Our work was supported in part by ARO Contracts DAAD19-02-1-0390 and W911NF-04-1-0088, NSF awards 0400134, 982167 and 0118743, ONR Contracts N00014-01-1-0067 and N00014-01-1-0496, DARPA Contract N61339-04-C-0043 and Intel. We would like to thank NVIDIA corporation for their hardware and driver support. The TCP/IP databases used in our experimental evaluations were provided by Jasleen Kaur and Don Smith. We would like to thank our collaborators: Michael Henson, Ming Lin, Brandon Lloyd, Nikunj Raghuvanshi and Wei Wang for support and useful suggestions.

5. REFERENCES

[1] Anastassia Ailamaki, David J. DeWitt, and Mark D. Hill. Data page layouts for relational databases on deep memory hierarchies. *The VLDB Journal*, 11(3):198–215, 2002.

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

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

[4] Peter A. Boncz, Stefan Manegold, and Martin L. Kersten. Database architecture optimized for the new bottleneck: Memory access. In *Proceedings of the Twenty-fifth International Conference on Very Large Databases*, pages 54–65, 1999.

[5] Thomas Brinkhoff, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. Multi-step processing of spatial joins. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 23(2):197–208, June 1994.

[6] George P. Copeland and Setrag N. Khoshafian. A decomposition storage model. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 14(4):268–279, May 1985.

[7] Ext_depth_bounds_test specification, 2003. http://www.nvidia.com/dev_content/nvopenglspecs/GL_EXT_depth_bounds_test.txt.

[8] Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. Compressing relations and indexes. In *ICDE '98: Proceedings of the Fourteenth International Conference on Data Engineering*, pages 370–379, Washington, DC, USA, 1998. IEEE Computer Society.

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

[10] N. Govindaraju, N. Raghuvanshi, M. Henson, and D. Manocha. A cache-efficient sorting algorithm for database and data mining computations using graphics processors. Technical report, University of North Carolina-Chapel Hill, 2005.

[11] N. Govindaraju, N. Raghuvanshi, and D. Manocha. Fast and approximate stream mining of quantiles and frequencies using graphics processors. *Proc. of ACM SIGMOD*, 2005. to appear.

[12] Gpubench. http://graphics.stanford.edu/projects/gpubench/.

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

[14] P. Kipfer, M. Segal, and R. Westermann. Uberflow: A gpu-based particle engine. *SIGGRAPH/Eurographics Workshop on Graphics Hardware*, 2004.

[15] A. LaMarca and R. Ladner. The influence of caches on the performance of sorting. *Proc. of SODA*, pages 370–379, 1997.

[16] A. Lastra, M. Lin, and D. Manocha. Gpgp: General purpose computation using graphics processors. *http://www.cs.unc.edu/Events/Conferences/GP2*, 2004.

[17] Jignesh M. Patel and David J. DeWitt. Partition based spatial-merge join. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 25(2):259–270, June 1996.

[18] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. McGraw-Hill Higher Education, 2000.

[19] Jingren Zhou and Kenneth A. Ross. Implementing database operations using SIMD instructions. In Michael Franklin, Bongki Moon, and Anastassia Ailamaki, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, June 3–6, 2002, Madison, WI, USA*, pages 145–156, New York, NY 10036, USA, 2002. ACM Press. ACM order number 475020.