Accelerating Braided B+ Tree Searches on a GPU with CUDA

Jordan Fix, Andrew Wilkes, Kevin Skadron

University of Virginia Department of Computer Science
Charlottesville, VA 22904
{jsf7x, ajw3m, skadron}@virginia.edu

Abstract. Previous work has shown that using the GPU as a brute force method for SELECT statements on a SQLite database table yields significant speedups. However, this requires that the entire table be selected and transformed from the B-Tree to row-column format. This paper investigates possible speedups by traversing B+ Trees in parallel on the GPU, avoiding the overhead of selecting the entire table to transform it into row-column format and leveraging the logarithmic nature of tree searches. We experiment with different input sizes, different orders of the B+ Tree, and batch multiple queries together to find optimal speedups for SELECT statements with single search parameters as well as range searches. We additionally make a comparison to a simple GPU brute force algorithm on a row-column version of the B+ Tree.

Keywords: GPGPU, CUDA, B+ Tree, Braided Parallelism, Database

1 Introduction

In the past few years, there has been a growing trend in leveraging the high parallel throughput of graphics processors (GPUs) for general purpose computing (GPGPU). Querying for many results in a large database is a computationally-intensive task, and previous research has shown that running queries on the GPU through CUDA can lead to significant speedups over serial executions [1, 2]. However, we see that these approaches require significant revisions from the traditional database and its underlying data structure. To store records in an organized way, many conventional databases use a B+ Tree, an n-ary search tree with all records stored on the leaves of the tree. Although tree operations are inherently serial, the act of checking the target against the keys of a node can be parallelized. Our aim is preserve the B+ Tree while still exploiting the parallel architecture of the GPU for higher performance.

CUDA is a programming architecture developed by NVIDIA that allows for programmers to easily write parallel programs that are run on the GPU. CUDA uses a hierarchy of thread groups, shared memories, and barrier synchronization that are visible through an API in C [8]. Each thread can be identified by up to 3 dimensions. Threads grouped together form thread blocks that are run on one of the GPU's SIMD multiprocessors, called SMs. Thread blocks share their own memory separate from global memory, and this division of memory and thread blocks is the basis for our approach of running multiple separate queries concurrently.

Braided parallelism is a common technique used in GPGPU computing wherein the programmer executes multiple, independent tasks on the GPU in parallel. This paradigm fits nicely with our B+ Tree traversal algorithm and CUDA's structure

of separate thread blocks [7]: each thread block, running on a single SM, handles a single query, while different queries are processed in parallel on different SMs using different thread block, boosting overall throughput. This requires only a single data transfer for multiple queries, while fully leveraging the parallel nature of the GPU. Our results show that "batched queries" give speedups of up to 16x—including the transfer time of the result set—over sequential execution on a CPU

2 Related Work

Bakkum and Skadron [1] accelerated SQLite queries on the GPU using a brute force approach, with most speedups in the 35X range compared to sequential CPU execution. There are a few fundamental differences between this work and ours, primarily that we parallelize the B+ Tree structure on the GPU instead of transforming the database into a row-column format as Bakkum does. Secondly, we do not choose an industrially-accepted language such as SQLite, because we are interested in answering the more basic question of whether we can see speedups from putting a classically sequential data structure—a B+ Tree--on the GPU, which is highly parallel but exhibits poor single-thread performance. For a fair comparison. we implemented a brute force algorithm on a row-column version of the database, similar to Bakkum's approach.

Similar to Bakkum's work, the research in *Relational Query Co-processing on Graphics Processors* yield a parallel database that also utilizes the row-column format that is called GDB [4]. Again, they use row-column format.

A similar parallel tree data structure is the Fat BTree presented in [3]. They create a shared-nothing environment with minimal communication between processing elements. However, this structure is more suitable for parallel CPU environments because of its memory layout; therefore, we did not try to implement Fat BTree on the GPU. Furthermore, the Fat BTree work only compared its throughput time against other parallel structures and not against sequential execution, so we do not know how effective it is compared to a sequential baseline.

3 Implementation

Our initial codebase is a simple B+ Tree implementation by Aviram [6]. Our reasoning for using a simpler B+ Tree instead of one used in industry is that we are not concerned with details such as locking and memory management; we are purely interested in the theoretical performance gains to be achieved from using the GPU for tree searches. By using a standalone implementation, we can isolate the inherent performance of the data structure and avoid artifacts from extra features that might be present in a database implementation such as SQLite. Starting with Aviram's simple tree, we expanded its functionality to include a range search and integrated CUDA kernel code for parallel execution of B+ Tree searches. In order to create an appropriate data structure that could be effectively used on the GPU, we wrote a transfer function that takes the original data structure and creates an equivalent tree in a single contiguous block of memory. This is particularly useful for memory transfers between the host and device (and vice versa) and allows us to avoid reassigning the pointers to the memory space of the device. The major downside from the transform

function is that it takes a noticeable amount of time to execute. The transform function only needs to be run once during the startup for each tree, so this cost is amortized over the time that the tree is held in memory and used. Furthermore, a from-scratch implementation of a B+ Tree suitable for CUDA can altogether avoid the need for a transform function. However, due to the time constraint of the project, we were unable to achieve this.

A B+ Tree appears much like a binary search tree, although each node can have up to n-1 keys instead of just two, where n defined as the order of the B+ Tree. Note that it is not required for each key to have n-1 keys; this is just a maximum. Each node has n pointers that point to up to n children. Later, we see that choosing the order size has a large impact on our success with parallel implementations.

Tree search is an inherently sequential, logarithmic-running time task; the next child to visit is not known until the query value has been checked against the key(s) in the current node. In order to leverage the GPU's multitude of threads, we used a large node order. We chose an order empirically, testing performance with orders that ranged from 32 to 1024. We ended up finding that the optimal order for both CPU and GPU was 128, which appears to be the "sweet spot" in trading off between number of keys and the logarithmic structure of the tree for our specific implementation.

The time for the CPU to search for a specific value in the B+ Tree depends on where in the B+ Tree it is; if the search value is located on a leaf node that is further to the left side of the tree, then it needs to compare fewer values during traversal than a search for a larger value on the far right side of the tree. The specific query values that we searched for were randomized. Because most of our results used a large number of random queries bundled together, the mean search value for equally-size bundles will be approximately equal with similar standard deviation. Additionally, this random searching is more realistic in practice. We kept the same random search set when comparing each variable for the GPU versus the CPU.

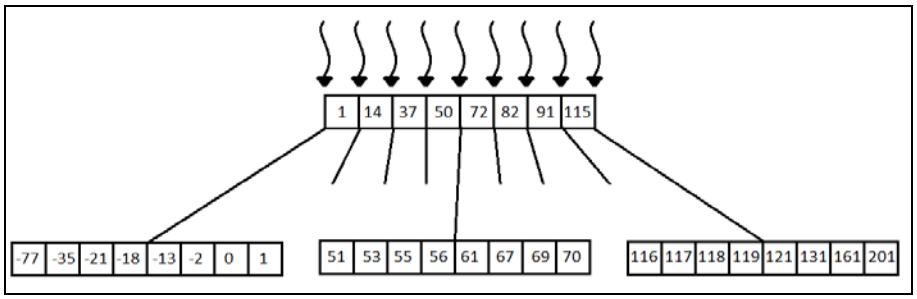

Fig. 1: Example of a B+ Tree with threads checking each key's values against the target value in parallel. In this example, we have 9 threads checking a node of a tree with order 9.

We break our kernel into two iterative phases: picking the key in parallel and then traversing to the next node sequentially. As shown in Figure 1, the first step works by assigning each thread to a possible next child node. Each of these threads compares the search key to values on the current node to determine if it is assigned to the next node to search; if so, it updates the current node. We then *syncthreads* before running the next round, and continue this until we are at a leaf node. This general

setup is seen across each of the kernels we have written. Upon comparing the results of running a single query on the CPU versus the GPU version (on a single thread block), we found that the CPU execution is generally 10-20x faster depending on the target value and tree size; the order was set to the empirically found optimum of 128 for both the GPU and CPU.

Extending our result to run a single query across multiple thread blocks, we found that the synchronization across thread blocks through the use of a barrier and memory fence caused the results to be even slower. Thus, we utilized braided parallelism, where we run independent queries in each thread block concurrently. This allows within-block synchronization but avoids the need for synchronization across the whole GPU. We have parameterized the number of queries to run at once, with each query targeting a random value that we know exists in the tree. This allows us to bundle up to 65,535 queries ($2^{16} - 1$) into a single kernel call, and this has resulted in our biggest speedups when comparing to a sequential execution.

Our next initiative was implementing a range function. If the range is from value a to b, we use our original design to find the two leaves on which value a and b exist and then copy these values back from the GPU. We find a and b in the same way that we search for individual values, avoiding the need to search across all of the leaves. The results are garnered from a simple contiguous memory copy between the starting and ending offsets.

As a fair comparison to the results of Sphyraena and our tree implementation, we wrote a standalone algorithm that simply checks all of the records (in row-column format) for the target key. In keeping with the simplicity of our data structures, the row-column format that we implement is simply a long array that contains one record per row. Because the records that we have chosen to model each have a single key in them, our column size is 1, and we can fit this into a 1D array. Thus, the number of records in the database will directly determine the size of the array. We implemented this as a braided algorithm, where each thread takes a proportional number of records to check in the same manner that is done in Sphyraena. Unlike the aforementioned tree data structure, this technique requires no synchronization across threads. However, this technique does not leverage the logarithmic nature of tree traversal, and so we predict that this method worsens as the amount of data in the DB increases.

We did not include results for a parallel CPU implementation, but if we were to parallelize it, then we would expect an n-core CPU to run at most n times better than its serial counterpart. Break-even points would increase, but we would still see speedups.

Additionally we did not explicitly vectorize for the CPU, because manual vectorization entails considerable effort, and this study would no longer be a fair comparison based on similar programming effort. We did turn off SSE and found that it had negligible effect on execution time.

4 Results

Results were gathered from a machine running Linux 2.6.32 with an Intel® Core™2 Quad CPU Q9400 running at 2.66GHz and 4 GB of memory. It was equipped with an NVIDIA GeForce GTX 480 with CUDA version 3.2, containing 1.5

GB of memory and 15 multiprocessors. The CPU results were compiled with GCC version 4.4.3 with optimization level O2.

The parameters that we chose to examine for a single search value were the number of rows and the size of the query bundle. For the range, we examined the effects of changing the size of the search range. The number of rows ranged from 2^{13} to a maximum of 2^{23} rows. We found there to be no difference in performance whether these rows were ints, floats, etc. The number of queries bundled together was varied from 1 up to $2^{16} - 1$, increasing by a factor of 4. This upper limit is due to CUDA restrictions on the dimensions of a grid of thread blocks. As the size of the bundle increases, more queries may be run in parallel, whereas the CPU must run these sequentially. Although the increase in bundle size meant that there was an increase in the memory transfer time for the return set, this time was minimal compared to the extra time it took for the CPU to run more queries. Therefore, a larger bundle size led to better GPU performance compared to the CPU. In experiments where we hold bundle size constant, we chose 10,000 bundles as a "conservative" number in order to not completely skew performance results when the focus is on other variables. Changing the bundle size has no direct effect on the performance of other variables.

We kept track of timing not only of the kernel call but also how much time it took to transfer both the initial B+ Tree to the GPU and the transfer time of the results back to main memory. The transfer time of the B+ Tree structure is important in this research project, but we expect future work could make use of pinned memory to access the B+ Tree from main memory [5].

Before analyzing the effects of bundling queries, it is important to compare single queries on the GPU vs. the CPU. In Table 1 below, we see that the CPU outperforms the GPU by a factor of about18 when we include transfer times of the result set. Thus, we conclude that without any query bundling, the CPU will consistently outperform the GPU.

Table 1. Analysis of Single Query Performance

Single Query Performance (100k rows, 128 order)						
GPU time, μs			CPU time, μs	CPU Speedup		
Without Transfer	With Results	All Transfers		Without Transfer	With Results	All Transfers
20	36	1370	2	**10**	**18**	**685**

Our next comparison is between the B+ Tree versus the brute force row-column organization utilized in Bakkum's work. Below in Figure 2, we compare speedups of our tree implementation over the brute force algorithm as we sweep the number of rows from 8,000 to 8,000,000:

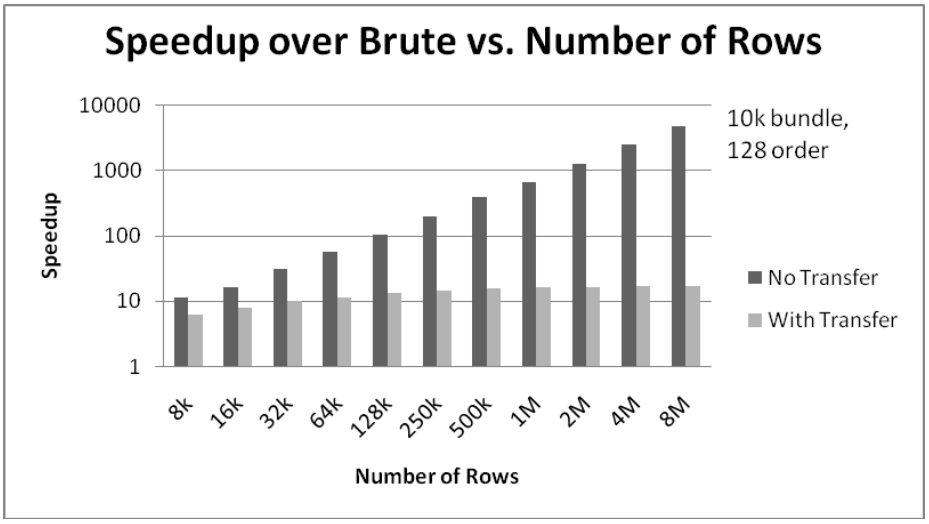

Fig. 2: Speedup over brute force vs. number of rows with 10k queries bundled and an order of 128. Results are shown for no memory transfer times and all transfer times included in the speedup.

We can immediately see the performance gains from only having to visit a subset of the total number of nodes in our B+ Tree versus brute force. In all row sizes shown here, we see speedups grow from around 10 into the tens of thousands. However, when taking transfer times into account, our speedup remains between 10 and 20. This is predictable, as the speedup gains from increasing the number of rows is compromised by having to initially transfer over an increasingly larger B+ tree. The B+ tree grows faster than the array used in brute force as the number of rows increases; in the case of 8M rows, the time for transfer of the B+ tree is 99.7% of the total time for the tree traversal algorithm, while the transfer time for brute force is only around 1.2% of the total time. Meanwhile, for 8K rows, the tree transfer time accounts for 48% of the total time of the tree traversal algorithm, while the transfer time for brute is only 2.6% of the total time. Still, we conclude that using a B+ Tree on the GPU is advantageous over brute force, as the speedup is always growing as the number of rows grows.

Table 2: 4 Tree Scenarios and the Best Architecture for Each

	Infrequent Updates	**Frequent Updates**
Small Size (<250k)	GPU	GPU
Large Size (>250k)	GPU	CPU

Now that we are confident in the B+ Tree implementation over brute force on the GPU, how does it compare to the CPU B+ Tree performance? We have broken this question into four separate database scenarios, seen in Table 2.

In trees that require infrequent updates, we only need to worry about the time it takes to transfer results back to host, as we assume the B+ Tree residing on the GPU memory does not need to be updated often; the cost of the uncommon update is amortized over the multitude of queries run. However, in trees exhibiting frequent

updates, we must focus on the transfer of the B+ Tree onto device memory in addition to results transfer back.

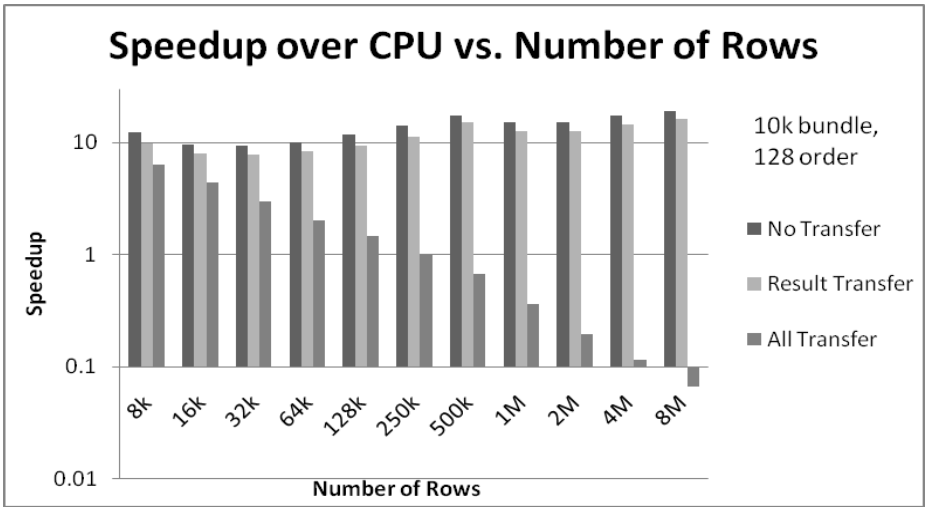

Fig. 3: Speedup over CPU vs. the number of rows with 10k queries bundled and an order of 128. Results are shown for no memory transfer times, result transfer times, and all transfer times included in the speedup.

Figure 3 shows the speedups of the GPU over the CPU implementation for varying numbers of rows. The GPU is superior in three of the four cases. We see that if we ignore transfers altogether, the GPU has a speedup of 9.4 to 19.2 over the CPU, with the higher speedups coming from larger datasets. The all-transfer bar represents the speedups for databases requiring frequent updates, and the result-transfer bar represents the speedups for databases that are infrequently updated. As we can see, the result-transfer speedup is always well-above 1, indicating that the GPU is faster for all sizes of infrequently updated databases. At 250k, we see a break-even point in the all-transfer bars (intersecting with 1 on the y-axis), indicating that the CPU begins to outperform the GPU on frequently-updated databases containing more than 250k records. We expect this break-even point to change depending on different applications and data sets used.

Figure 4 shows the speedup the B+ Tree achieves over the CPU with varying bundle size and 100,000 rows. We can see that as the bundle size increases, the speedup increases as well. With transfer of both the B+ Tree and result set, the GPU does not beat the CPU until a bundle size of around 5550; the peak speedup of 4.5x occurs at 64K queries bundled. However, assuming the B+ Tree is already on the GPU, the break-even point occurs at only 60 queries bundled, with a maximum speedup of 8.1x with 64K queries bundled.

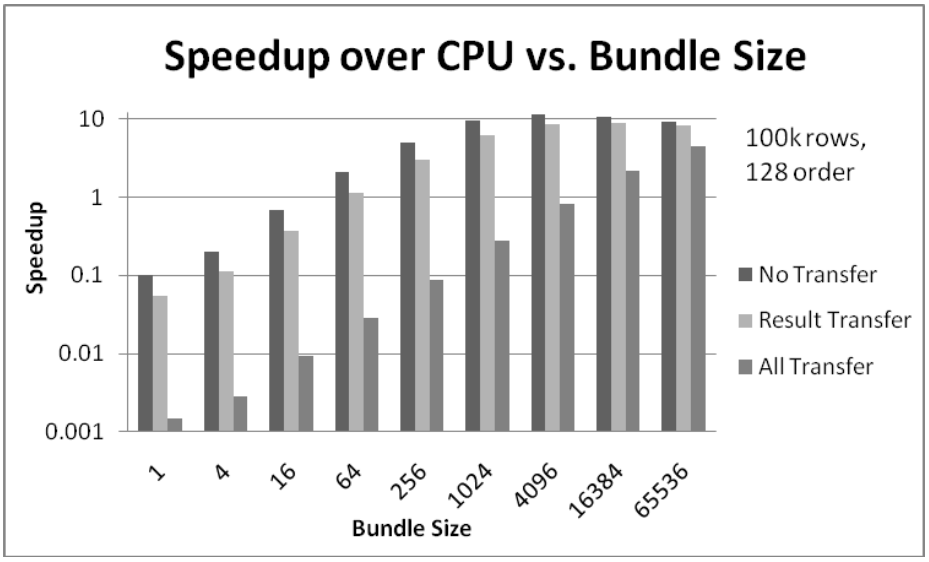

Fig. 4: Speedup over CPU vs. bundle size with 100k rows and an order of 128. Results are shown for no memory transfer times, result transfer times, and all transfer times.

Fig. 4 also shows that the time to transfer back more records as the bundle size gets bigger is minimal compared to the speedup achieved from each successive run. Looking at the GPU execution times including just result transfer, 44% of the runtime of a bundle size of 1 is the transfer time, while only 12% of the runtime of a bundle size of 65535 is the transfer time.

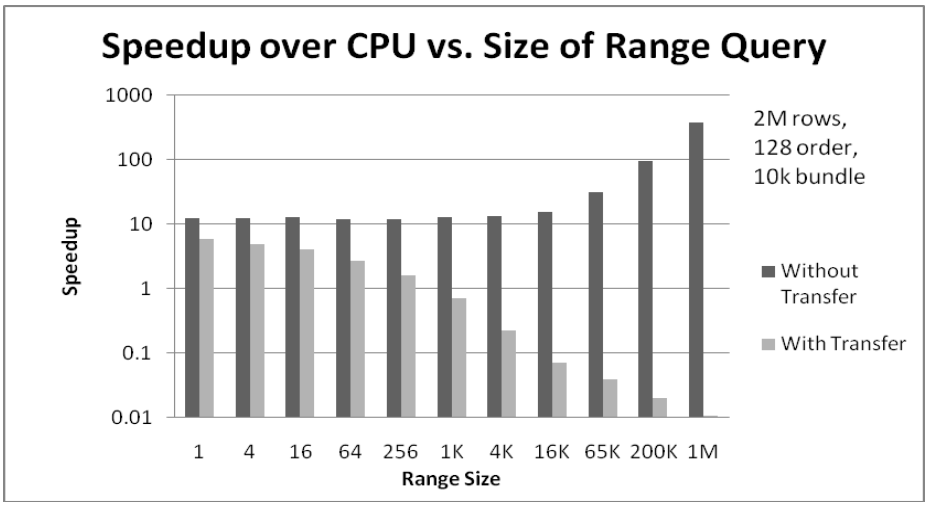

Fig. 5: Speedup over CPU vs. the size of range query with 2M rows, 10k queries bundled, and an order of 128. Results are shown for both with and without transfer times included in the speedup.

Figure 5 shows the GPU speedup over the CPU as the size of the range query changes. Here we see that without transfer, the GPU only outperforms the CPU up until a range of about 650. As the range size increases, the size of memory that must be copied back from the GPU increases as well, as reflected by the numbers. Even though the GPU is outperforming the CPU as the range size increases, this increase in memory transfer times outweighs any speedup seen by the actual selection algorithm.

5 Future Work

Currently, our functionality only allows for searching for a single value or range in the B+ Tree. Future work on this matter could explore implementing more complex queries over more complex records. We believe that this would further increase the speedup of the GPU versus the CPU, because this would likely be implemented by searching first on the B+ Tree index and then by brute force for the remaining conditions. This was the basis for Bakkum's work [1], which saw large speedups, where they argued that the logarithmic time saved by using a B+ Tree would be amortized given enough search conditions.

We also limited our keys and records to only statically sized data types, i.e., no strings. We felt it was unnecessary to implement this for an initial study on the potential value of GPU B+ Tree acceleration.

One of the main advantages of keeping the B+ Tree structure for traversal on the GPU instead of transforming it into a row-column format is that we open the possibility up to having both the CPU and GPU operate on the same B+ Tree. By pinning the B+ Tree in memory, both devices can directly access it without having to deal with transferring a modified tree between them. If GPU functionality were integrated into an RDBMS such as SQLite, this would allow SQLite or the programmer to decide on which device to run individual queries based on current device usage and which device could execute the query faster.

We stopped short of attempting to pin the B+ Tree in memory because the B+ Tree implementation we were using had node structs containing pointers to arrays that were allocated on the fly in the heap, making transfer to the GPU complicated. For this initial study, we decided that it would be more time efficient for our research goals to simply transform that B+ Tree node struct on the CPU to a B+ Tree "knode" struct with statically sized arrays located in each struct that are valid for use in CUDA, instead of rewriting the entire B+ Tree implementation. In order to integrate GPU functionality into SQLite, there will most likely need to be substantial changes made to both SQLite and our implementation.

6 Conclusions

The results of this project emphasize the power of parallelizing a task even when that task requires some synchronization. We see that we pay a price for keeping the threads synchronized; however, the logarithmic number of records that must be visited (which we can do in parallel) overcomes this cost, outperforming serial executions with suitable order, dataset, and bundle size per query. This logarithmic advantage is evidenced by the fact that the GPU B+ Tree implementation beats the

GPU brute force approach on almost all query sets. The results are encouraging in that the larger the dataset size, the larger the speedup.

Furthermore, we see a tremendous advantage in braided parallelism when compared to the CPU: although individual queries are very slow on the GPU, braiding many queries into a single, concurrent run overcomes the inefficiency of smaller queries. With a large bundle, we saw speedups of up to 16x over the serial version.

We see theoretical speedups when we compare the pure runtime of GPU range queries versus the runtime of CPU range queries, but we see that any memory transfers immediately negate the speedups seen on the GPU. This is certainly a venue for future work.

Based on the results, we argue that using our GPU implementation of a B+ Tree is advantageous over serial executions when faced with a database with infrequent updates or a small database that does have frequent updates. This, of course, is assuming that the ability to braid many tree searches together at once is a possibility. Assuming that the database is one that is actively used, however, we expect that there will be no problem constructing large enough bundles to see significant speedups over a serial implementation

Acknowledgements

This work was supported in part by NSF grants CSR-0916908 and CCF-0903471, and SRC grant 2009-HJ-1972. We would also like to thank the anonymous reviewers for their helpful comments and suggestions

References

1. P. Bakkum and K. Skadron. Accelerating SQL Database Operations on a GPU with CUDA. In *GPGPU '10: Proceedings of the Third Workshop on General-Purpose Computation on Graphics Processing Units*, pages 94-103, New York, NY, USA, 2010. ACM.
2. N. K. Govindaraju, B. Lloyd, W. Wang, M. Lin, and D. Manocha. Fast computation of database operations using graphics processors. In SIGGRAPH '05: ACM SIGGRAPH 2005 Courses, page 206, New York, NY, USA, 2005. ACM.
3. H. Yokota, Y. Kanemasa, J. Miyazaki. Fat-Btree: An Update-Conscious Parallel Directory Structure. In *Proceedings of the 15th International Conference on Data Engineering (ICDE '99).* IEEE Computer Society, Washington, DC, USA.
4. B. He, M. Lu, K. Yang, R. Fang, N. K. Govindaraju, Q. Luo, and P. V. Sander. Relational Query Co-Processing on Graphics Processors. ACM Trans. Database Syst., 34(4):1{39, 2009.}
5. CUDA 2.2 Pinned Memory APIs http://developer.download.nvidia.com/assets/cuda/files/CUDA2.2PinnedMemoryAPIs.pdf
6. A. Aviram. Original B+ Tree source: Date: 26 June 2010. http://www.amittai.com/prose/bplustree.html

7. A. Lefohn,. Programming Larrabee: Going Beyond Data Parallelism. Presentation at *SIGGRAPH 2008*. http://s08.idav.ucdavis.edu/lefohn-programming-larrabee.pdf
8. CUDA 2.0 Programming Guide http://developer.download.nvidia.com/compute/cuda/2_0/docs/NVIDIA_CUDA_Programming_Guide_2.0.pdf