GiST Scan Acceleration using Coprocessors

Felix Beier*
Ilmenau University of Technology
felix.beier@tu-ilmenau.de

Torsten Kilias†
Ilmenau University of Technology
torsten.kilias@tu-ilmenau.de

Kai-Uwe Sattler
Ilmenau University of Technology
kus@tu-ilmenau.de

ABSTRACT

Efficient lookups in huge, possibly multi-dimensional datasets are crucial for the performance of numerous use cases that generate multiple search operations at the same time, like point queries in ray tracing or spatial joins in collision detection of interactive 3D applications. These applications greatly benefit from index structures that quickly filter relevant candidates for further processing. Since different lookup operations are independent from each other, they might be processed in parallel on modern hardware like multi-core CPUs or GPUs. But implementing efficient algorithms for all kinds of indexes on various hardware platforms is a challenging task. In this paper, we present a new approach that extends the existing GiST index framework with an abstraction layer for the hardware where index operations are executed. Furthermore, we provide first performance evaluations for the scan execution on CPUs and an Nvidia Tesla GPU.

1. INTRODUCTION

Efficient search operations in huge, possibly multi-dimensional, datasets are crucial for the performance of numerous use cases in a wide range of applications. For example in computer graphics 2D images have to be created from 3D models, consisting of billions of triangles as most common primitive, to display them on a screen. Ray tracing [17] is a well-known technique for this rendering process. For each pixel the path of light is simulated with rays that are shot into the scene to calculate its color. Each ray requires a search operation (point query) to determine the intersection point with visible objects and light sources. To obtain photo-realistic results, this process can be repeated recursively but leads to an exponential growth in the number of search operations.

Another use case is collision detection [13] which is required, e.g., in physical simulations. The task is to determine if two or more 3D models do collide, i.e., if parts of them intersect or not. This is an example for a (spatial) join operation where, naively, the intersection test has to be performed for each triangle of the first model against all other triangles of the second one, requiring a lookup operation each.

Those, and other examples massively benefit from well-known index structures like R-trees [6] or B-trees [3] to speed up lookups. Furthermore, they generate many independent search operations at the same time which can be processed in parallel. Hence, they might benefit from recent trends in hardware development like multi-core CPUs, GPUs, FPGAs, etc. which provide the possibility to process many computations in parallel. To efficiently utilize these hardware components, a fine grained parallelism is required and several characteristics like memory hierarchies, explicit memory transfers, or different processing models have to be considered.

Therefore, the efficient implementation of numerous index structures is a challenging task, requiring a lot of development resources. Especially in scenarios where the actual structure and properties of the data is unknown in advance, e.g., in scientific contexts [1], it is a problem to decide which index type shall be implemented. It might be necessary to prototype and evaluate several to find the best one for each case. Frameworks like GiST [9] support the rapid development of new custom index structures. In the following, we present a novel approach to extend GiST with an abstraction layer for the hardware where tree operations are actually executed. We focus on scan operations in scenarios where numerous search operations have to be processed simultaneously and provide first performance evaluations for their execution on CPUs and an Nvidia Tesla GPU.

2. BACKGROUND & RELATED WORK

2.1 Index Frameworks

The approach presented in this paper is based on the GiST index framework [9] which eases the development of height-balanced index tree structures. Nodes in a generalized search tree (GiST) consist of an array of (key, ptr)-entries where *key* denotes the search key and *ptr* a pointer to the indexed

*This work is partially funded by the TMBWK ProExzellenz initiative, Graduate School on Image Processing and Image Interpretation.

†Torsten Kilias receives funding from the Federal Ministry of Economics and Technology (BMWi) under grant agreement "01MD11014A, 'MIA-Marktplatz für Informationen und Analysen' (MIA)". Research was conducted while Torsten Kilias was working with University of Technology Ilmenau.

DaMoN'12 May 21 2012, Scottsdale, AZ, USA

data or subtree nodes. To develop a new index structure, the key data structure has to be specified as well as some key-dependent methods. The most important one for the present approach is the *consistent(E,q)* predicate which returns false for an entry E if the given value that is searched with a query q can not be found in E's subtree. If it *might be* found, the predicate returns true. Other complex index-invariant methods like a deletion, insertion, or height-balancing the tree are implemented by the framework. For GiST details, please refer to [9].

The GiST idea to hide the complexity of tree management operations in order to speedup the development of new index structures with similar properties but different key and/or query predicates, was adopted for other scenarios than height-balanced search trees. SP-GiST [2] implements a similar approach for space-partitioning trees like octrees or k-d trees. For executing tree methods efficiently on modern hardware platforms, CC-GiST [11] extends GiST with pointer and key compression techniques to become cache conscious.

2.2 Index Operations on the GPU

Several approaches exist for processing index structures on special hardware like GPUs (which we also refer to as "device" in the following). Indexes are either embedded in other GPU algorithms like joins [8] where the index scan results are directly used without any data transfer back to the CPU (which we also refer to as "host"). Another approach is to use a coprocessor accelerated index as a separate building block which offloads queries to the device and transfers results back to the host. [19] implements such an approach for R-trees. It requires that the entire tree data structure - an array containing child node IDs for each tree node and another one for the coordinates of corresponding minimal bounding rectangles (MBRs) - is stored on the device's main memory. Scans are executed iteratively for each tree layer, starting at the root. The query predicate is evaluated in parallel for all child nodes of the current layer and matching children are marked in a boolean result array. After each iteration these results are logically ANDed with the array of the previous iteration. After processing the leaf layer, the result array contains the final query results, i.e., all matching leaf nodes.

To reduce response time of a single query, [16] implements speculative query processing for prefix trees indexing string data. For one query, nodes of deeper layers than the one currently scanned are processed in parallel on separate cores. The intent is to reduce the number of iterations required for processinging a single query. Results of these deep path scans are merged after each iteration to filter false positive results. A speedup is achieved if the computational overhead for scanning unnecessary nodes can be compensated with saving additional iterations.

To maximize performance of both, single query response time as well as throughput in batch query processing, the Fast Architecture Sensitive Tree (FAST) [10] implements a blocking strategy in an in-memory binary search tree. Nodes of (sub)trees are grouped together and stored contiguously in memory to maximize locality of data access and therefore cache efficiency. Special hardware functions like SIMD capabilities are considered as well. For this special tree structure, scan and bulk load operations were parallelized and implemented for a CPU and GPU system.

2.3 Contributions of this Paper

When analyzing the approaches in sections 2.1 and 2.2, several challenges can be identified. Existing GiST frameworks support the development of new index structures. But they lack the capability to efficiently utilize modern hardware. Furthermore, only single queries are processed which is not optimal for throughput-oriented use cases as mentioned in section 1. Existing index implementations for GPUs require that the entire index data is stored in device memory. Hence, transfer times which can be critical for algorithms using coprocessors [5,12] were ignored in performance evaluations. For large-scale applications this assumption is unrealistic. Some implementations do not efficiently utilize the hardware, e.g., [19] uses only one CUDA block (cf. section 3.5.1) for scanning and therefore utilizes only one of the available cores. Furthermore, it is not computationally efficient. All index nodes must be scanned in order to process one query. The filtering idea of an index tree is therefore ignored.

The intent of our approach is to extend the GiST framework to hide the complexity of implementing and tuning general index tree algorithms for special hardware like multicore CPUs or GPUs. Since at the current state of development only a prototype was implemented, our approach is subject to some restrictions:

- Unlike FAST [10] which implements scan and creation operations, we focus on scans only.
- Generally, the presented algorithms could be specialized for arbitrary parallel coprocessors. Currently we provide a sample implementation for a GPU.

3. EXTENDING GIST WITH A HARDWARE ABSTRACTION LAYER

3.1 Requirements

Several aspects have to be considered when designing the new extension for the existing GiST framework:

Fine-grained parallelism: To maximize the benefit achievable through parallel execution of a large set of index scan operations, the tasks have to be partitioned into independent subtasks that can be efficiently executed on the available processing units.

Exploiting hardware capabilities: Since different types of processing hardware have special characteristics like caches, SIMD capabilities, support of asynchronous operations etc., the operations have to be tailored to each specific platform for maximizing the achievable performance. Especially coprocessors like GPUs require the explicit transfer of the data to the device which introduces an overhead that has to be compensated somehow. Otherwise, a possible speedup in comparison to algorithms carefully tuned for CPUs as processors can easily be nullified [5, 12] - or even worse a slowdown is experienced.

Out-of-core implementation: To be scalable in terms of size of indexed data, the new index framework must not rely on the entire data to be stored in the available memory of the respective execution hardware. Usually the available main memory of coprocessors is much smaller than the available host main memory. Existing GPU approaches (cf. section 2.2) assume that all data is stored or generated in the

GPU's main memory and therefore transfer times can be neglected. Since coprocessors are shared resources for multiple tasks, we consider this assumption as unrealistic.

Work efficiency: For scalability in terms of computations per input data, each additional input element must only add a (nearly) constant overhead. Otherwise, a linearly increasing workload can not be handled with adding a linear amount of (co)processors, resulting in increasing hardware costs to handle it.

3.2 Scan Processing Model

Our GiST extension was developed to maximize index query throughput. Therefore, a synchronous processing model [18] was implemented which is illustrated in figure 1. The application requests the scan of a batch of index queries that shall be processed simultaneously. After scanning the tree, all matching leaf nodes per query are returned. Generally, each query might have multiple result leaf nodes. Therefore, a set of iterators (one per query) is returned. Depending on the application's use case, they can be processed in the same order like in the input query batch - or any order, i.e., the next available result. Depending on the internal scheduling not all queries are processed equally fast, hence the former access operation may block even if results of following queries are already available while the latter isn't.

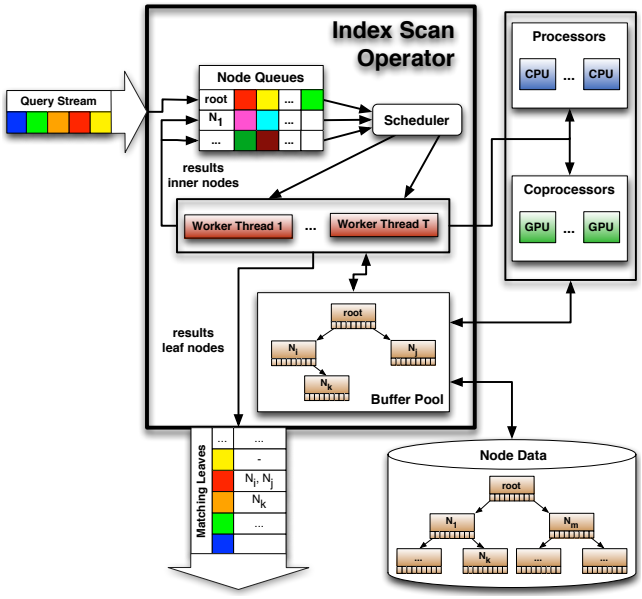

Figure 1: Scan Operator Overview

Query batches are managed in queues per tree node. New batches start at the root. Nodes are scanned iteratively like in [19], i.e., all queries in the batch of a node are processed in a single step. The scan results - matching (query, child node)-pairs - are either added to

a) a queue for the child node in case the scanned node was an inner node and matching children have to be processed in the next iteration

b) the result list of the query if the scanned node was a leaf.

The decision which processing unit of the available hardware executes a node scan is implemented in a scheduler component and can be adapted for each supported hardware platform, depending on its characteristics. Node data is transferred to the (co)processors and, if necessary, can be stored on disks. To avoid data transfers, caching strategies are implemented. The iterations are synchronized by the framework with a single host thread.

3.3 Scan Parallelization

To achieve maximal benefit from massively parallel hardware, a fine-grained parallelization for the scan operation has to be found. The processing scheme is illustrated in figure 2. In our model, a set of queries is executed simultaneously for a number of tree nodes which are independent from each other. For each query, all child node entries (referred to as slots in the following) have to be tested with the GiST key predicate (cf. section 2.1). For the slot keys, two cases exist:

1. **No strict order is defined between the keys:** The data ranges represented by the corresponding subtrees may overlap, i.e., even for point queries multiple child nodes may match. In this case, all child nodes can potentially belong to the result set. Therefore, the key predicate test is required for each of them. An example for this property is the R-Tree where bounding hypercuboids are stored as keys which may intersect within a node.

2. **Keys are strictly ordered:** The represented data ranges do not overlap. Due to this, a binary search can be used to find matching child nodes for a single point query (or multiple binary searches for range queries). The most common example for this is the B-Tree.

In our implementation we focus on the first case since it is the most general one. The second case can be regarded as optimization to speed up the scan within a node.

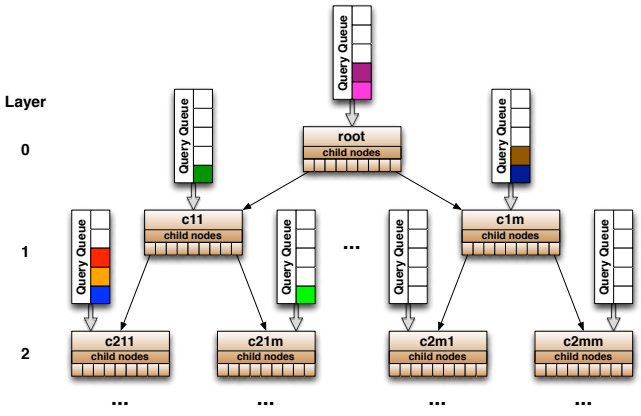

Figure 2: Index Tree Scan

3.3.1 *Intra-Node Parallelization*

Within a node the tests of all (query, slot)-pairs are independent from each other. Therefore, they can be executed in parallel with SIMD operations similar to [10]. We obtain an $(n \times m)$-test matrix for a query batch of size n that shall be scanned on a node with m slots. Let x denote the SIMD capability of the execution hardware (usually $x << n,\ x << m$). For inner nodes one slot is scanned for x queries (matrix column) and for leaf nodes one query

is scanned for x slots (matrix row). Using this scheme allows sequential writes of the respective result lists (cf. section 3.2) which is, due to hardware-dependent memory transfer transactions, usually faster than scattered write operations.

3.3.2 Inter-Node Parallelization

Since all nodes to be processed in an iteration are independent from each other, their scans can be parallelized too. All nodes are scheduled to the available (co)processors and therefore the approach scales linearly with the number of cores. In contrast to existing GPU approaches where a query (batch) has to be finished before the next one can be processed, this independent node scheduling allows the implementation of a pipeline, i.e., within each iteration new queries might become ready to be scanned by the index, which are considered in future iterations. They could be scheduled to the root node by the application or become ready if they were waiting in the previous iteration for a data transfer to complete after a cache miss.

3.4 Parameter Determination

The main index parameters that are relevant for the node scan are the number of slots s per node and the number of queries q within a node batch. To be cache efficient, both, the node and the batch have to fit into the (co)processor cache. In this case, all matrix tests can be executed without cache miss penalties. Therefore, the hardware-dependent cache size as well as the SIMD capability x (cf. section 3.3.1) determine efficient values for s and q. If heterogenous hardware is used - e.g., with a different cache size - the other parameter can be adjusted accordingly and considered during the scheduling.

When indexing very large datasets, it may happen that the index does not completely fit into the available main memory. In this case, like in any disk-based DBMS, nodes can be stored on pages which are fetched from disk if necessary. Usually only a small, cacheable part of all nodes is required for processing the application's lookup operations. For processing multiple query batches one after another, the upper tree layers are required very often. Furthermore, locality can be exploited for multiple search operations. For example when rendering a camera movement within a 3D-scene, the same (or neighboring) spatial areas need to be scanned again in consecutive frames [4].

3.5 GPU Implementation

As use case for a special co-processing hardware we implemented our framework for a GPU besides the straightforward CPU implementation. We decided to use the CUDA programming model [15] to abstract from the specific underlying GPU. Other programming models like OpenCL [14] were also possible, but we chose CUDA since we used an Nvidia GPU for our experiments.

3.5.1 Mapping to CUDA Processing Model

In CUDA, a parallel algorithm (kernel) of independent subtasks (blocks) is processed by a GPU consisting of multiple independent cores on separate dies, called streaming multiprocessors (SMs). Each SM can, depending on memory requirements and hardware capabilities, execute one or more blocks consisting of multiple threads. Within a SM there are w thread processors which work in lockstep mode, i.e., all of them execute the same instruction on different parts of input data - or idle in case code branches differ. For current GPUs, those thread groups (warps) usually have sizes of 16 or 32. The threads of a block can be explicitly synchronized to work on shared data. In our framework, we consider (node, query batch)-pairs to be scanned as separate tasks and therefore map them to CUDA blocks. The elements of the key predicate test matrix are mapped to CUDA threads.

Besides this processing hierarchy, CUDA defines a memory hierarchy which is depicted in figure 3. The global main memory (VRAM) on the GPU device (up to several GB in size) can be directly accessed by the host system. Data that shall be processed has to be transferred explicitly via the connecting PCIe bus. In our environment we achieve transfer rates of ≈5.5 GB/s up and down stream. For access via the SMs, the VRAM achieves transfer rates of ≈75 GB/s. To reduce transfers, we use the VRAM to cache node and query data between different iterations. Furthermore, it is used to store result data which can be preallocated since the maximum size of the matrices is known in advance. On a SM's die, there is a small (some KB in size) shared memory buffer (SHM) which achieves access rates ≈2 orders of magnitude higher than the VRAM. We use it to cache node and query data for a single scan.

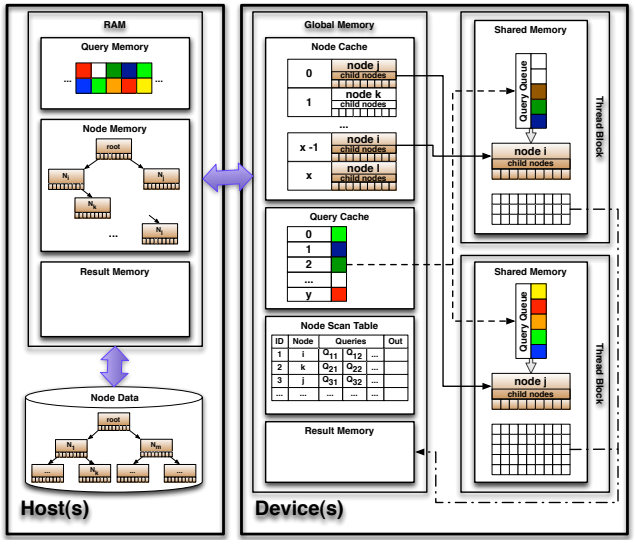

Figure 3: Index Scan - CUDA Implementation

3.5.2 Scan Preparation

To determine the parameters for each block, a preparation kernel is required. It fills a Node Scan Table (cf. figure 3) with the starting positions of each node data and query batch. Therefore, all cached node and query entries are scanned in parallel for each task and a pointer to the starting address is inserted into the table which is later used to load the data into the SHM cache. Additionally, the position of the results in the output data buffer is computed.

3.5.3 Scan Execution

After the scan has been prepared, the actual matrix test kernel is launched. First, all node slots and queries to be scanned are loaded in parallel into the SHM of the active SM. To assure that all required data has been stored, all threads are synchronized after this step. Second, the scan

is executed. For each (query, slot)-pair the index-dependent key predicate is evaluated as explained in section 3.3.1. The predicate has to be provided by the index developer as well as the actual key types. Note, that for storing the key data no special method needs to be provided by the developer since the framework loads raw byte arrays of fixed size for all nodes. Predicate results are directly written into the global result memory buffer since the size of the matrix grows quadratically in the order of the number of slots and queries.

3.5.4 Result Creation

Since - depending on a key predicate's selectivity - the rows or columns of the result matrix are usually sparsely populated, we implemented an optional result creation kernel as final phase to compress them with removing gaps between the (queryID, slotID)-entries. Therefore we store the rows/columns one after another in the SHM (which always fit) and execute a parallel prefix scan/stream compaction primitive [7] on the array which removes all NULL entries.

4. EVALUATION

We used our framework to implement a 3-dimensional R-tree with 64-bit values as coordinates and conducted some experiments on our test server having two Intel Xeon X5690 6-core CPUs and two Nvidia Tesla C2050 GPUs which were connected to the host system via PCIe 2.0 16x bus.

Experiments were executed to analyze the impact of the index parameters on the CPU and GPU scan algorithms and profiled the efficiency of our GPU implementation. Parameters that are relevant for our framework are:

- **slots:** the number of child entries per node which impacts the total node size
- **queries per task:** the number of queries that are scanned per node
- **tasks:** the number of nodes scanned in parallel in each iteration, impacting the utilization of available cores
- **selectivity:** the number of matching child nodes according to the range of a query. All nodes were artificially generated with disjoint rectangles of equal area as child entires. Due to this, we were able to generate queries with certain selectivities to simulate different result sizes.

4.1 Impact of Tree Parameters

The first question at hand is if executing a node scan on the GPU is beneficial, i.e., if it is faster (including the overhead) than the same scan on the CPU. We compared total CPU and GPU scan execution times with the varying tree parameters *slots* and *queries per slot* in steps of 32 (warp size as SIMD unit on the GPU cf. section 3.3.1). To reduce the number of measurements, we used a constant number of $tasks = 128$ which is much larger than the number of cores on the GPU to keep it fully utilized. Queries were generated with $selectivity = 0.25$ to include result processing times. As a measure, we define:

$$speedup = \frac{CPU\ time}{GPU\ time} \quad (1)$$

for the execution on the device vs. one thread on the host. The results are illustrated in figure 4. The GPU algorithm

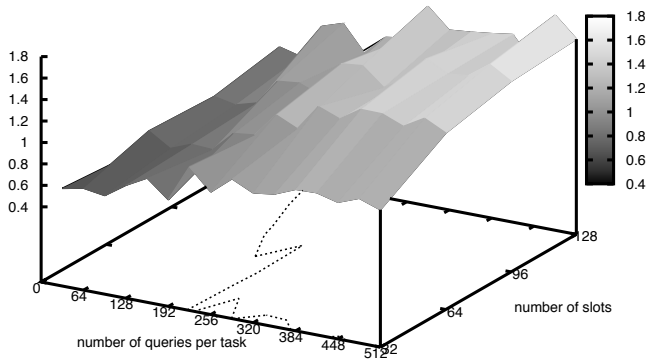

Figure 4: GPU Speedup

is up to 1.8 times faster than its CPU counterpart for scanning large query batches (448-512) on nodes with 96-128 slots. But there are several parameter combinations where the GPU algorithm is up to 2.5 times slower. This slowdown is experienced when scanning short query lists on small nodes since the GPU can not exploit its full potential through highly parallel processing for such small inputs.

So, how do we chose the parameters? Finding an optimal value for the number of slots is quite difficult. In contrast to index implementations where the size of disk pages determines the sizes of index nodes, it now becomes important to know where a node shall be processed - which, besides the installed hardware, depends on the workload. A fixed node size seems not to be useful. For high-loaded nodes like the first tree layers where most queries pass through, a high fanout would be beneficial since the GPU provides best performance here. For low-loaded nodes like those in layers near the leaves, a CPU scan execution on smaller nodes would be faster since query batches are expected to be shorter there.

To expect CPU and GPU runtimes for different parameter values (especially for different query batch sizes, since node sizes won't change at runtime) is important for the framework to perform scheduling decisions. Wrong expectations could lead to a slowdown. Therefore, the model of figure 4 has to be implemented in the scheduler component. The dotted line on the x-y projection marks the break-even parameters where both implementations have approximately the same runtime. All scans on the left side shall be executed on the host while the others are faster with co-processing on the device.

Implementing such a decision model that is general enough for use in a framework is non-trivial. It depends on the actual (co)processors that are used, and the actual index implementations provided by the developer. The keys and predicates for the currently implemented R-tree are relatively simple. A predicate evaluation translates just to 2n coordinate comparison operations where n denotes the dimensionality of the tree. For more complex predicates like nearest neighbor searches a GPU implementation may be more beneficial due to the increased computational complexity. To solve this scheduling problem we suggest executing some calibration scans for estimating the model. We are currently working on a self-learning scheduler that approximates expected runtimes with polynomials and adjusts its parameters according to actually measured execution times. But further details on that are out of the scope of this paper.

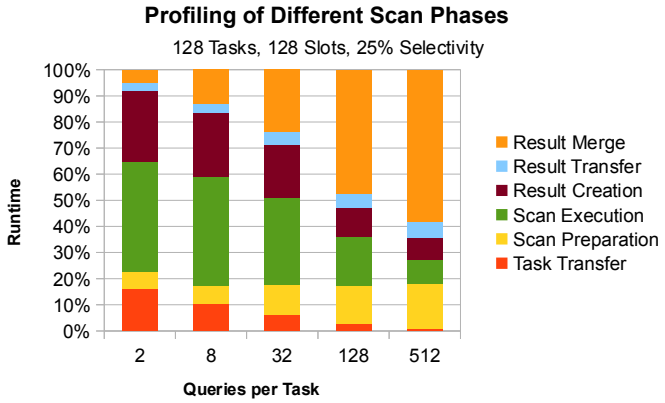

Figure 5: Scan Phases - Percental Runtimes

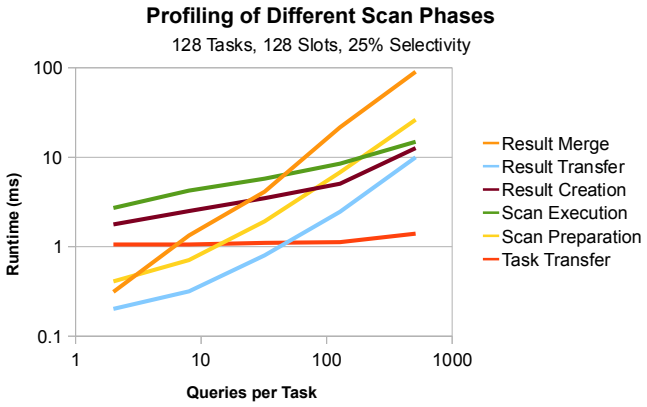

Figure 6: Scan Phases - Runtimes

One may discuss if comparing runtimes of an entire GPU to just a single host thread is valid. In our model, that is exactly the decision the scheduler has to do. Each processing unit that can be separately controlled is managed by a worker thread (cf. figure 1). For host processors, each CPU core (12 in our scenario) can execute an independent thread, or even more when hyperthreading is enabled. GPU cores on the device are not working independently but are executing the same kernel. From the scheduler point of view, it has to be decided which subsets of all nodes in the current iteration have to be distributed to which worker thread to achieve a maximum overall throughput. This can be achieved with the hybrid system approach where many independent CPU cores can scan many low-loaded nodes while attached GPU devices process a smaller number of high-loaded nodes in the meantime. Moreover, for processing the final results after the queries we streamed through the index, additional cores will be required.

4.2 Scan Profiling

Finally, we analyze the efficiency of our GPU framework implementation. We omit the host counterpart here since its algorithm is quite simple and we don't have explicit control on the usage of caches as they are implemented by the hardware. During the first predicate evaluation in a test matrix row / column, the data elements are stored into the executing CPU's L2 cache (which is larger than the GPU's SHM) and are reused for all further evaluations.

To analyze the GPU algorithm, where the entire caching strategy is managed by the framework, we profiled the different scan phases (cf. section 3) with varying queries per task because this parameter impacts the complexity of all scan phases. The number of tasks was fixed at 128 for guaranteeing a full GPU utilization. A selectivity of 0.25 was chosen to include the result processing phases on the host and the device. As shown in figures 5 and 6 the impact of input transfers decreases with increasing queries per tasks since the computational complexity grows quadratically with respect to the input size. For result transfers, as expected, the impact increases since the result size is in the same order as the computation (matrix). What we did not expect is a large overhead (>50%) for merging GPU results with the batches maintained on the host which is the main bottleneck of our prototype. This overhead grows dramatically with the size of the result sets depending on the selectivity (not shown here). Detailed profiling revealed that maintaining the shared data structures used for storing query batches requires expensive synchronizations on the host side. This can be avoided by merging results locally and executing a small synchronizing phase afterwards. Large performance improvements are expected here. Further tuning can be done with utilizing Nvidia hardware characteristics like coalesced reads, avoiding bank conflicts, or overlapping kernel execution with copy operations [15].

5. CONCLUSIONS & OUTLOOK

Recent trends in hardware development such as multi-core CPUs and GPUs provide great opportunities for improving the performance of index operations by fine-grained parallelization. However, particularly co-processing units like GPUs require a signifiant effort for an efficient implementation of the index structures.

In this paper, we have introduced our approach of bringing the GiST framework to these platforms. In contrast to other works, which rely on the assumption of keeping the entire index structure in the GPU memory, our approach implements a hybrid out-of-core strategy hiding the complexity of efficient data transfer between host and GPU memory. Furthermore, we focus on a multi-query processing model to mitigate the transfer time.

The experimental evaluation of an R-tree implementation shows that for large tree nodes and query batch sizes that can be processed in parallel, a scan on the GPU can be nearly twice as fast as the same scan on the host. Those speedups are expected to be larger after further tuning the prototype. Nevertheless, careful scheduling is required because, for small nodes and/or a small number of queries, a GPU-based processing may otherwise lead to a slowdown of 2.5. For future work we therefore plan to:

1. implement a learning scheduler component for automating this task
2. support index load and update operations
3. investigate further index implementations to show the benefit of a GiST-based framework to speed up the development
4. integrate support for other coprocessors and models

6. REFERENCES

[1] A. Ailamaki. Managing scientific data: lessons, challenges, and opportunities. In *Proceedings of the 2011 international conference on Management of data*, SIGMOD '11, pages 1045–1046, New York, NY, USA, 2011. ACM.

[2] W. G. Aref and I. F. Ilyas. Sp-gist: An extensible database index for supporting space partitioning trees. *Journal of Intelligent Information Systems*, 17:215–240, 2001. 10.1023/A:1012809914301.

[3] D. Comer. Ubiquitous b-tree. *ACM Comput. Surv.*, 11(2):121–137, June 1979.

[4] W. T. Correa, J. T. Klosowski, and C. T. Silva. Visibility-based prefetching for interactive out-of-core rendering. In *Proceedings of the 2003 IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, PVG '03, pages 2–, Washington, DC, USA, 2003. IEEE Computer Society.

[5] C. Gregg and K. Hazelwood. Where is the data? why you cannot debate cpu vs. gpu performance without the answer. In *Performance Analysis of Systems and Software (ISPASS), 2011 IEEE International Symposium on*, pages 134 –144, april 2011.

[6] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, SIGMOD '84, pages 47–57, New York, NY, USA, 1984. ACM.

[7] M. Harris, S. Sengupta, and J. D. Owens. Parallel Prefix Sum (Scan) with CUDA. In H. Nguyen, editor, *GPU Gems 3*. Addison Wesley, Aug. 2007.

[8] B. He, K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo, and P. Sander. Relational joins on graphics processors. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 511–524, New York, NY, USA, 2008. ACM.

[9] J. M. Hellerstein, J. F. Naughton, and A. Pfeffer. Generalized search trees for database systems. In *Proceedings of the 21th International Conference on Very Large Data Bases*, VLDB '95, pages 562–573, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[10] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. D. Nguyen, T. Kaldewey, V. W. Lee, S. A. Brandt, and P. Dubey. Fast: fast architecture sensitive tree search on modern cpus and gpus. In *Proceedings of the 2010 international conference on Management of data*, SIGMOD '10, pages 339–350, New York, NY, USA, 2010. ACM.

[11] W.-S. Kim, W.-K. Loh, and W.-S. Han. Cc-gist: Cache conscious-generalized search tree for supporting various fast intelligent applications. In S. Mehrotra, D. Zeng, H. Chen, B. Thuraisingham, and F.-Y. Wang, editors, *Intelligence and Security Informatics*, volume 3975 of *Lecture Notes in Computer Science*, pages 657–658. Springer Berlin / Heidelberg, 2006.

[12] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey. Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu. In *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, pages 451–460, New York, NY, USA, 2010. ACM.

[13] M. Moore and J. Wilhelms. Collision detection and response for computer animation. In *Proceedings of the 15th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '88, pages 289–298, New York, NY, USA, 1988. ACM.

[14] A. Munshi. The OpenCL Specification, 2011.

[15] C. Nvidia. Nvidia cuda. *Changes*, 7(6(36)):187, 2011.

[16] P. B. Volk, D. Habich, and W. Lehner. Gpu-based speculative query processing for database operations. In *Proceedings of the 1st International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures*, 2010.

[17] T. Whitted. An improved illumination model for shaded display. In *ACM SIGGRAPH 2005 Courses*, SIGGRAPH '05, New York, NY, USA, 2005. ACM.

[18] D. Wu, F. Zhang, N. Ao, G. Wang, X. Liu, and J. Liu. Efficient lists intersection by cpu-gpu cooperative computing. In *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pages 1 –8, april 2010.

[19] X. Xiao, T. Shi, P. Vaidya, and J. J. Lee. R-tree: A hardware implementation. In *CDES'08*, pages 3–9, 2008.