Concurrent Data Structures on GPU

Xi He
Georgia State University

The LaTeX `acmtrans` document style formats articles in the style of the ACM transactions. Users who have prepared their document with LaTeX can, with very little effort, produce camera-ready copy for these journals.

Categories and Subject Descriptors: G.4 [**MATHEMATICAL SOFTWARE**]: Parallel and vector implementations; G.1.0 [**NUMERICAL ANALYSIS**]: General—*Parallel algorithms*

General Terms: Algorithm, Performance

Additional Key Words and Phrases: Parallel algorithm, SIMD, Data strucuture

1. INTRODUCTION

In Computer Science, a data structure is a particular way of storing and organizing data in a computer so that it can be used efficiently. Different kinds of data structures are suited to different kinds of applications, and some are highly specialized to specific tasks. Table II lists various categories of data structures.

According to Flynn's taxonomy [Flynn 1972], computer architectures can be classified into four categories: SISD (Single Instruction Single Data), SIMD (Single Instruction Multiple Data), MISD (Multiple Instruction Single Data) and MIMD (Multiple Instruction Multiple Data). MISD is the uncommon architecture, and people usually pay more attention to three other ones. A SISD computer exploits no parallelism in either the instruction or data streams. Traditional uniprocessor personal computers are examples of such architecture. The data structures above is originally designed for this architecture.

In a MIMD computer, multiple autonomous processors simultaneously execute different instructions on different data. Shared memory multiprocessor systems are the typical MIMD computers. Many researches have been carried out on the concurrent data structure in shared memory systems. Data structures like stacks, queues, linked lists, hash tables, search trees, priority queues have been fully studied. The primary difficulty for concurrent data structure is concurrency: because threads are executed concurrently on different processors, and are subject to operating system scheduling decisions, page faults and interrupts, we must think of the computation as completely asynchronous, so that the steps of different threads can be interleaved arbitrarily. A nature and common way to prevent such interleaving

Linear data structures	Arrays	Array, Bit array, Dynamic array, Matrix
	Lists	Doubly linked list, Linked list, Self-organizing list, Skip list
Trees	Binary Trees	Binary search trees, Red-black tree, AVL tree, Splay tree
	B Trees	B tree, B+ tree, B* tree
	Heaps	Binary heap, Binomial heap, Fibonacci heap, Skew heap
	Tries	Tries, Suffix tree, Suffix array
	Multiway trees	Disjoint set, Ternary search tree
	Space-partitioning trees	Kd-tree, R-tree, Quadtree, Octree
	Application-specific trees	Decision tree, Parse tree, Syntax tree
Hashes	Hashes	Hash table, Hash tree
Graphs	Graphs	Adjacency list, Adjacency matrix

Table I. The list of data structures

is to use a mutual exclusion lock. But precaution must be taken on the use of lock because naive locking scheme can severely undermine scalability. The first problem with locking scheme is the sequential bottleneck. Many improvement on concurrent data structures is to reduce the number of locks acquired and lock granularity. The second problem with locking scheme is that it suffers from memory contentions. To address this problem, some lock implementations are designed to avoid such problems for various types of shared memory architectures.

Computers with SIMD architecture can perform the same operation on multiple data simultaneously. The first use of SIMD instructions was in vector supercomputers of the early 1970s such as the CDC Star-100 and the Texas Instruments ASC. Vector processing was especially popularized by Cray in the 1970s and 1980s. Small-scale (64 or 128 bits) SIMD has become popular on general-purpose CPUs in the early 1990s and continuing through 1997 and later with Motion Video Instructions (MVI) for Alpha. SIMD instructions can be found, to one degree or another, on most CPUs, including the IBM's AltiVec and SPE for PowerPC, HP's PA-RISC Multimedia Acceleration eXtensions (MAX), Intel's MMX and iwMMXt, SSE, SSE2, SSE3 and SSSE3, AMD's 3DNow!, ARC's ARC Video subsystem, SPARC's VIS and VIS2, Sun's MAJC, ARM's NEON technology, MIPS' MDMX (MaDMaX) and MIPS-3D. The IBM, Sony, Toshiba co-developed Cell Processor's SPU's instruction set is heavily SIMD based.

In this paper, we discuss some of concurrent data structures in GPU, a computational platform with SIMD-like architecture.

2. STACK

Stack is an important data structure for many application. Implementing a stack in GPU is terribly inefficient. For this reason, in [Ernst et al. 2004], the authors present a multi-stack data structure that matches perfectly with SIMD architecture. Figure 1 shows a multi-stack with nine stacks.

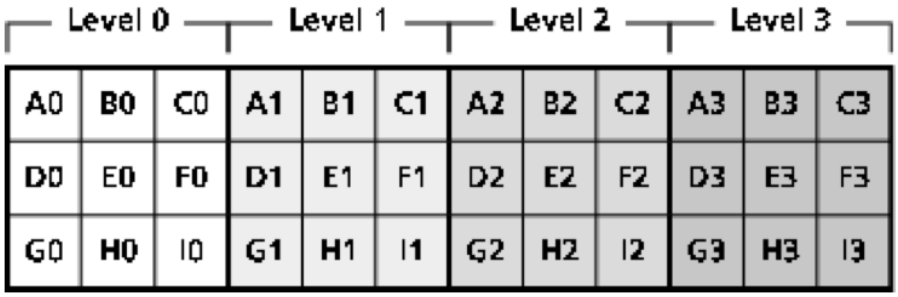

Fig. 1. A GPU implementations for stack, storing nine stacks with four levels. [Ernst et al. 2004]

To implement a multi-stack data structure, a N*N*K array is allocated to store the stack elements, where N*N is the number of stack in the multi-stack and K is the maximum depth of the stack. Another N*N array is also needed to store the stack pointers. When conducting the push or pop operation, each SIMD core can be responsible for each stack, and change the stack pointers.

3. SPACE PARTITIONING TREES

3.1 R Tree

R tree [Guttman 1984] is a commonly used data structure that indexes geometric objects based on their Minimum Bounding Rectangle (MBR). In [Kunjir and Manthramurthy 2009], the authors implement parallel R-tree search on CUDA GPUPU. Two structures required for the R-Tree search algorithm on the CPU are created and then load it into the device memory. Whenever a search query arrives, we send it to the GPU and the threads execute the CUDA R-Tree search using the R-Tree data that has been initially copied. If the R-Tree changes, the structures for the CUDA R-Tree algorithm need to be copied once again. The assumption in this paper is that this will not be often because spatial data does not change very frequently between queries.

The two structures that need to be stored in the GPU memory before hand are two arrays. The first is an array of MBR co-ordinates, which we call Coord. Coord[i] refers to the bottom-left and top-right co-ordinates of the i-th MBR in

the index. The second is an array of structures called Node. Node[i] consists of (mbrID, childNodes[t]), and represents the R-Tree node with id i. The mbrID is an index into the Coord array that gives the co-ordinates of the MBR of that node. childNodes is an array of size t, where t is the capacity of the tree. Each childNode element is an index into the Node array representing the children of the node i.

When the search query is made, a kernel call is made into the GPU. A thread block is launched. The threads in the thread block first copy the two structures into shared thread-block memory for efficiency. The threads next declare two more shared memory areas, for arrays of bits currentSearch and nextSearch. The length of each array equals N, the number of nodes in the R-Tree (which is less than the number of geometries indexed by the tree). These two bit-arrays are shared between all threads. The following is the algorithm:

(1) Clear the nextSearch array. (in parallel)
(2) Barrier
(3) For each bit i belonging to this thread, if currentSearch[i] is set, then for each child node j (looked up from childNodes) that overlaps with the query MBR:
    —If the child node is a leaf, mark it as part of the output.
    —If the child is not a leaf, mark it in the nextSearch array.
(4) Barrier
(5) Copy nextSearch into currentSearch (in parallel)
(6) Barrier

The algorithm executes iteratively, until the nextSearch array has no set bits. Then the output is copied back to CPU.

3.2 KD-tree

a kd-tree (short for k-dimensional tree) is a space-partitioning data structure for organizing points in a k-dimensional space. kd-trees are a useful data structure for several applications, such as searches involving a multidimensional search key.

[Zhou et al. 2008] presents a kd-tree construction algorithm for GPU architecture. Different from previous parallel kd-tree construction algorithm, this algorithm builds nodes completely in BFS order. The algorithm develops a special strategy for large nodes at upper tree levels so as to further exploit the fine-grained parallelism of GPUs. For these nodes, the algorithm parallelizes the computation over all geometric primitives instead of nodes at each level. Finally, in order to maintain kd-tree quality, the algorithm introduces novel schemes for fast evaluation of node split costs.

However, this above algorithm consumes excessive GPU memory, and this becomes a serious issue for interactive applications involving very complex models with more than a few million triangles. In [Hou et al. 2010], the author proposes to use the PBFS (partial breadth-first search) construction order to control memory consumption while maximizing performance. The paper applies the PBFS order to two hierarchy construction algorithms. The algorithm for kd-tree construction automatically balances between the level of parallelism and intermediate memory usage. With PBFS, peak memory consumption during construction can be efficiently controlled without costly CPU-GPU data transfer. The paper also develops

memory allocation strategies to effectively limit memory fragmentation. The resulting algorithm scales well with GPU memory and constructs kd-trees of models with millions of triangles at interactive rates on GPUs with 1GB memory. Compared with existing algorithms, this algorithm is an order of magnitude more scalable for a given GPU memory bound.

3.3 Octree

An octree is a tree data structure in which each internal node has exactly eight children. Octrees are most often used to partition a three dimensional space by recursively subdividing it into eight octants. The implementation of octrees in GPU includes two aspects. One is how to construct the octrees in parallel; The other is how to perform the search operation on octrees.

3.3.1 *Octree Construction.* The most important thing for constructing octrees is how to represent octrees in GPU. The common way is to represent octrees as the traditional trees[Benson and Davis 2002], [Ziegler et al. 2007]. The root node has eight pointers, each of which points to a child node. Recursively each child node has eight grandchild node, and so on until the child node is the leaf node. When constructing this kind of octrees, the algorithm follows the breadth search first order, and allows constructing multiple child nodes in parallel.

000	001	010	011	100	101	110	111
10000	1	10010	10011	100	101	110	111
1111000	10001	1111010	1111011	1001100	1001101	1001110	1001111
	1111001			11100	11101	11110	11111

Fig. 2. Hash representation of the quadtree. The hash function uses a 3 bits key (top row) to group the tree nodes (bottom rows)

An alternative octree representation is hashed octree[Lefebvre and Hoppe 2006]. In this structure, octree nodes are stored in a hash table. Instead of being accessed through pointers, each node can be located in the hash table by calculating its hash value. Figure 2 is an example of the hashed octree. The advantage of this representation is that any node of the tree can be directly accessed in constant time. The disadvantage is that the access time for each node is slower than pointer octree. To avoid collision, perfect hashing is used as the hashing scheme. As a result, any significant change in octree structure implies a complete rebuilding of the hashed octree.

For spatial ordering of the nodes and to generate the indexes for the hashed octree, the Morton code is used[Ajmera et al.] . This method is efficient to generate

unique index for each node, while offers god spatial locality and easy computation. Another advantage of Morton code is their hierarchical order, since it is possible to create a single index for each node, while preserving the tree hierarchy. The index can be calculated from the tree hierarchy, recursively when traversing the tree. The root has index 1, and the index of each child node is the concatenation of its parent index with the direction of their octant, coded over 3 bits. The bottom-up traversal is also possible, as if to find the parent index we only have to truncate the last 3 bits of a child index.

3.3.2 *Octree Search.* A direct search procedure in an octree returns the leaf whose cube contains a given position in space. In pointer octrees, a search can only be done starting from the root node and traversing hierarchically the tree until the desired leaf is reached. This method has complexity of log8(n), and O(n) at worse case. The algorithm below shows a search in a pointer octree. In this method, the position and size of each traversed cube can be directly deduced from the recursion. In [Castro et al. 2008], the author proposed optimized searches that offer a different access method. Since the leaves are the most distant nodes from the root node, it is better to start from a node closer to the desired leaves than from the root node. However, to access a random node in the hashed octree, we need its Morton code computable from position and depth. We know the position from the search input, but the depth must be estimated. This depth is estimated by the weighted median of the expected depth, since it minimizes the number of traversal operations.

3.4 Bounding Volume Hierarchy

A bounding volume hierarchy (BVH) is a tree structure on a set of geometric objects. All geometric objects are wrapped in bounding volumes that form the leaf nodes of the tree. These nodes are then grouped as small sets and enclosed within larger bounding volumes. These, in turn, are also grouped and enclosed within other larger bounding volumes in a recursive fashion, eventually resulting in a tree structure with a single bounding volume at the top of the tree.

[Hou et al. 2010] presents an algorithm is out-of-core BVH (bounding volume hierarchy) construction for very large scenes based on the PBFS construction order. At each iteration, all constructed nodes are dumped to the CPU memory, and the GPU memory is freed for the next iterations use. In this way, the algorithm is able to build trees that are too large to be stored in the GPU memory. Experiments show that this algorithm can construct BVHs for scenes with up to 20M triangles, several times larger than previous GPU algorithms.

[Lauterbach et al. 2009] presents two BVH construction algorithms. One is called LBVH. It uses a linear ordering derived from spatial Morton codes to build hierarchies extremely quickly and with high parallel scalability. The other algorithm is a top-down approach that uses the surface area heuristic (SAH) to build hierarchies optimized for fast ray tracing.

Figure 3 shows a 2-D example of BVH construction. The construction algorithm first picks the barycenter of each triangle to represent this triangle. Then it quantizes each of the three coordinates of the representative points into k-bit integers. The 3k-bit Morton code for a point is constructed by interleaving the successive bits of these quantized coordinates. To lay out these points in order along a Mor-

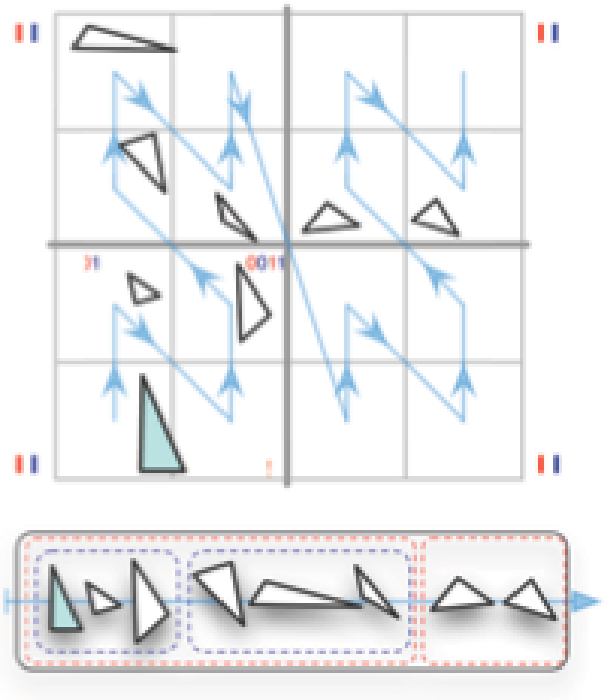

Fig. 3. Example 2-D Morton code ordering of triangles with the first two levels of the hierarchy [Lauterbach et al. 2009]

ton curve, the algorithm sorts the representative points in increasing order of their Morton codes. With the sorted primitive sequence, the algorithm also determines what levels each adjacent pair of primitives should be split into separate nodes, and results in a list of pairs with each pair representing the index of the primitives and the levels of splits between the primitives. Resort this split list by the level of splits and the information for constructing the BVH is ready. Each step in this algorithm can be massively paralleled, thus making this algorithm very fast although the built BVH is not optimized.

Instead of pursuing the fast running time, the other algorithm focuses on the performance of BVH. When partitioning the triangles,the algorithm tries different split scheme, calculates the cost for different schemes and choose the split scheme with minimum cost. The paper also proposes the third algorithm that combines these two algorithms ,and this hybrid algorithm has the scalability of the LBVH algorithm and performance optimizations in the SAH algorithm.

4. HASH TABLE

Hash table is an effective data structure for implementing dynamic set operation such as insert, search or delete. It uses a hash function to calculate the slot for each key. Ideally, each key has its unique slot. The time complexity for search operation

is O(1). In reality, two keys might hash to the same slot. We call this situation collision. There are two common ways for resolving such collisions. One is called chaining. In chaining, all the elements that hash to the same slot were put into a linked list. As a result, for different slots, the length of the linked list is different. The time complexity for search operation is no longer O(1). Another alternative way for resolving collisions is called open address. In open addressing, all elements are stored in the hash table itself. If the slot that a key hash to is occupied by another key, the open addressing algorithm will compute the next available slot for that key.

However, the chaining or open addressing method does not fit into the highly parallel environment of the GPU for two reason:

—Synchronization: algorithms for populating a traditional hash table tend to involve sequential operations. Chaining for example, requires serialization of access to the list structure since multiple items might be added to the same linked list at the same time.

—Variable work per access: Take open addressing for example. The number of probes required to look up an item in typical sequential hash tables varies per query. Variable work would lead to inefficiency on the GPU because the SIMD cores force all threads to wait for the worse-case number of probes.

In [Lefebvre and Hoppe 2006], among the first to use the GPU to access a hash table, the authors addressed the issue of synchronization and variable lookup time by using perfect hash table. Perfect hash table is a two-level hashing scheme over a static set of elements that guarantees no collision and thus the O(1) search time in the worse case. Since there are exactly two memory accesses and no need for synchronization, perfect hash table is perfect for SIMD architecture. The limitation of this methods lies on the fact all the elements in the hash table must be static. If any or all of the data items in the hash table change, the hash table has to be rebuilt. An alternative way to obtain random access to sparse data on GPU is to sort the key using radix sorting [Satish et al. 2009], and then do the binary search on the sorted keys.

In [Alcantara et al. 2009], the authors adopt cuckoo hashing [Pagh and Rodler 2001]. The basic idea of cuckoo hashing is to use two hash functions instead of only one. This provides two possible locations in the hash table for each key. In one of the commonly used variants of the algorithm, the hash table is split into two smaller tables of equal size, and each hash function provides an index into one of these two tables. When a new key is inserted, a greedy algorithm is used: The new key is inserted in one of its two possible locations, "kicking out", that is, displacing, any key that might already reside in this location. This displaced key is then inserted in its alternative location, again kicking out any key that might reside there, until a vacant position is found, or the procedure enters an infinite loop. In the latter case, the hash table is rebuilt in-place using new hash functions. Figure 4 illustrates the cuckoo hashing insertion.

To parallelize the cuckoo hashing, the authors use three sub-tables (d = 3). In the first iteration, they attempt to store every item into the first sub-table, T1, by writing each item into its position in the table simultaneously. The algorithm requires that exactly one write succeeds when collisions occur, and that every thread

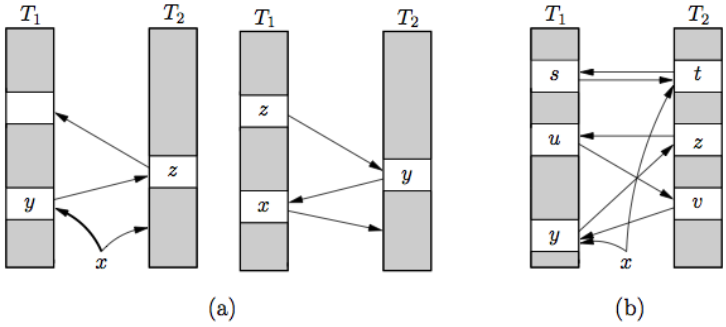

Fig. 4. Examples of Cuckoo Hashing insertion. Arrows show possibilities for moving keys. (a) Key x is successfully inserted by moving keys y and z from one table to the other. (b) Key x cannot be accommodated and a rehash is necessary. [Pagh and Rodler 2001]

should be able to tell which one succeeded. The items that fail attempt to write themselves into T2 in the second iteration. Those that fail proceed to T3. Finally, those that fail in T3 return to T1, evict the item occupying the location they want to occupy, and again try to write themselves into the now-empty locations. The evicted items, and those items which failed to find a location in T1, continue to T2. The iterations continue until all items are stored, or until a maximum number of iterations occur, in which case we decide that we have had an unfortunate choice of hash functions and we restart the process. The authors report this method performs better than the previous one.

5. IMPLEMENTATION

Figure II lists GPU implementations of different data structures. Because GPU does not support dynamic memory allocation, array is the only choice for representing data structures. Compared to the data structure representation in CPU, the biggest difference for the data structure representation in GPU is the use of static pointers, i.e. the use of array index as pointers. There are also some innovative representation for data structures in GPU, such as the one that uses hash calculation instead of pointers to locate other nodes.

Many implementations run using only a block of threads. One reason is that the threads in the same block can use shared memory to communicate, which is much faster than global memory. The other reason is that CUDA does not support global synchronization, but do for thread in the same block.

6. CONCLUSION

During the collections of paper relating to concurrent data structure in GPU, we realize that the development of concurrent data structure in GPU has just started, and has a lot of space for further research. The related papers we can find is very limited, and most of them are from computer graphic field. For example, half of the papers are about kd-tree, bounding volume hierarchy or octrees. These data

Linked List	[Yang et al. 2010]
Stack	[Ernst et al. 2004]
R-tree	[Kunjir and Manthramurthy 2009]
kd-tree	[Zhou et al. 2008], [Hou et al. 2010], [Popov et al. 2007]
Octree	[Benson and Davis 2002], [Ziegler et al. 2007], [Ajmera et al.], [Castro et al. 2008], [Sun et al. 2008], [Lefebvre et al. 2005], [Zhou et al. 2010]
Decision tree	[Grahn et al. 2010], [Sharp 2008]
Bounding Volume Hierarchy	[Lauterbach et al. 2009]
Hash table	[Lefebvre and Hoppe 2006], [Alcantara et al. 2009], [Pagh and Rodler 2001]
Graphs	[Luo et al. 2010], [Vineet et al. 2009]

Table II. The list of data structures

structures are designed to solve the ray tracing related problems.

Most papers focus on how to utilize the GPU to construct the data structure and carry out the search operation in parallel. The basic strategy is to modify the algorithm to make it easier to be paralleled, or reduce the problems into a classical parallel-able problem. Therefore, the currently implemented data structures are mostly those which can built once and then be used for following query operations, like kd-tree, hash table, graphs. The missing data structure in the above list is the data structures like heap, binary search tree that are built dynamically. This make senses because GPU is more suitable for collective operation than single operation. But we think some modification to current algorithm can make it possible to utilize the GPU huge computation power even for single operation, and enable a serial of data structures in GPU. To achieve that, a locking scheme for GPU might also need to be figured out.

Another concern when implementing concurrent data structure in GPU is the limit of GPU memory. Some of the papers aim to accommodate the GPU memory with the huge data. Reducing the communication of GPU and CPU and letting GPU do as much as work is another research goal.

APPENDIX A: Classification

—Introduction and Background: [Breitbart],[Owens et al. 2007],[Lefohn et al. 2006]

—Linked List: [Yang et al. 2010]

—Stack: [Ernst et al. 2004]

—R-tree: [Kunjir and Manthramurthy 2009]

—kd-tree: [Zhou et al. 2008], [Hou et al. 2010], [Popov et al. 2007]

—Octree: [Benson and Davis 2002], [Ziegler et al. 2007], [Ajmera et al.], [Castro et al. 2008], [Sun et al. 2008], [Lefebvre et al. 2005], [Zhou et al. 2010]

—Decision tree: [Grahn et al. 2010], [Sharp 2008]

—Bounding Volume Hierarchy: [Lauterbach et al. 2009]

—Hash table: [Lefebvre and Hoppe 2006], [Alcantara et al. 2009], [Pagh and Rodler 2001]

—Graphs: [Luo et al. 2010], [Vineet et al. 2009]

—Binary tree: [Zhou et al. 2008], [Hou et al. 2010], [Popov et al. 2007], [Lauterbach et al. 2009],[Grahn et al. 2010], [Sharp 2008]

—Other tree: [Kunjir and Manthramurthy 2009],[Benson and Davis 2002], [Ziegler et al. 2007], [Ajmera et al.], [Castro et al. 2008], [Sun et al. 2008], [Lefebvre et al. 2005], [Zhou et al. 2010]

—Data structure construction: [Lefebvre and Hoppe 2006], [Alcantara et al. 2009], [Pagh and Rodler 2001], [Zhou et al. 2008], [Hou et al. 2010],[Ziegler et al. 2007],[Ajmera et al.],[Sun et al. 2008],[Zhou et al. 2010],[Lauterbach et al. 2009]

—Data structure query: [Kunjir and Manthramurthy 2009], [Popov et al. 2007],[Castro et al. 2008],[Grahn et al. 2010], [Sharp 2008]

APPENDIX B: Annotated Bibliography

—Fast, Parallel, GPU-based Space Filling Curves and Octrees [Ajmera et al.]

Space Filling Curves (SFC) are particularly useful in linearization of data living in two and three dimensional spaces and have been used in a number of applications in scientific computing, and visualization. The paper provides a parallel implementation of SFCs and octrees on GPUs that rely on algorithms designed to minimize or eliminate communications.

—Real-time parallel hashing on the GPU [Alcantara et al. 2009]

The paper demonstrates an efficient data-parallel algorithm for building large hash tables of millions of elements in real-time. The paper combines two parallel algorithms for the construction: a classical sparse perfect hashing approach, and cuckoo hashing, which packs elements densely by allowing an element to be stored in one of multiple possible locations. Experiment result shows that this hybrid algorithm has the realtime performance.

—Octree textures [Benson and Davis 2002]

This paper proposes the use of a new kind of texture based on an octree, which needs no parameterization other than the surface itself, and yet has similar storage requirements to 2D maps. In addition, it offers adaptive detail, regular sampling over the surface, and continuity across surface boundaries. The paper addresses texture creation, painting, storage, processing, and rendering with octree textures.

—Statistical optimization of octree searches [Castro et al. 2008]

This paper proposes to estimate the depth of an arbitrary node through a statistical optimization of the average cost of search procedures. Since the highest costs of these algorithms are obtained when starting from the root, this method improves on both the memory footprint by the use of hashed octrees, and execution time through the proposed optimization.

—Stack implementation on programmable graphics hardware [Ernst et al. 2004]

The paper presents a technique that allows the implementation of a stack on programmable graphics hardware, using textures and fragment shaders. This development enables a whole new class of GPU algorithms, including recursive functions on complex data structures.

—A CUDA Implementation of Random Forests-Early Results [Grahn et al. 2010]

This paper presents a GPU-based parallel implementation of the Random Forests algorithm. An experimental comparison between the CUDA-based algorithm (CudaRF), and state-of-the-art parallel (FastRF) and sequential (LibRF) Random forests algorithms shows that CudaRF outperforms both FastRF and LibRF for the studied classification task.

—Memory-scalable gpu spatial hierarchy construction [Hou et al. 2010]

In this paper, the authors propose to use the PBFS (partial breadth-first search) construction order to control memory consumption while maximizing performance. Two hierarchy construction algorithms are applied with PBFS order. The first algorithm is for kd-trees that automatically balances between the level of parallelism and intermediate memory usage. The second algorithm is for out-of-core BVH (bounding volume hierarchy) construction for very large scenes based on the PBFS construction order.

—Using graphics processing in spatial indexing algorithms [Kunjir and Manthramurthy 2009]

In this paper, the authors explore the use of graphics processing capabilities to speed up spatial indexing in spatial databases. The authors implement parallel R-Tree search using NVidia's CUDA GPGPU architecture and also added a

geometry intersection test in the PostGIS extension of the PostgreSQL database engine using the OpenGL framework.

—Fast BVH construction on GPUs [Lauterbach et al. 2009]

The paper presents two novel parallel algorithms for rapidly constructing bounding volume hierarchies on manycore GPUs. The first uses a linear ordering derived from spatial Morton codes to build hierarchies extremely quickly and with high parallel scalability. The second is a top-down approach that uses the surface area heuristic (SAH) to build hierarchies optimized for fast ray tracing. Both algorithms are combined into a hybrid algorithm that removes existing bottlenecks in the algorithm for GPU construction performance and scalability leading to significantly decreased build time.

—Perfect spatial hashing [Lefebvre and Hoppe 2006]

The paper explores using hashing to pack sparse data into a compact table while retaining efficient random access. Specifically, the researchers design a perfect multidimensional hash function one that is precomputed on static data to have no hash collisions. Because the hash function makes a single reference to a small offset table, queries always involve exactly two memory accesses and are thus ideally suited for parallel SIMD evaluation on graphics hardware. Whereas prior hashing work strives for pseudorandom mappings, we instead design the hash function to preserve spatial coherence and thereby improve runtime locality of reference. The paper demonstrates numerous graphics applications including vector images, texture sprites, alpha channel compression, 3D-parameterized textures, 3D painting, simulation, and collision detection.

—Octree textures on the GPU [Lefebvre et al. 2005]

This paper details how to implement octree textures on today's GPUs. The octree is directly stored in texture memory. It also discuss the tradeoffs between performance, storage efficiency and rendering quality.

—An effective GPU implementation of breadth-first search [Luo et al. 2010]

In this paper, the authors present a new GPU implementation of BFS that uses a hierarchical queue management technique and a three-layer kernel arrangement strategy. It guarantees the same computational complexity as the fastest sequential version and can achieve up to 10 times speedup.

—Stackless KD-Tree Traversal for High Performance GPU Ray Tracing [Popov et al. 2007]

In this paper the authors present a novel packet ray traversal implementation that completely eliminates the need for maintaining a stack during kd-tree traver-

sal and that reduces the number of traversal steps per ray. While CPUs benefit moderately from the stackless approach, it improves GPU performance significantly. This algorithm achieves a peak performance of over 16 million rays per second for reasonably complex scenes, including complex shading and secondary rays.

—Implementing decision trees and forests on a gpu [Sharp 2008]

The paper describes a method for implementing the evaluation and training of decision trees and forests entirely on a GPU, and show how this method can be used in the context of object recognition.

—Interactive relighting of dynamic refractive objects [Sun et al. 2008]

The paper presents a new technique for interactive relighting of dynamic refractive objects with complex material properties. The authors describe their technique in terms of a rendering pipeline in which each stage runs entirely on the GPU. The rendering pipeline converts surfaces to volumetric data, traces the curved paths of photons as they refract through the volume, and renders arbitrary views of the resulting radiance distribution. The rendering pipeline is fast enough to permit interactive updates to lighting, materials, geometry, and viewing parameters without any precomputation.

—Fast minimum spanning tree for large graphs on the gpu [Vineet et al. 2009]

The paper presents a minimum spanning tree algorithm on Nvidia GPUs under CUDA, as a recursive formulation of Boruvkas approach for undirected graphs. The authors implement it using scalable primitives such as scan, segmented scan and split. The irregular steps of supervertex formation and recursive graph construction are mapped to primitives like split to categories involving vertex ids and edge weights.

—Real-Time Concurrent Linked List Construction on the GPU [Yang et al. 2010]

The paper introduces a method to dynamically construct highly concurrent linked lists on modern graphics processors. Once constructed, these data structures can be used to implement a host of algorithms useful in creating complex rendering effects in real time. The authors present a straightforward way to create these linked lists using generic atomic operations available in APIs such as OpenGL 4.0 and DirectX 11. The authors also describe several possible applications of our algorithm.

—Data-parallel octrees for surface reconstruction [Zhou et al. 2010]

The paper presents the first parallel surface reconstruction algorithm that runs entirely on the GPU. Like existing implicit surface reconstruction methods, this

algorithm first builds an octree for the given set of oriented points, then computes an implicit function over the space of the octree, and finally extracts an isosurface as a water-tight triangle mesh. A key component of the algorithm is a novel technique for octree construction on the GPU. This technique builds octrees in real-time and uses level-order traversals to exploit the fine-grained parallelism of the GPU. Moreover, the technique produces octrees that provide fast access to the neighborhood information of each octree node, which is critical for fast GPU surface reconstruction.

—Real-time kd-tree construction on graphics hardware [Zhou et al. 2008]

The paper presents an algorithm for constructing kd-trees on GPUs. This algorithm achieves real-time performance by exploiting the GPUs streaming architecture at all stages of kd-tree construction. Unlike previous parallel kd-tree algorithms, this method builds tree nodes completely in BFS (breadth-first search) order. The paper also develops a special strategy for large nodes at upper tree levels so as to further exploit the fine-grained parallelism of GPUs. For these nodes, the algorithm parallelizes the computation over all geometric primitives instead of nodes at each level. Finally, in order to maintain kd-tree quality, the algorithm introduces novel schemes for fast evaluation of node split costs.

—Real-time quadtree analysis using HistoPyramids [Ziegler et al. 2007]

This research article demonstrates how graphics hardware can be utilized to build region quadtrees at unprecedented speeds. To achieve this, a data-structure called HistoPyramid registers the number of desired image features in a pyramidal 2D array. Then, this HistoPyramid is used as an implicit indexing data structure through quadtree traversal, creating lists of the registered image features directly in GPU memory, and virtually eliminating bus transfers between CPU and GPU.

ACKNOWLEDGMENTS

We wish to thank Dr. Sushil Prasad and other classmates in parallel algorithm class for the constructive comments and valuable feedback.

REFERENCES

Ajmera, P., Goradia, R., Chandran, S., and Aluru, S. Fast, Parallel, GPU-based Space Filling Curves and Octrees. *I3D2008*.

Alcantara, D., Sharf, A., Abbasinejad, F., Sengupta, S., Mitzenmacher, M., Owens, J., and Amenta, N. 2009. Real-time parallel hashing on the GPU. *ACM Transactions on Graphics (TOG) 28,* 5, 1–9.

Benson, D. and Davis, J. 2002. Octree textures. In *ACM Transactions on Graphics (TOG)*. Vol. 21. ACM, 785–790.

Breitbart, J. Data structure design for GPU based heterogeneous systems. In *High Performance Computing & Simulation, 2009. HPCS'09. International Conference on*. IEEE, 44–51.

Castro, R., Lewiner, T., Lopes, H., Tavares, G., and Bordignon, A. 2008. Statistical optimization of octree searches. In *Computer Graphics Forum*. Vol. 27. Wiley Online Library, 1557–1566.

Ernst, M., Vogelgsang, C., and Greiner, G. 2004. Stack implementation on programmable graphics hardware. In *Proceedings of Vision, Modeling, and Visualization*. 255–262.

Flynn, M. 1972. Some computer organizations and their effectiveness. *Computers, IEEE Transactions on 100,* 9, 948–960.

Grahn, H., Lavesson, N., Lapajne, M., and Slat, D. 2010. A CUDA Implementation of Random Forests-Early Results. In *Third Swedish Workshop on Multi-core Computing*. Chalmers Institute of Technology.

Guttman, A. 1984. *R-trees: a dynamic index structure for spatial searching*. Vol. 14. ACM.

Hou, Q., Sun, X., Zhou, K., Lauterbach, C., and Manocha, D. 2010. Memory-scalable gpu spatial hierarchy construction. *IEEE Transactions on Visualization and Computer Graphics*.

Kunjir, M. and Manthramurthy, A. 2009. Using graphics processing in spatial indexing algorithms. *Research report, Indian Institute of Science, Database Systems Lab*.

Lauterbach, C., Garland, M., Sengupta, S., Luebke, D., and Manocha, D. 2009. Fast BVH construction on GPUs. In *Computer Graphics Forum*. Vol. 28. Wiley Online Library, 375–384.

Lefebvre, S. and Hoppe, H. 2006. Perfect spatial hashing. In *ACM SIGGRAPH 2006 Papers*. ACM, 579–588.

Lefebvre, S., Hornus, S., and Neyret, F. 2005. Octree textures on the GPU. *GPU gems 2*, 595–613.

Lefohn, A., Sengupta, S., Kniss, J., Strzodka, R., and Owens, J. 2006. Glift: Generic, efficient, random-access GPU data structures. *ACM Transactions on Graphics (TOG) 25,* 1, 60–99.

Luo, L., Wong, M., and Hwu, W. 2010. An effective GPU implementation of breadth-first search. In *Proceedings of the 47th Design Automation Conference*. ACM, 52–55.

Owens, J., Luebke, D., Govindaraju, N., Harris, M., Kr
"uger, J., Lefohn, A., and Purcell, T. 2007. A Survey of General-Purpose Computation on Graphics Hardware. In *Computer graphics forum*. Vol. 26. Wiley Online Library, 80–113.

Pagh, R. and Rodler, F. 2001. Cuckoo hashing. *AlgorithmsESA 2001*, 121–133.

Popov, S., G
"unther, J., Seidel, H., and Slusallek, P. 2007. Stackless KD-Tree Traversal for High Performance GPU Ray Tracing. In *Computer Graphics Forum*. Vol. 26. Wiley Online Library, 415–424.

Satish, N., Harris, M., and Garland, M. 2009. Designing efficient sorting algorithms for manycore GPUs.

Sharp, T. 2008. Implementing decision trees and forests on a gpu. *Computer Vision–ECCV 2008*, 595–608.

Sun, X., Zhou, K., Stollnitz, E., Shi, J., and Guo, B. 2008. Interactive relighting of dynamic refractive objects. In *ACM SIGGRAPH 2008 papers*. ACM, 1–9.

Vineet, V., Harish, P., Patidar, S., and Narayanan, P. 2009. Fast minimum spanning tree for large graphs on the gpu. In *Proceedings of the Conference on High Performance Graphics 2009*. ACM, 167–171.

Yang, J., Hensley, J., Gr
"un, H., and Thibieroz, N. 2010. Real-Time Concurrent Linked List Construction on the GPU. In *Computer Graphics Forum*. Vol. 29. Wiley Online Library, 1297–1304.

Zhou, K., Gong, M., Huang, X., and Guo, B. 2010. Data-parallel octrees for surface reconstruction. *IEEE Transactions on Visualization and Computer Graphics*.

Zhou, K., Hou, Q., Wang, R., and Guo, B. 2008. Real-time kd-tree construction on graphics hardware. In *ACM Transactions on Graphics (TOG)*. Vol. 27. ACM, 126.

Ziegler, G., Dimitrov, R., Theobalt, C., and Seidel, H. 2007. Real-time quadtree analysis using HistoPyramids. *Real-Time Image Processing 2007 6496*.