Design and Implementation of a Parallel Priority Queue on Many-core Architectures

Xi He, Dinesh Agarwal, and Sushil K. Prasad
Department of Computer Science
Georgia State University
Email: xhe8, dagarwal2@student.gsu.edu, sprasad@gsu.edu

***Abstract*—An efficient parallel priority queue is at the core of the effort in parallelizing important non-numeric irregular computations such as discrete event simulation scheduling and branch-and-bound algorithms. GPGPUs can provide powerful computing platform for such non-numeric computations if an efficient parallel priority queue implementation is available. In this paper, aiming at fine-grained applications, we develop an efficient parallel heap system employing CUDA. To our knowledge, this is the first parallel priority queue implementation on many-core architectures, thus represents a breakthrough. By allowing wide heap nodes to enable thousands of simultaneous deletions of highest priority items and insertions of new items, and taking full advantage of CUDA's data parallel SIMT architecture, we demonstrate up to 30-fold absolute speedup for relatively fine-grained compute loads compared to optimized sequential priority queue implementation on fast multicores. Compared to this, our optimized multicore parallelization of parallel heap yields only 2-3 fold speedup for such fine-grained loads. This parallelization of a tree-based data structure on GPGPUs provides a roadmap for future parallelizations of other such data structures.**

I. Introduction

A priority queue is an important abstract data structure for many non-numeric computations such as discrete event simulation, branch-and-bound algorithms, and multi-processor scheduling. Here is a scenario of how a priority queue is typically used in a practical application. A large complex network simulator is a software that can simulate the behavior of a large network (communication network, logic circuit, immune system, or social network) under certain network events. A common network event would be a message passing from one compute node to another compute node. The important task for the network simulator is to access the network events in the order of the events' timestamps, and process them. For efficient execution, these network events are stored in a priority queue with the events' timestamps as priority. For each iteration, the earliest network event is popped from the priority queue for processing. Processing of network events can trigger new network events, as a message will continue to route to the next compute node after it has been carried to an intermediate compute node, these new network events will be inserted into the priority queue for future processing. The process is repeated so that the network's behavior can be simulated and evaluated. What is fundamentally so difficult about these irregular discrete problems? As we can see, many of the priority queue application are computation-intensive, but the parallelism is regular neither in space nor on time. Although the network may have millions of nodes (or agents), activity may only be in a few nodes, say 1% of them, at any given step. What is worse, this set of active nodes, which can be in tens of thousands, can be arbitrarily scattered across the network. Likewise, in a state space search of an exponential space, say for a SAT solver or a game of Chess, the set of high-potential nodes (possibly closer to the goals) in any given step is a tiny set, again scattered dynamically in space as the computation unfolds. The network simulator can process multiple independent network events simultaneously [1], but one of the primary problems is how to efficiently retrieve multiple earliest network events and insert new events to the priority queue in parallel.

With the emergence of GPGPU (General Purpose Graphics Processing Unit), a powerful computing platform is readily available to scientists and engineers. Unfortunately, without the support of an efficient parallel priority queue, a significant class of related applications is not able to run on GPGPUs. Considering the SIMT nature and inefficient implementation of locks, developing an efficient parallel priority queue on GPGPUs has been an outstanding challenge.

Our main contribution lies in our design of a parallel heap based priority queue data structure for many-core architectures. We follow this up with an efficient implementation of the parallel heap system employing CUDA. Parallel heap is a wide-node heap data structure which was shown by Prasad [2], [3] to be the first theoretically efficient data structure to enable O(p) operations in O(log n) time for $p \leq n$, using p processors on a n-sized heap on EREW PRAM shared memory model of computation. This work thus also represents a successful PRAM to GPGPUs port of a complex data structure. The development of the parallel heap system also enables the aforementioned class of priority queue based applications on GPGPUs.

Most parallelizations of data structures behave very well for coarse grained applications when the frequency of updates of the data structure is low. The effectiveness of parallelization is validated when the data structure is stress-tested with finer-gained applications, with frequent updates, creating severe contention among the competing processes/threads. Our parallelization on CUDA architectures is designed specifically targeting the fine-grained application on CUDA's SIMT architecture. We expose larger amount of concurrency of the underlying application in a data parallel fashion, allowing thousands of insertions and deletions, and concurrent execution of large

number of highest priority items with relatively finer grain. We demonstrate that for such compute loads one can obtain 20 to 30 fold absolute speedups on GPUs. Compared to this, our optimized multicore parallelization of parallel heap yields only 2-3 fold speedup for such fine-grained loads.

The paper's organization is as follows: Section II briefly reviews the related work. After introducing the basic ideas of the parallel heap and its key operations in Section III, the system design and implementation is described in Section IV. In Section V, comprehensive experiments conducted to evaluate the system performance are described. We offer some conclusions and future roadmap in Section VI.

II. Related Work

A. Nvidia GPGPUs and CUDA

Modern Nvidia GPGPUs are fully programmable graphic processing units. A GPGPU consists of an array of parallel processors which are often referred to as streaming multiprocessors (SM). For example, in Nvidia's 480 GTX chip, a single SM consists of 32 scalar processors with each scalar processor equipped with certain amounts of registers. Each SM also has 48 KB on-chip memory which has lower access latency and higher bandwidth compared to the global memory which is accessible to every SM, and has larger size. The SMs employ a SIMT (Single Instruction Multiple Thread) architecture. A group of 32 threads called a warp is the minimum execution unit. Once scheduled on a SM, the threads in a warp share the same instruction and can execute in a fairly synchronous fashion.

CUDA (Compute Unified Device Architecture) is a parallel computing architecture developed for parallel processing and enables the programmers to access to the instruction sets and memory in the Nvidia GPGPUs. A typical CUDA program is organized into host programs and one or more parallel kernel programs. The host programs are executed on CPU while the parallel kernel programs run on GPGPUs.

There are some performance considerations while programming using CUDA. The first one is the number of threads spawned in a CUDA program. When the instructions executed by the threads in a warp must wait for a long-latency operation such as the global memory read, other warps are scheduled to be executed. The mechanism of tolerating the latency of expensive operations using work from other warps is often referred to as latency hiding. The mechanism can work due to the fact that the thread management in GPGPUs is implemented in hardware and is extremely efficient. Therefore, a CUDA program should be designed to spawn as many threads as possible so that enough threads are available to hide the latency of expensive operations. The second consideration is the on-chip shared memory provided by the SM. Considering that the shared memory is much faster than the global memory but smaller, it is advantageous if a CUDA program can partition the data in the global memory into pieces that can fit into the shared memory, load these pieces into the shared memory one by one, process them, and then write the results back to the global memory.

B. Priority Queue

There are a number of priority queue implementations in the literature. The binary heap [4] is the most popular priority queue implementation. It can be built in $O(n)$ time, and its insertion or deletion can be completed in $O(log(N))$ time. If the priority queue does not need to support the union operation, the binary heap usually is the best candidate for a general-purpose comparison-based priority queue implementation. Another two important priority queue implementations are binomial heaps [5] and fibonacci heaps [6]. They are designed to support efficient union operations and therefore are also called mergable heaps. The time complexity of the union operation for two binomial heaps is $O(log(N))$.

There are three approaches to parallelize a priority queue. The first one is to speed up the individual queue operation using a small number of processors. For example, the parallel priority queue in [7] can support insertion and deletion in constant time with $O(logN)$ processors. The second approach is to support simultaneous insertion or deletion of k items where k is a constant. The parallel heap [3] is based on this approach. A parallel heap can insert $O(k)$ new items or delete the $O(k)$ highest items in $O(logN)$ time with $O(k)$ processors. The third approach is to allow concurrent insertions or deletions over the queue. The concurrent heap [8], a practical parallel priority queue on shared memory architecture, is an example of this approach. However, the maximum speedup a concurrent heap can achieve is bounded by $O(logN)$. Also, for a fine-grained application in which many insertions or deletions compete for the access of the heap, the heap will suffer severe memory contention and performance gets downgraded.

For the many-core architecture, a memory synchronization or lock is hard and inefficient to implement. So the priority queue implementation based on locking is not preferable. The parallel heap is built on the barrier concept, and since the kernel call in CUDA implicitly enforces a barrier on all threads, the parallel heap is more suitable for the CUDA system. Also, GPGPUs can be best utilized when there are enough threads to be executed. The insert and delete operations in a parallel heap with large heap nodes can provide considerable threads to fill the GPGPUs. Lastly, the concurrent kernel feature introduced in Nvidia Fermi-based GPGPUs can enable the parallel heap's pipeline schemes, and further enhance system's performance.

CUDA 4.0 and above can provide concurrent kernels support, allowing multiple independent kernels to be executed simultaneously and more efficiently, provided there are enough idle processors on the GPGPU. CUDA 4.0 can support at most 16 concurrent kernels whereas CUDA 5.0 allows up to 32 concurrent kernels. The concurrent kernels are managed by streams that can contain a number of kernels. When a CUDA program launches a kernel, it specifies which stream the kernel will be assigned to. Once assigned, the kernels in the same stream are executed in FIFO fashion while kernels in distinct streams are independent, and can run concurrently. CUDA also provides runtime APIs to synchronize between streams. The feature of concurrent kernels makes it possible to enable the parallel execution of priority queue applications and priority

queue maintenance so that the computing resources of the GPGPU can be fully utilized and the overall efficiency of the priority queue system is enhanced.

C. Parallel Data Structures on Many-core architectures

As GPGPUs are widely used in both industry and academics, the research into parallel data structures on many-core architectures have become very urgent and promising. A number of tree-based data structures, such as R-tree [9], KD-tree [10]–[12], Octree [13]–[19], decision tree [20], [21], and bounding volume hierarchy [22] have been ported to GPGPUs. Besides the tree-based data structures, hash tables on GPGPUs have also been discussed in [23]–[25].

The difficulty of porting data structures to GPGPU lies in how data structures' operations are implemented so that GPGPUs' streaming processors can be fully utilized and a good speedup can be achieved. GPGPUs' SIMT architecture, latency toleration mechanism, and inefficient synchronization are important factors to be considered. We classify data structures' constructions into two categories: online and offline constructions. An online construction refers to the construction in which the data structure is built by inserting new items iteratively as they arrive. On the contrary, an offline construction is that the data structure is built with all of the items available a priori. A fair amount of research effort has been put into the parallel offline construction considering the data parallel computation and synchronization-free feature in offline constructions. Parallel searches are another popular research topics for the similar reason.

However, to our knowledge, there is no prior research on dynamic update operations for parallel data structures on many-core architectures, due to the complications of synchronization, dead-lock, and other problems. The past solution for implementing update operations on many-core architectures is to simply rebuild the data structures [26]. The parallel heap data structure we present in this paper demonstrates a possible way for efficiently implementing parallel data structures' insert or delete operations on many-core architectures in an offline fashion.

III. Parallel Heap

A. Overview

A binary heap data structure [4] can be viewed as a full and complete binary tree where each node of the binary heap contains one item. Each item has an associated value and priority. For binary min-heaps, the smaller the value of an item, the higher its priority. The basic property of a binary min-heap is that the value of the item in each heap node is no larger than those in its children. Thus the item with the highest priority is always kept at the root node. Since insertion of a new item or deletion of the item with the highest priority might destroy such property, a "heapify" process is usually required to restore the binary heap's property after insertions or deletions.

A concurrent heap data structure [8] is simply a parallelized binary heap in which parallelism is obtained by concurrent insertion and deletion over the heap. Similar to the binary heap, each node in the concurrent heap contains one item and satisfies the basic heap property. Unlike the binary heap, the insert operation in concurrent heaps proceeds in a top-down fashion. An insert item moves along a unique path from the root to the target node. This adjustment allows both insert and delete operations to get executed in a pipelined fashion without worrying about "dead lock".

A parallel heap data structure [3] in some sense can be regarded as an improved concurrent heap with more considerations for fine-grained applications. A parallel heap with node capacity $r \geq 1$ is a complete binary tree such that

- each node (except the last node) contains r sorted keys.
- all r keys at a node have values less than or equal to the keys at its children.

Since there are r keys in a single heap node, the insert and delete operations over these keys can be aggregated and processed together in batches. Much more parallelism therefore can be extracted and exploited. Also, the pipelined insert and delete operations in a parallel heap can be regulated in a step-by-step manner, making the kernel-based CUDA platform very suitable for its implementation.

B. Insert and Delete Operation

A parallel heap's insert and delete operations are performed in a series of delete-insert cycles. Every delete-insert cycle takes care of the insertion of k ($k <= 2r$) new keys and the deletion of r keys with the highest priority. The following are three steps of execution in each delete-insert cycle.

1) Merge the k new keys with the r keys at the root node and sort these keys. Delete the first r sorted keys for the parallel heap application, and substitute the keys at the root node with the second r keys. The remaining keys are used to initiate a new insert-update process in the next step.
2) Initiate a new delete-update and insert-update process starting at the root node. Simultaneously, process the delete-update and insert-update processes at the even-level of the parallel heap.
3) Process the delete-update and insert-update processes at the odd-level of the parallel heap.

In Step 1, k new keys are input to the parallel heap.[1] If k is equal to and less than r, then no insert-update process will be initiated in Step 2. Particularly, when k is less than r, $(r-k)$ keys have to be retrieved from the end of the parallel heap or from an insert-update process heading towards the last node. If k is more than r and the last node does not have enough space to hold $k-r$ keys, then two insert-update processes in Step 2 are launched to insert the keys into two distinct heap nodes. Otherwise, only one insert-update is needed.

As the second r keys are placed at the root node, the heap's property might be destroyed. As a result, a delete-update process is initiated by merging the keys at the root node with those at its children. The smallest r keys are kept at the root node while the second r smallest keys are placed at the left child node if its largest key is bigger than that

[1] We assume $k <= 2r$, but other value can also work.

of the right child node, otherwise the second r smallest keys are placed at the right child node. Finally, the largest r keys are placed at the other child node. It can be proved that with such a placement, only the child node placed with the largest r keys might destroy the heap's property, so the delete-update process may continue on this node. The delete-update process repeats until it goes to the bottom of the heap or if the parallel heap property gets satisfied midway. Similarly, an insert-update process starts at the root node, and 'sinks' toward their target node at the bottom of the parallel heap after being repeatedly merged with the keys at the intervening nodes, and carrying down the larger keys each time. In Step 2 and Step 3 of the delete-insert cycle, each delete-update or insert-update process is moved down two levels in a parallel heap, so there are always multiple delete-update and insert-update processes coexisting in the parallel heap and carried out in a pipelined fashion for overall optimality.

Figure 1, 2, and 3 illustrate a delete and insert operation of a sample parallel heap. In Figure 1, four new keys 9, 27, 31,and 38 enter a 4-level parallel heap with $r = 2$. The four keys are merged with two keys, 18 and 23, at the root node in the buffer. The smallest 2 keys in the buffer, 9 and18, are deleted and transferred to the priority queue application. The next smallest 2 keys, 23 and 27, are placed at the root node and a delete-update process is initiated to maintain the destroyed heap property. The remaining keys, 31 and 38, are to be inserted into the parallel heap through two insert-update processes along the two insertion paths, as shown in Figure 2.

The delete-update process first restores the heap property at the root node by merging and replacing the keys at node 1 (root node), node 2, and node 3. After placing the largest keys, 31 and 32, on node 3, the heap property at node 3 is destroyed and the delete-update process has to continue to work on node 3, node 6 and node 7. Afterwards, the delete-update process is done since there are no children for node 6, and node 7. The first insert-update process proceeds by merging with keys at node 1, node 2, and node 5 and carrying the largest keys down each time. When the insert-update process is done, the key 38 is inserted into node 11. Similarly, the second insert-update process works with node 1, node 3, and node 6. The key 42 eventually goes to node 12. Figure 3 shows the parallel heap after the delete-update and insert-update operations. Note that the delete-update process only has one even-level and one odd-level merging, so it can be done in one delete-insert cycle. The two insert-update processes have two even-level and one odd-level merging, so it is done in two consecutive delete-insert cycles.

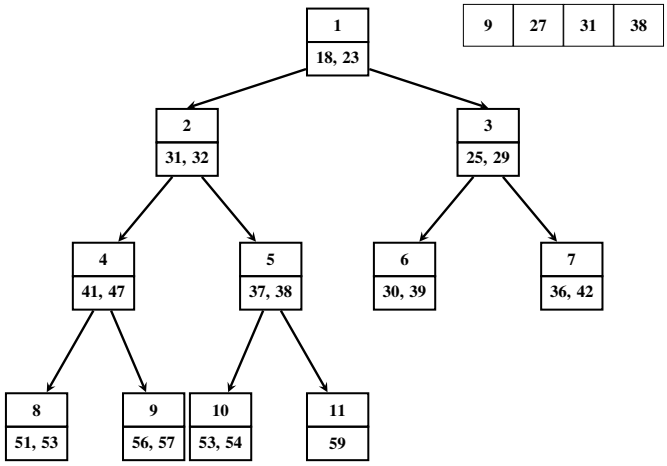

Fig. 1: A parallel heap with r=2. The upcoming delete-insert cycle has 4 input keys: 9, 27, 31, 38.

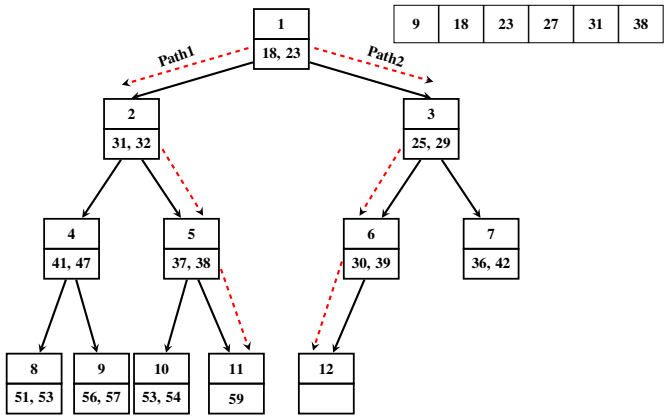

Fig. 2: The input keys are merged with the keys at the root node. One delete-update process and two insert-update processes are launched.

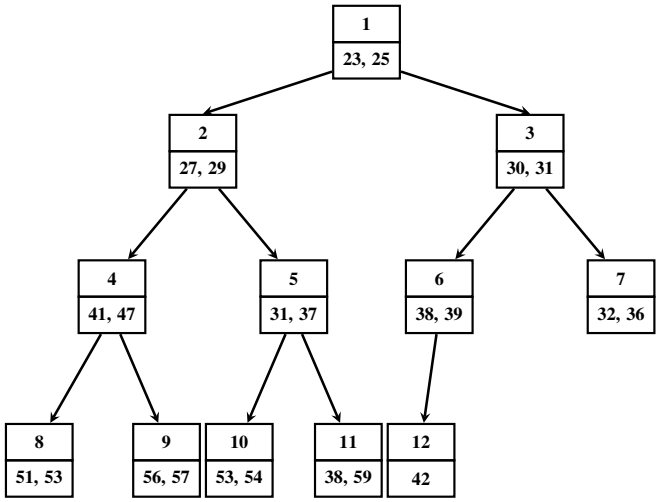

Fig. 3: The parallel heap after a delete and insert operation.

IV. System Design And Implementation

A. Overview

Our parallel heap system architected for CUDA platforms is composed of three major components: a controller, a parallel heap manager and a priority queue application, as shown in Figure 4. The controller is located on the CPU side while the other two components are on the GPU side. The controller is at the heart of the system acting as the mediator between the parallel heap manager and the priority queue application, and controlling the flow of the entire system. The parallel heap manager consists of a parallel heap and other related

data structures in the device memory, and a set of kernel functions that implements the interfaces of a priority queue and maintains the parallel heap data structure. The priority queue application notifies the controller when its output data, the set of newly-produced insert items, is ready, it then suspends itself and waits for the input data, the set of highest priority items to be deleted from the parallel heap. Then the controller asks the parallel heap manager to merge the new items from the priority queue application with items at the heap's root node, to sort them and to return the r smallest items to the priority queue application. Once done, the controller informs the priority queue application to resume with the r smallest data items from the parallel heap manager, and at the same time, requests the parallel heap manager to launch a new delete-insert cycle to maintain the parallel heap. The above process is repeated until the priority queue application is completed.

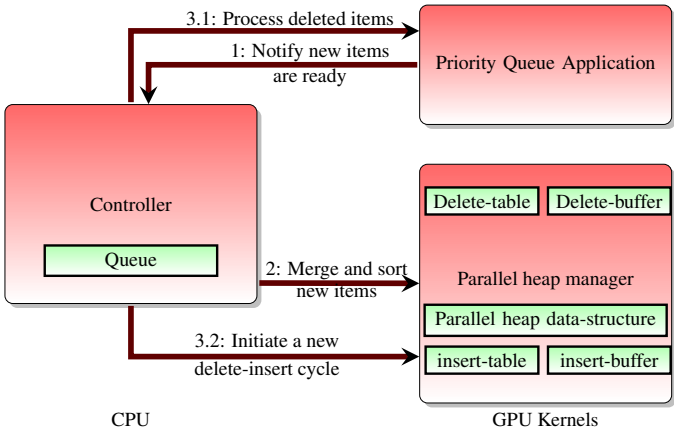

Fig. 4: System Architecture Control Flow

B. The Controller Component

The programming model of CUDA is different from that of the traditional CPU-based systems. In the CPU-based systems, programs are executed on CPU and related data structures are maintained in the main memory. In CUDA, however, most computation tasks are done on the GPGPU side in forms of kernel calls, and therefore their respective data structures need to be stored in the device memory. The programs running on the CPU are used to perform the sequential tasks and are combined with the programs on the GPGPU side to form a complete application. The controller in the parallel heap system is the component particularly required by the programming model of CUDA. It is responsible for managing the parallel heap manager and the priority queue application, and connecting these two components together. As shown in Figure 5, after the merging of the new items with the items at the root and the deletions of the smallest r items performed by the parallel heap manager, the controller needs to synchronize both the parallel heap manager and the priority queue application with a global barrier, and then initiate a new delete-insert cycle in the parallel heap manager and resume the priority queue application. The concurrent kernel feature of new CUDA-enabled GPGPUs make it possible to execute the parallel heap manager and the priority queue application concurrently, extracting additional parallelism for the parallel heap system. In our design, the kernel functions of the parallel heap manager and those of the priority queue application are assigned to distinct CUDA streams, with the kernel functions within a stream being executed in the FIFO order. The insert and delete operations in each delete-insert cycle of a parallel heap is implemented as a set of kernel functions with the kernel calls acting as stream-level barriers to enforce the pipelined update of the parallel heap.

The implementation of the controller is also very critical to the whole system's performance. One of the most important design considerations for the CUDA programs is the data parallel SIMT architecture of the GPGPUs. SIMT architectures require that all threads within a warp must execute the same instruction in any clock, and therefore the conditional branches are not preferred in CUDA programs. For example, in an *if-then-else* construct, if some threads in a warp take the *then* path and some the *else* path, two passes are needed for the execution of the construct. One pass will be used to execute all threads that take the *then* path and a pass will be used to execute the others. In the implementation of the parallel heap manager, however, a number of special cases such as the delete-update process of the last partially-filled heap node, the earlier terminated delete-update processes, etc., have to be considered. The kernel function will be full of conditional branches and its performance will suffer if all of these special cases are dealt with in one kernel function. Therefore we have seperate kernel functions implemented for each of these special cases and the controller determines which kernel functions should be executed in accordance with the current status of the parallel heap.

To allow the controller to better collaborate with the parallel heap manager, maintaining the status of the parallel heap within the controller is necessary even though the parallel heap data structure is stored in device memory. In each iteration, the number of new items produced can dynamically change the state of the heap, such as the total number of heap nodes and the number of levels in the heap. To maintain these critical information that many kernel calls rely on, a queue data structure is employed by the controller to keep track of these changes in our implementation. By pushing the number of new items in each iteration into the queue and popping the number of items as delete-update and insert-update processes get completed, the current status of the parallel heap is computed and maintained in the controller component.

C. Design for Delete-update and Insert-update Processes

As explained in Section III, there are usually one delete-update and up to two insert-update on-going processes on every two adjacent levels of the parallel heap. Two data structures (Figure 6) are designed to facilitate the concurrent execution of these delete-update processes. One is referred to as the *delete-table*. This table only has one column. Each row in the table corresponds to one delete-update process, and stores the index of the target heap node. The other data structure, *delete-buffer*, provides the working space for

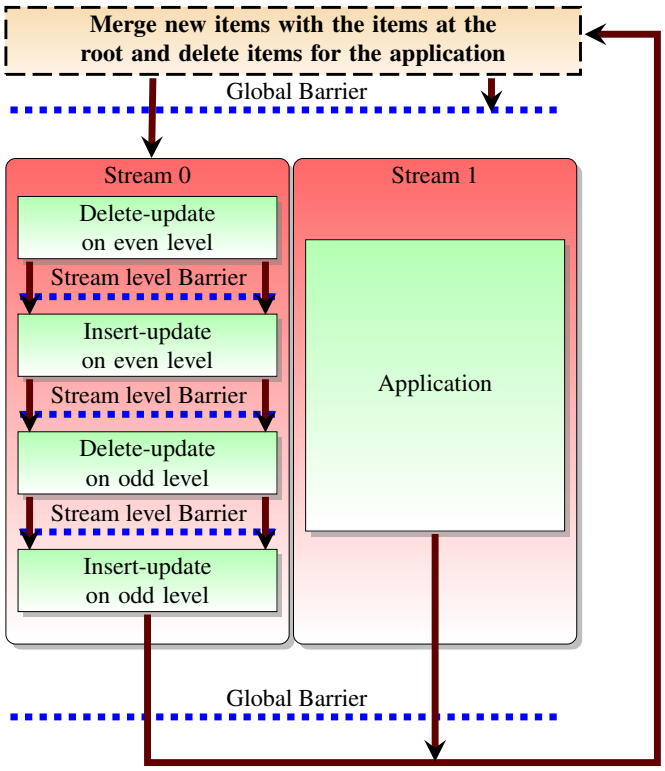

Fig. 5: The flow chart of the parallel heap system

the delete-update process. All of the delete-update processes are launched in a function call, in which each delete-update process is assigned a set of threads for execution. The delete-update processes read their respective rows in the *delete-table* for locating the target heap node, copy the items at the target heap node and its children into the *delete-buffer*, and sort the items in the *delete-buffer*. Once done, the items are written back to the appropriate heap nodes and the *delete-table* is updated for the next processing. The design can also be improved by keeping the largest r items in the *delete-buffer* after the completion of the current processing as these items will be reused in the next iteration, and thus reducing the expensive device-to-device memory copy. Accordingly, the *delete-table* and *delete-buffer* need to be adapted to a queue structure with a pointer pointing to the starting row. In Figure 6, there are two delete-update processes in the heap. They will restore the heap property at node 1 and node 4. The *delete-buffer* stores the items at node 1 and its children, and the items at node 4 and its children for processing, respectively.

Two similar data structures, *insert-table* and *insert-buffer*, are designed for insert-update processes. The *insert-table* has four columns representing the index of the target heap node, the offset of the next available slot in the target heap node, the level where the target heap node is located in the parallel heap, and the number of items to be inserted, respectively. The insert-update process uses these information to compute and select the target node in its insert path for the current processing. Before arriving to their target nodes, the to-be-inserted items are temporarily stored in the *insert-buffer*, which also provides working space for merging and sorting items. In

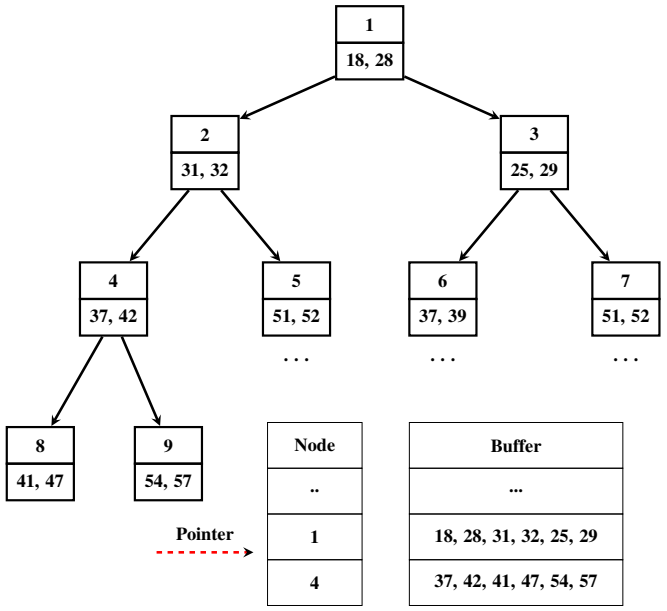

Fig. 6: Data structures for delete-update processes

Figure 7, there are two insert processes depicted, one at node 7 and another at node 1. A key 54 is to be inserted into node 15. A insert-update process therefore merges it with the items at node 7. Likewise, two keys 56 and 78, currently at node 1, are headed to node 16. The insert-update process merges them with the items at node 1.

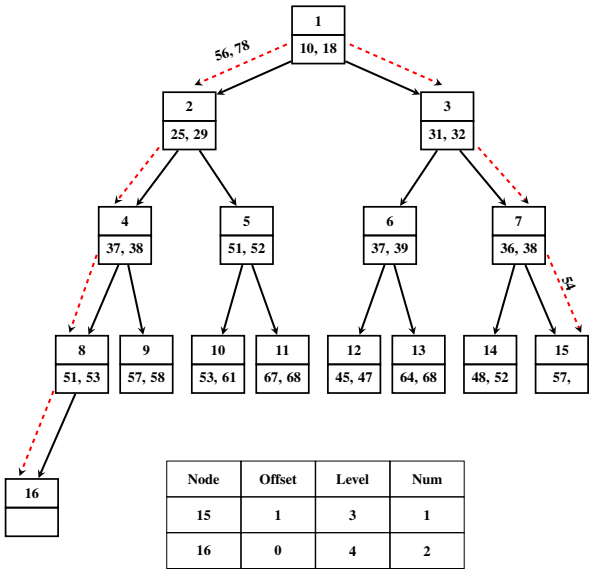

Fig. 7: Data structures for insert-update processes

When dealing with the situation in which there are not enough new items for deletions, a kernel function is invoked to search for the to-be-inserted items in the *insert-buffer*. $r-k$ to-be-inserted items are deleted from the *insert-buffer* and corresponding insert-update processes are cancelled if there are more than $r-k$ items in the *insert-buffer*. Otherwise, the items from both the *insert-buffer* and the end of the

heap are taken. This can be a tricky operation if this has to be concurrently executed while insert processes are ongoing. However, when this kernel is invoked by the CPU based controller, it does not have to contend with any other pipelined processes.

D. Sorting Implementation

In the original paper [3], only merge operations are needed by both delete-update and insert-update processes. In our implementation, however, we choose to sort items at heap nodes instead of merging them (for delete-update processes, two merge operations are required) considering inefficiency of the implementation of the merge operation on GPGPUs and the relatively less work of adapting the existing sorting routine to satisfy our need. The most efficient sorting routine on GPGPUs so far is the radix sort implementation [27], which is the starting point of our sorting implementation.

The main modification we made over the original sorting routine is to enable concurrent sorting and make full use of GPGPUs' computing resources. In the modified radix sort implementation, $r/512$ thread blocks are allocated for each sequence of items to be sorted, and the functionality is broken down into multiple kernel functions. The program is partitioned into passes, in each of which items within a sequence are sorted based on item's radix-4 digit. A synchronization is required between passes. Each pass consists of the following phases:

- Assign $r/512$ number of thread blocks for each sequence. Within a sequence, the items are partitioned into tiles. Each thread block loads its tile onto the shared memory and sorts the tile.
- Allocate a histogram table for each thread block. Within a sequence, compute the histogram and thus the global offset for each item.
- Within a sequence, write each item to its correct position in accord with its global offset.

E. The simulation of a priority queue application

The parallel heap system we present in the paper can support different kinds of priority queue based applications. A kernel function is used to simulate a typical application:

```
__global__ void app(int thinktime,int r){
    int tx=threadIdx.x;
    int bx=blockIdx.x;
    int i;
    if(bx*SIZE+tx<r){
        for(i=0;i<thinktime;i++){
          ......
        }
    }
}
```

The kernel function spawns r threads with each thread running a for-loop $thinktime$ times. By varying the value of $thinktime$, we can vary the grains of the application. Some trivial codes need to be in the for loop to prevent the for loop from being compiled away.

V. Experiments

A. Experimental Setup

Our current system for experiments is a Linux Infiniband 88-core cluster with heterogeneous nodes composed of multicores and GPGPUs. The multicores processors we use in the experiment are two Intel Xeon 5410 chip, each of which has Intel's two quad-core chips paired onto a multi chip module (MCM). Each core operates at 2.3 GHz and can fetch and decode four instructions per cycle. Each core has on-die, primary 32-kB instruction cache and 32-kB write-back data cache in each core and 12 MB (2 x 6MB) Level 2 cache. The graphics card for our experiments is Nvidia GeForce GTX 480. It is based on Fermi architecture, and equipped with 480 cores and 1.5 GB device memory.

We have three goals in our experiments. Firstly, we would like to evaluate the performance of our CUDA-based parallel heap system under the load of applications with different granularities. We would also like to see the impact of heap node size, which corresponds to the amount of available parallelism in the application, on the system's performance. Secondly, we want to compare the performance of the best sequential binary heap system with our parallel heap system. Lastly, we want to test the performance difference of the parallel heap implementation on CUDA architectures versus on multi-core architectures.

There are two important experiment modes. One is the hold model [28], in which the number of items to be inserted is exactly equal to the number of items to be deleted. The other model is the relaxed hold model, which randomly generates zero, one or two items for each item deleted. The relaxed hold model, which is more realistic for applications, is employed in our experiment.

The number of elements to be deleted or inserted, the think time, and the number of initial items in the heap are important factors in the experiment. We use r, n, t, and s to represent the size of a heap node, the number of initial items in the heap, the think time, and the total number of inserted or deleted items through the entire experiment over all delete-think-insert cycles respectively.

B. Performance Analysis

The performance of the parallel heap system (Parallel Priority Queue or PPQ) on CUDA is tested with varying think time and size of heap nodes. It is also compared with a sequential binary heap system (SeqHeap) and multicore implementation of PPQ.

Figure 8 shows the performance comparison of the parallel heap system implemented with sequential kernels and concurrent streaming kernels — the former corresponds to the application getting executed after parallel heap maintenance kernels, while the latter executes the two sets of kernels simultaneously in different streams. The red line is used to represent a parallel heap system implemented with sequential kernels. We can observe that the increase of its execution time is proportional to the increasing duration of the think time while other conditions remain unchanged. On the other hand, the blue line, standing for a parallel heap system implemented

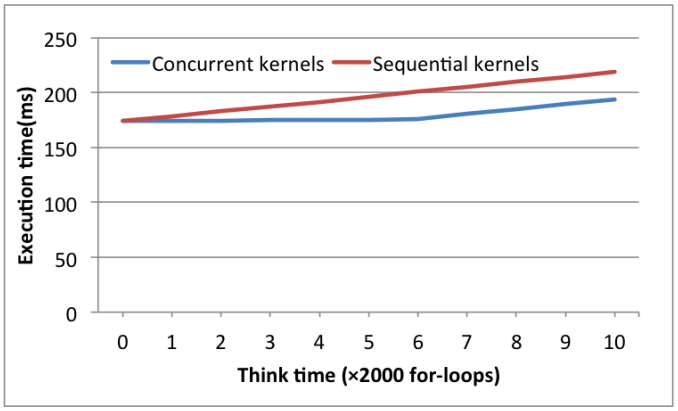

Fig. 8: Effect of concurrent kernels versus sequential kernels. ($s = 2^{14}$; $n = 2^{17}$; $r = 2^{8}$)

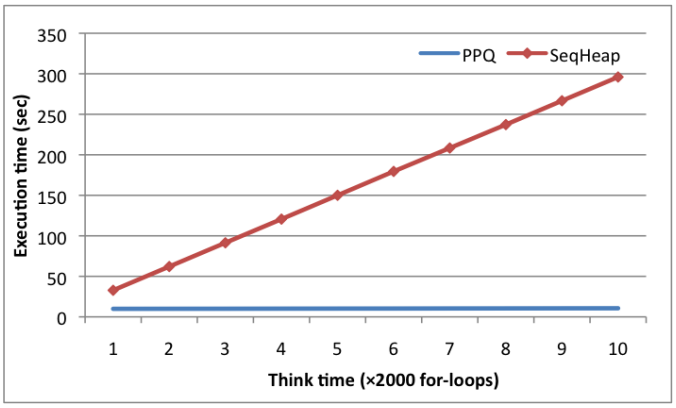

Fig. 10: Comparing of a parallel heap on CUDA and a sequential heap with different think time. ($s = 2^{22}$; $n = 2^{26}$; $r = 2^{17}$)

with concurrent kernels, is observed to stay stable when dealing with a fine-grained application ($t < 12000$). This is due to the fact that the execution of the application is tolerated by the maintenance work of the parallel heap. However, as the grain of the application becomes coarser, the system's performance gets dominated by the application's performance, and as a result the duration of the think time becomes significant for the system's performance (beyond $t > 12000$). Therefore, for all subsequent experiments, we report data employing concurrent kernels.

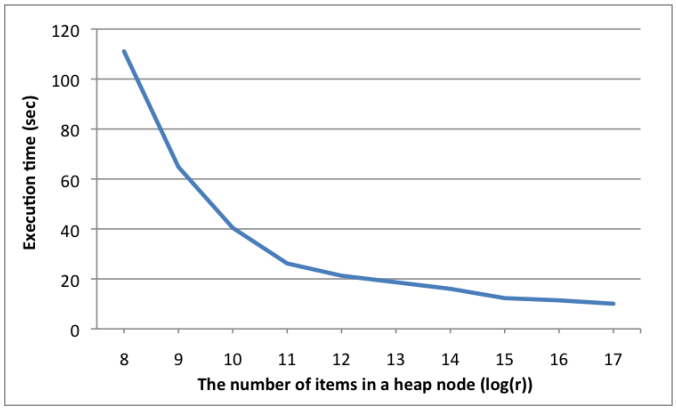

Fig. 9: Varying size of heap nodes. ($t = 6000$; $s = 2^{22}$; $n = 2^{26}$)

Figure 9 shows the performance of our parallel heap with different size of heap nodes. We can observe that the more items a heap node contains, the better performance the system achieves. On one hand, to insert or delete same amount of items, the number of the required delete-insert cycles can be reduced if a heap node can contain more items and thus more items can be updated in one update process. On the other hand, the latency toleration mechanism in GPGPUs required that enough threads exist in GPGPUs so that expensive global memory reads/writes can be tolerated. In other words, the wait for global memory reads/writes can keep processors in GPGPUs idle if not enough live threads can be scheduled to run, and in that case, launching more threads in GPGPUs can help utilize these idle processes without much additional overhead. Consequently, the increment of the number of items to be updated in one update operation does not cause proportional increase in this update operation's execution time and the overall system performance can thus enhance with fewer delete-insert cycles.

The wide heap node in the parallel heap can also be justified. With more computing resource, one can execute a very large complex heap-based application that can exhibit increasingly larger amount of concurrency. In our CUDA parallel heap system, we try to push more work to the GPGPUs to make full use of the GPGPUs. As the size of the heap node increases, the heap maintenance and the application kernels get more work to perform and hence the overall performance of the system improves.

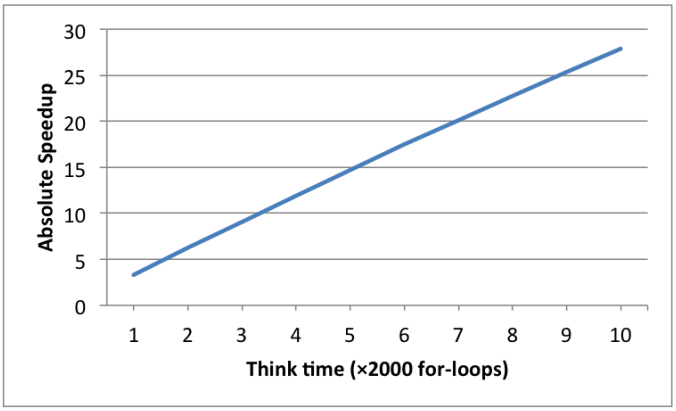

Fig. 11: Absolute speedups of parallel heap with different think time. ($s = 2^{22}$; $r = 2^{17}$; $n = 2^{26}$)

We also implemented a sequential binary heap system (SeqHeap) on much faster CPU multi-cores for comparison. This system is more efficient than the conventional binary heap system as it can combine a pair of consecutive insert and delete operations into one insert-delete operation. The red line in Figure 10 shows that the duration of the think time is significant to the performance of the sequential heap. The blue line represents the PPQ performance under varying loads of different fine-to-medium grained applications. As we can see,

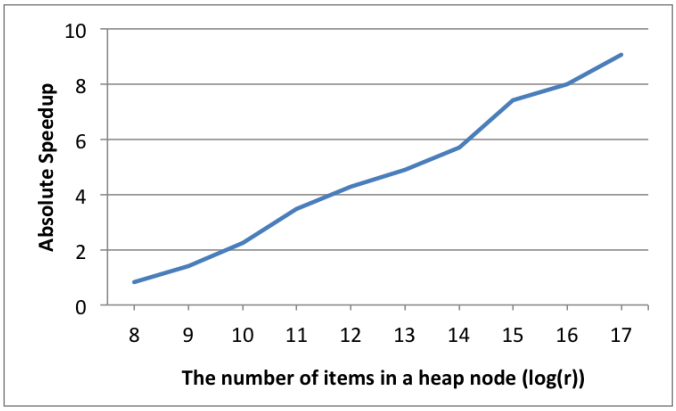

Fig. 12: Absolute speedups with varying size of heap nodes. ($t = 6000$; $s = 2^{22}$; $n = 2^{26}$)

the whole system's performance remains stable with different compute grains. As in the discussion for Figure 8, due to the concurrent kernel techniques we employ, the execution of fine-grained applications is well tolerated by the parallel heap's maintenance work. The system's performance is not sensitive to the duration of the think time for such compute loads. Moreover, as shown in Figure 10, with large enough size of the heap nodes and more maintenance work, even the execution of medium-grained applications can also be tolerated.

Figure 11 shows the duration of the think time is significant to the absolute speedup. As discussed above, the execution time of SeqHeap increases with the increasing duration of the think time while the execution time of PPQ does not. Hence more speedup can be achieved as the duration of the think time increases. Figure 12 indicates that the absolute speedup we can achieve rises with the increasing size of the parallel heap node. This is reasonable because wider heap node can enhance the performance of PPQ while SeqHeap nodes can contain only one item.

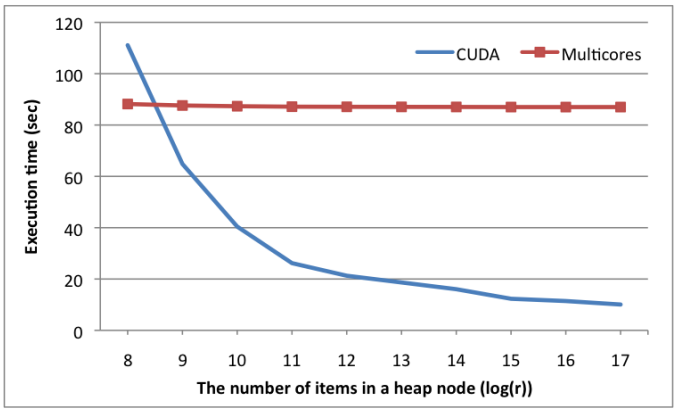

Fig. 13: The performance comparison with multicore-based parallel heap implementation with varying node sizes. ($t = 6000$; $s = 2^{22}$; $n = 2^{26}$)

Finally, the performance difference of GPGPUs based and multicores based implementations over varying node sizes is displayed in Figure 13. The multicore pthread-based implementation of a parallel heap [29] is based on bus-based shared memory implementation in [30], and can achieve better performance for medium grained applications. However, both of these implementations have some drawbacks. Due to the hardware restriction, the number of available processors is limited. Therefore, only one processor is assigned to deal with the delete or insert operation for one level of the parallel heap even though there is a lot of parallelism that we can exploit. That partly explains why the CUDA implementation has a better performance than the multi-core implementation, especially when the size of the heap node is large.

VI. Conclusion

In this paper, we discuss the characteristic of many-core architectures and design a priority queue implementation that can take that into consideration. We described the system design and implementation of an efficiently implemented parallel heap. Experiments have shown that the larger a heap node is (the more concurrency the priority-queue-based application exhibits), the more efficient the system becomes. A good speedup can be achieved with large heap nodes. Experimental result also implies that the overall performance of the system is oblivious to the finer compute grain of different applications. In particular, fine-grained applications, which typically face serious bottleneck accessing a shared priority queue are well parallelized for the operational range of parameters studied.

References

[1] R. Fujimoto, "Parallel discrete event simulation," in *Proceedings of the 21st conference on Winter simulation*. ACM, 1989, pp. 19–28.
[2] S. Prasad, "Efficient parallel algorithms and data structures for discrete-event simulation," *PhD Dissertation*, 1990.
[3] N. Deo and S. Prasad, "Parallel heap: An optimal parallel priority queue," *The Journal of Supercomputing*, vol. 6, no. 1, pp. 87–98, 1992.
[4] T. Cormen, *Introduction to algorithms*. The MIT press, 2001.
[5] J. Vuillemin, "A data structure for manipulating priority queues," *Communications of the ACM*, vol. 21, no. 4, pp. 309–315, 1978.
[6] M. Fredman and R. Tarjan, "Fibonacci heaps and their uses in improved network optimization algorithms," *Journal of the ACM (JACM)*, vol. 34, no. 3, pp. 596–615, 1987.
[7] G. Brodal, "Priority queues on parallel machines," *Algorithm TheorySWAT'96*, pp. 416–427, 1996.
[8] R. Nageshwara and V. Kumar, "Concurrent access of priority queues," *Computers, IEEE Transactions on*, vol. 37, no. 12, pp. 1657–1665, 1988.
[9] M. Kunjir and A. Manthramurthy, "Using graphics processing in spatial indexing algorithms," *Research report, Indian Institute of Science, Database Systems Lab*, 2009.
[10] K. Zhou, Q. Hou, R. Wang, and B. Guo, "Real-time kd-tree construction on graphics hardware," in *ACM Transactions on Graphics (TOG)*, vol. 27, no. 5. ACM, 2008, p. 126.
[11] Q. Hou, X. Sun, K. Zhou, C. Lauterbach, and D. Manocha, "Memory-scalable gpu spatial hierarchy construction," *IEEE Transactions on Visualization and Computer Graphics*, 2010.
[12] S. Popov, J. Günther, H. Seidel, and P. Slusallek, "Stackless kd-tree traversal for high performance gpu ray tracing," in *Computer Graphics Forum*, vol. 26, no. 3. Wiley Online Library, 2007, pp. 415–424.
[13] D. Benson and J. Davis, "Octree textures," in *ACM Transactions on Graphics (TOG)*, vol. 21, no. 3. ACM, 2002, pp. 785–790.
[14] G. Ziegler, R. Dimitrov, C. Theobalt, and H. Seidel, "Real-time quadtree analysis using HistoPyramids," *Real-Time Image Processing 2007*, vol. 6496, 2007.
[15] P. Ajmera, R. Goradia, S. Chandran, and S. Aluru, "Fast, Parallel, GPU-based Space Filling Curves and Octrees," *I3D2008*.
[16] R. Castro, T. Lewiner, H. Lopes, G. Tavares, and A. Bordignon, "Statistical optimization of octree searches," in *Computer Graphics Forum*, vol. 27, no. 6. Wiley Online Library, 2008, pp. 1557–1566.

[17] X. Sun, K. Zhou, E. Stollnitz, J. Shi, and B. Guo, "Interactive relighting of dynamic refractive objects," in *ACM SIGGRAPH 2008 papers*. ACM, 2008, pp. 1–9.

[18] S. Lefebvre, S. Hornus, and F. Neyret, "Octree textures on the GPU," *GPU gems*, vol. 2, pp. 595–613, 2005.

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

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

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

[22] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha, "Fast BVH construction on GPUs," in *Computer Graphics Forum*, vol. 28, no. 2. Wiley Online Library, 2009, pp. 375–384.

[23] S. Lefebvre and H. Hoppe, "Perfect spatial hashing," in *ACM SIGGRAPH 2006 Papers*. ACM, 2006, pp. 579–588.

[24] D. Alcantara, A. Sharf, F. Abbasinejad, S. Sengupta, M. Mitzenmacher, J. Owens, and N. Amenta, "Real-time parallel hashing on the GPU," *ACM Transactions on Graphics (TOG)*, vol. 28, no. 5, pp. 1–9, 2009.

[25] R. Pagh and F. Rodler, "Cuckoo hashing," *AlgorithmsESA 2001*, pp. 121–133, 2001.

[26] J. Yang, J. Hensley, H. Gr
"un, and N. Thibieroz, "Real-Time Concurrent Linked List Construction on the GPU," in *Computer Graphics Forum*, vol. 29, no. 4. Wiley Online Library, 2010, pp. 1297–1304.

[27] N. Satish, M. Harris, and M. Garland, "Designing efficient sorting algorithms for manycore gpus," in *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*. IEEE, 2009, pp. 1–10.

[28] D. Jones, "An empirical comparison of priority-queue and event-set implementations," *Communications of the ACM*, vol. 29, no. 4, pp. 300–311, 1986.

[29] D. Agarwal, "Memory hierarchy aware parallel priority based data structures," in *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, may 2011, pp. 2042 –2044.

[30] S. Prasad and S. Sawant, "Parallel heap: A practical priority queue for fine-to-medium-grained applications on small multiprocessors," in *Parallel and Distributed Processing, 1995. Proceedings. Seventh IEEE Symposium on*. IEEE, 1995, pp. 328–335.