Performance Evaluation of Concurrent Lock-free Data Structures on GPUs

Prabhakar Misra and Mainak Chaudhuri
Department of Computer Science and Engineering
Indian Institute of Technology
Kanpur, INDIA
{*prabhu,mainakc*}*@cse.iitk.ac.in*

***Abstract*—Graphics processing units (GPUs) have emerged as a strong candidate for high-performance computing. While regular data-parallel computations with little or no synchronization are easy to map on the GPU architectures, it is a challenge to scale up computations on dynamically changing pointer-linked data structures. The traditional lock-based implementations are known to offer poor scalability due to high lock contention in the presence of thousands of active threads, which is common in GPU architectures. In this paper, we present a performance evaluation of concurrent lock-free implementations of four popular data structures on GPUs. We implement a set using lock-free linked list, hash table, skip list, and priority queue. On the first three data structures, we evaluate the performance of different mixes of addition, deletion, and search operations. The priority queue is designed to support retrieval and deletion of the minimum element and addition operations to the set. We evaluate the performance of these lock-free data structures on a Tesla C2070 Fermi GPU and compare it with the performance of multi-threaded lock-free implementations for CPU running on a 24-core Intel Xeon server. The linked list, hash table, skip list, and priority queue implementations achieve speedup of up to 7.4, 11.3, 30.7, and 30.8, respectively on the GPU compared to the Xeon server.**

***Keywords*-linked list; hash table; skip list; priority queue; concurrent; lock-free; GPU; CUDA;**

I. Introduction

Graphics processing units (GPUs) have become one of the preferred vehicles for high-performance general purpose computing. This computing paradigm is commonly known as general purpose computing on GPU (GPGPU) or GPU computing. Regular data-parallel computations with little or no synchronization have been efficiently mapped on the GPUs. However, a large number of general purpose ordinary programs have irregular accesses to pointer-linked data structures that change dynamically through addition and deletion of items. Achieving scalable performance on such data structures requires highly concurrent implementations. In small to medium-scale parallel machines with tens of active thread contexts, it may be acceptable to have some amount of lock-based synchronization. However, this would introduce prohibitive performance overhead in GPUs where the number of active threads can easily extend to thousands. Possibility of high lock contention at this scale rules out lock-based implementations.

In this paper, we present an evaluation of lock-free concurrent implementation of a few important data structures on GPUs. To the best of our knowledge, this is the first detailed evaluation of a number of lock-free data structures on GPUs. We present four implementations of a set with the help of linked list, hash table, skip list, and priority queue. The first three data structures support concurrent lock-free addition, deletion, and search operations on the set, while the concurrent priority queue offers lock-free retrieval and deletion of the minimum element and addition operations. Our choice of data structures is governed by their importance in general purpose computing. Linked lists form the building block for many important data structures, such as, graphs. Hash tables are often used to reduce average case search time. We present a lock-free design of a closed-address hash table, which builds upon our lock-free linked list design. Skip lists offer expected logarithmic search time and our lock-free priority queue builds upon a lock-free implementation of the skip list. All our implementations use the CUDA (Compute Unified Device Architecture) C++ programming model and rely on the CUDA atomic primitives such as atomic compare-and-swap (CAS), atomic increment, etc..

We measure the performance of these data structures by executing a mix of the concurrent operations supported by each of the data structures. Our evaluation is carried out on a Tesla C2070 Fermi GPU as well as a 24-core Intel Xeon server. The GPU implementations of the lock-free linked list, hash table, skip list, and priority queue achieve speedup of up to 7.4, 11.3, 30.7, and 30.8, respectively compared to the lock-free multi-threaded CPU execution.

The concurrent implementations of the four data structures chosen by us have been studied in great detail in the context of CPUs and we review some of these contributions in Section I-A. Section II summarizes the CUDA programming environment. Section III presents the lock-free implementations of the four data structures on GPU. We discuss the evaluation methodology and the performance results in Sections IV and V.

A. Related Work

In this paper, we have implemented four lock-free data structures on CUDA-enabled GPUs. While a significant amount of research has been done on lock-free data structures in the context of traditional CPUs, there is very little known about the performance of these data structures on the GPUs. Herlihy and Shavit discuss concurrent implementations of several data structures on shared memory multiprocessors using JAVA [10]. We summarize relevant portions of this literature on CPU-based implementations and discuss a few studies relevant to GPU implementations.

Lock-free linked list implementation using atomic CAS operations is proposed by Valois [29]. This implementation supports linearizable operations [12] i.e., each operation appears to take place atomically at some point (the linearization point) during its execution. Valois also proposes a reference count-based solution to the ABA problem related to memory management of data structures operated on by atomic CAS. Subsequently, Harris [9]

presents improved algorithms for lock-free linked lists supporting linearizable addition and deletion operations. Michael further improves this implementation to be compatible with lock-free memory management and shows how to use his lock-free linked list implementation to construct a lock-free hash table [19]. This lock-free implementation of linked list has been known as the Harris-Michael algorithm. We will detail a variation of this algorithm in Section III-A.

Hsu and Yang present the design of a concurrent extensible closed-address hash table with minimal locking with the help of optimistic concurrency control protocols via dynamic verification of consistent view of the hash buckets (or directories) [13]. Ellis [6] details the design of lock-based linear hashfiles in the context of databases with the help of the lock-coupling protocol [3] where an algorithm locks the next component of a data structure before releasing the lock on the current component. Kumar's construction of concurrent extensible hash tables further lowers the locking overhead [16]. Greenwald implements a non-blocking resizable and linearizable closed-address hash table using double-word CAS (DCAS) operations that can atomically operate on two independent memory locations [8]. Shalev and Shavit present a lock-free resizable and linearizable hash table employing CAS operations via a technique called recursive split-ordering [24]. The central observation in this design is that all items of the hash table can be maintained in a single lock-free linked list and new buckets can be introduced via references into this list. As a result, a resizing operation does not require moving an item from an old bucket to a new bucket atomically.

Sequential as well as concurrent skip lists are introduced by Pugh as a randomized alternative to the deterministic balanced search trees [21], [22]. Skip lists enjoy simpler operations to maintain an expected logarithmic search time compared to the complex balancing operations in a search tree. This simplicity of skip lists becomes especially important in efficient concurrent implementations [7]. While Pugh's concurrent implementation is lock-based, Fraser presents a CAS-based lock-free skip list by treating each level of a skip list as a lock-free linked list [7]. Herlihy, Lev, and Shavit present a lock-free skip list [10] that is partially based on Fraser's construction. We will discuss the construction due to Herlihy, Lev, and Shavit in Section III-C. Cederman, Tsigas, and Chaudhry present a brief evaluation of a lock-free skip list on GPUs and show that it performs significantly better than a software transactional memory-based implementation [5]. Their lock-free construction is based on the implementation due to Sundell and Tsigas [28].

Rao and Kumar present lock-based designs of concurrent priority queues [23]. A fine-grain lock-based priority queue design is discussed by Hunt et al. [14]. Shavit and Zemach present lock-free bounded priority queues based on arrays and trees [25]. Lotan and Shavit [18] discuss lock-free quiescently consistent [1], [25] as well as linearizable priority queues based on lock-free skip lists. We will present the details of the quiescently consistent implementation in Section III-D. Sundell and Tsigas discuss another lock-free implementation of priority queues based on skip lists [27]. Finally, there have been efforts among the hardware designers to pipeline the priority queue operations for efficient cache replacement policies [2] and smart scheduling in high-speed networks [15].

Xiao and Feng present lock-based and lock-free implementations of global barriers to synchronize independent thread blocks on a GPU [30]. Stuart and Owens explore implementations of barriers, mutexes, and semaphores on GPUs [26].

Our evaluation of the lock-free data structures considered in this paper is the first attempt to gain a detailed understanding of the performance of these data structures on GPUs. We note that a recent study has explored the performance of lock-based and lock-free queues on GPUs [4].

II. Background: Programming with CUDA

Our implementations use the CUDA C++ programming model. A detailed introduction to CUDA can be found in [20]. CUDA is a parallel computing architecture consisting of a parallel programming model and a parallel thread execution (PTX) instruction set architecture that can leverage the parallelism available in the Nvidia GPUs. CUDA offers a software development environment that extends a traditional high-level language such as C, C++, Fortran, etc. to help programmers express the parallelism in the applications. The CUDA API provides libraries that can be called from the high-level language code. In the following, we discuss some of the features of this API.

The functions that are launched on the GPU for concurrent execution are called kernel functions. The declaration of kernel functions must be preceded by `__global__`. The CUDA keywords `__device__` and `__host__` are used with function declarations and indicate whether a function will be called from the GPU code or the CPU code, respectively. The default is the latter when no keyword is specified, but both must be specified if a function must be compiled for the GPU as well as the CPU. Before a kernel is launched, a thread grid configuration is specified and the threads in this configuration will execute the launched kernel on the GPU. The grid is logically arranged in a one, two, or three dimensional array of thread blocks. A thread block is further organized into a one, two, or three dimensional array of threads. A thread block may contain up to 1024 threads on Fermi GPUs that we use. A typical grid would have several tens of thousand of threads. Each thread within a thread block and each block within a grid receives a unique identifier through thread-private in-built variables. These are initialized depending on the position of a thread within a grid and can be accessed by a thread from the kernel function. Each thread executes an instance of the kernel function. Usually each instance operates on a different segment of input decided by the id of the thread executing that instance.

The CUDA thread model is tied closely to the array of streaming multiprocessors (SMs) that the GPU hardware consists of. The Fermi GPU that we use in this study has 14 SMs and each SM has 32 CUDA cores, 16 load/store units, and four special function units for executing the transcendental functions. Each of the 448 CUDA cores on the GPU is equipped with fully pipelined integer and floating-point units, but does not have any support for out-of-order instruction issue, branch prediction, or speculative execution. Each SM has a large register file and a configurable shared memory and L1 cache.

Individual thread blocks are scheduled on different SMs and all the threads of a thread block execute concurrently on one SM. Multiple thread blocks can execute concurrently on one SM obeying the scheduling constraints. When a thread block terminates on an SM, a new thread block is scheduled on the SM. Each SM is designed to manage and schedule hundreds of concurrent threads through a single instruction multiple thread (SIMT) architecture. The threads in a block can communicate through the shared memory and synchronize with barriers.

The SM schedules and executes the threads in groups of 32 parallel threads called warps. The threads in a warp start execution at the same program counter, but they can branch and diverge independently. If threads of a warp diverge due to a conditional branch, the warp sequentially executes each taken branch path by disabling the threads that are not on that path. When all such paths are executed, the threads in a warp converge back to a common path. Frequent control flow divergences can hurt performance severely. When an SM receives one or more thread blocks, it partitions them into warps and executes each warp independently. A warp scheduler switches out the warp blocked on a long-latency event (such as global memory access) and schedules a ready-to-execute warp. The context of a warp remains in the SM throughout the life time of a warp. As a result, warp scheduling is very fast on GPUs. The number of thread blocks and warps that can reside on an SM depends on the register and shared memory need of each concurrent instance of the executing kernel function. In Fermi GPUs, an SM can accommodate up to 1536 threads.

Different thread blocks cannot communicate through the per-SM shared memory and must use the global memory for this purpose. While there are different flavors of memory fence instructions to maintain memory consistency within a thread block and across thread blocks, the only way to implement synchronization between arbitrary threads in a grid is through the atomic operations executing in the relatively slow global memory. Although the data in global memory can be cached in the globally shared 768 KB L2 cache of the Fermi architecture, the access latency is still much higher than the L1 cache or the per-SM shared memory. Since all the states of a data structure that we implement must be shared across all the threads to support concurrent operations on arbitrary elements in the data structure, all our data structures are kept in the global memory. The global memory to be used by a kernel on the GPU can be allocated from the CPU before launching the kernel with the help of CUDA APIs. The CUDA memory allocation function (`cudaMalloc`) returns a pointer to the global memory allocated on the GPU and this pointer must be passed to the kernel so that it can access the allocated memory. CUDA APIs further offer memory copy functions (`cudaMemcpy`) to copy contents between the CPU memory and the GPU global memory. These calls are useful in setting up kernel inputs in the GPU memory and bringing back kernel outputs from GPU memory to CPU memory.

We close this section with a brief discussion on the atomic operations that we use in this study. We use two atomic operations offered by CUDA, namely, atomicCAS and atomicInc, to implement our lock-free data structures. The atomicCAS operation takes three arguments, namely, an address A, an expected value V_{exp}, and a new value V_{new}. It reads the value V_{old} at address A. If V_{old} equals V_{exp}, it stores V_{new} at address A; otherwise it leaves the contents of A unchanged. It always returns V_{old}. By comparing the return value with V_{exp}, one can check if the execution of atomicCAS has successfully stored V_{new}. An atomicCAS operation of a thread T_1 to address A may fail if some other thread T_2 updates the contents of A with a value different from V_{exp} of T_1. We note that single-word atomicCAS has an infinite consensus number [11], thereby offering the most powerful synchronization primitive for implementing lock-free and wait-free operations.

The atomicInc operation takes two arguments, namely, an address A and a value V_{max}. It reads the value V_{old} at address A. If V_{old} is greater than or equal to V_{max}, it resets the contents of A to zero; otherwise it stores $V_{old} + 1$ at address A. It always returns V_{old}. The atomicInc operation atomically increments a memory location with wrap around at a chosen maximum value.

III. Lock-free Data Structures

This section presents the design of the lock-free data structures. We use quiescent consistency and linearizability as the correctness criteria of our concurrent data structures. A data structure is said to follow quiescent consistency if all operations (or function calls) on the data structure appear to happen in some sequential order and two groups of operations separated by a period of quiescence appear to take effect in their real-time order. For example, consider two concurrent enqueue operations to a FIFO queue adding elements x and y to the queue. After these two operations complete, a third enqueue operation adds z to the queue. If the implementation of the FIFO queue follows quiescent consistency, z will appear after both x and y, although the relative order of x and y is arbitrary. A data structure is linearizable if its implementation supports linearizable operations i.e., each operation appears to take place atomically at some point (the linearization point) during its execution.

Before we move on to the discussion of the data structures, we formally define two terms that we will use: lock-freedom and wait-freedom. An implementation of an operation or a function is lock-free if infinitely often some call to the function among a number of concurrent calls finishes in a finite number of steps. An implementation of an operation or a function is wait-free if every call to the function finishes in a finite number of steps [17]. Every wait-free implementation is also lock-free. Lock-free and wait-free implementations guarantee forward progress without depending on warp scheduling and other thread scheduling policies.

A. Linked List

The linked list implements a set supporting three operations, namely, add, $delete$, and $search$. The elements in the linked list are kept sorted in the ascending order starting from the head. The $add(x)$ call adds the element x to the set, if it is already not in the set and returns one. If x is already in the set, it does not insert x and returns zero. The $delete(x)$ call removes the element x from the set, if it is in the set and returns one. If x is not in the set, it returns zero. The $search(x)$ call returns one, if x is in the set and zero otherwise. The add and $delete$ implementations are lock-free, while the $search$ implementation is wait-free. All the three operations are linearizable. Our implementation follows the construction presented in [10], which is a variation of the Harris-Michael construction. In the following, we assume that each linked list node has two fields, namely, `element` holding the value and `next` holding the address of the next node.

The $add(x)$ and $delete(x)$ functions first walk through the sorted list trying to locate x. Let the address of the node holding the minimum element k such that $k \geq x$ be $CURR(x)$ and the address of the node before $CURR(x)$ be $PRED(x)$ i.e., $PRED(x) \rightarrow next$ is equal to $CURR(x)$. If k is not equal to x, the $delete$ operation returns zero. However, in this case, the add operation creates a new linked list node with element value x and makes its `next` field hold $CURR(x)$. Finally, the add operation executes an atomicCAS on the location $\&(PRED(x) \rightarrow next)$ to make $PRED(x) \rightarrow next$ point to the newly created node. If the atomicCAS is successful, the add operation returns one; otherwise the entire add operation starts over again by trying to locate x.

If k is equal to x, the add operation returns zero. In this case, the $delete$ operation needs to remove the node containing x from the

list. This node is pointed to by $CURR(x)$. It is important to note that changing the value of $PRED(x) \rightarrow next$ from $CURR(x)$ to $CURR(x) \rightarrow next$ through an atomicCAS operation does not work because concurrent deletion of two consecutive nodes (e.g., pointed to by $CURR(x)$ and $CURR(x) \rightarrow next$) using this protocol may not, in fact, delete the second node. We follow the well-known pointer marking protocol to implement lock-free $delete$. Each linked list node has a single-bit `mark` field, which is normally reset. When a node is deleted, this bit is set in that node with the help of atomicCAS signifying logical deletion of the node. The $delete(x)$ operation, in addition to marking the node being deleted, tries to delete the node physically by an invocation of atomicCAS. If the atomicCAS fails, it just leaves the node logically deleted and returns one.

The responsibility of physically removing a marked node falls on subsequent add and $delete$ operations. Recall that these operations first walk the linked list looking for the addition/deletion site in the list. Any marked node encountered during this walk are physically removed. This is done with the help of an atomicCAS operation by changing the `next` field of the node previous to the marked node. Let the previous node be $PREV$. There is a danger that before the change in the `next` field of $PREV$ is affected, $PREV$ may get marked and in such a situation proceeding with the atomicCAS on $\&(PREV \rightarrow next)$ may lead to an inconsistent state of the list. Therefore, it is important that the `mark` and the `next` fields be changed together by the atomicCAS operation and if any one of these is modified by some other concurrent operation before the atomicCAS is completed, the atomicCAS will fail. This is achieved by stealing the least significant bit of the `next` field to store the mark bit. Since the least significant two bits of a word-aligned 32-bit CUDA pointer would be zero, whenever the `next` field is used as a pointer, the least significant bit is zeroed. If the atomicCAS operation fails during the physical removal of a marked node, the entire list walk starts over from the head.

The $search(x)$ operation walks the list from the head looking for x. If x is found and its node is not marked, the function returns one; otherwise it returns zero. Since this operation does not involve any atomicCAS calls, it is guaranteed to finish in a finite number of steps, and hence, is wait-free.

B. Hash Table

We implement a lock-free closed-address hash table by leveraging our linked list construction. The hash table implements a set and supports the same three functions as the linked list. The hash table is implemented as a single linked list and an array of pointers into the list stores the starting points of the buckets. Our hash table supports a constant number of buckets and this number is fixed at the time of CUDA kernel launch. The linked list node that starts a bucket stores a special key value so that this can be used to indicate the end of the previous bucket as well as the beginning of a new bucket. Specifically, the head node of bucket i stores the key (0x80000000 OR i), where "OR" is the bitwise OR operation. The add, $delete$, and $search$ operations for bucket i begin at this node of the linked list. The bucket ends with the key value (0x80000000 OR $(i + 1)$). This design limits the actual key values to be of 31 bits and the number of buckets to 2^{31} (in a 64-bit implementation, the key range and the number of buckets can be expanded by choosing a different key pattern for the head nodes). The segment of the linked list within each bucket is sorted as in the original linked list implementation. The tail node of the complete linked list stores the special key 0xffffffff. This node is necessary to indicate the end of the last bucket. The head nodes and the tail node can never be deleted.

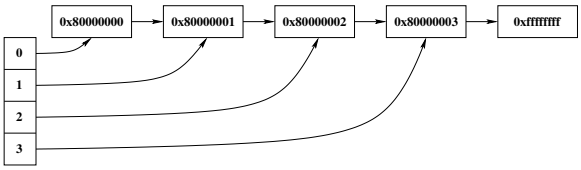

Figure 1. Hash table at the time of initialization.

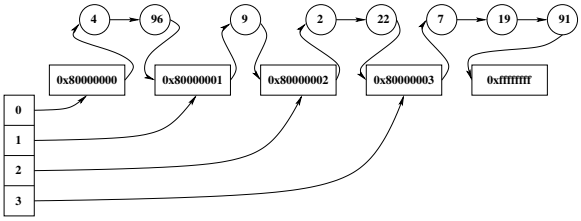

Figure 2. Hash table with a few keys inserted.

Figure 1 shows an example hash table with four buckets just after initialization. Figure 2 shows the same hash table after a few keys are inserted. The hash function used is key value modulo number of buckets. On an identical mix of add, $delete$, and $search$ operations, the set implemented with a hash table is expected to offer better throughput compared to a single linked list even though the hash table is implemented using a single linked list. This is because in the hash table, multiple non-interfering operations can be in flight in the independent segments of the linked list belonging to different buckets.

C. Skip List

Our lock-free skip list implementation follows the linearizable construction presented in [10]. We implement a set supporting the usual add, $delete$, and $search$ operations. A skip list can be seen as a hierarchy of linked lists as shown in Figure 3. The keys present in level $n + 1$ form a subset of the keys present in level n. For example, in Figure 3, at level zero, all the keys are present and the keys are linked up through an ordinary sorted linked list. At level one, however, only the keys 3, 7, and 16 are present and they are linked up through a separate linked list at level one. When a new key is inserted in the skip list, a random level r is generated with expected value $\frac{1}{1-p}$ for some predefined $p \in [0, 1]$, where p is the probability of finding a key at level $n + 1$, given that the key is present at level n. This value r serves as the maximum level up to which the new key can be present. If this value is bigger than a predefined maximum level, the new key is inserted in all the levels. One such function for generating the maximum level for a new key is presented in [21]. Assuming that the maximum number of levels is N, levels zero to $N - 2$ are used for maintaining the keys in the skip list. Level $N - 1$ is reserved for linking up the head and the tail nodes of the list. The head node holds a key (m in Figure 3) smaller than all the keys in the allowable range, while the tail node holds a key (M in Figure 3) bigger than all the keys in the allowable range.

The linked lists at different levels in a skip list should be seen as shortcuts for reaching a particular key skipping over several

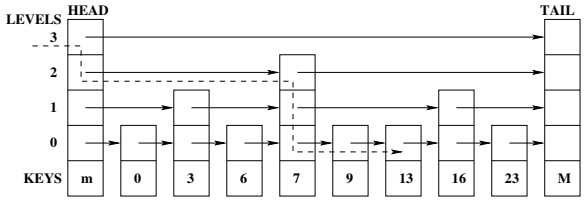

Figure 3. A skip list with four levels. Search path for key 13 is shown dotted.

elements at level zero. For example, Figure 3 shows the search path for key 13. The search always starts at the highest level of the head node. It compares the search key with the key of the next node at that level. If the search key is smaller, the operation climbs down the levels until it reaches a level where the key in the next node is smaller than or equal to the search key. The search operation moves on to the next node at this level and the process is repeated until the key is found or it becomes evident that the key is not present in the list.

An $add(x)$ operation for a new key x locates the site of insertion by traversing the skip list, generates a random maximum level r for this key, and inserts a new node in the linked lists at levels zero to r. A $delete(x)$ operation for a key x already present in the skip list locates the key and removes the node from all the linked lists it is present in. Clearly, a lock-free skip list can leverage our lock-free linked list implementation. The only difference is that the physical removal of a marked node may require physical removals in multiple linked lists. Also, an add operation may have to insert into multiple linked lists. It is not possible to make multiple physical removals or insertions atomic with single-word atomicCAS. Also, depending on the path followed by an add or $delete$ operation, some middle level of a marked node may be removed physically leaving the other levels unchanged, thereby violating the subset property of two consecutive levels. Nonetheless, the implementation guarantees linearizable operations by making sure that an add operation links a new node at level zero first and moves bottom-up; a $delete$ operation marks the node to be deleted starting from its maximum level and moves top-down. A key is defined to be present in the skip list until it is found unmarked at level zero.

Due to the complex code structure of a skip list and the potentially multiple atomicCAS operations (executing in the relatively slow global memory) needed to insert a key or physically remove a key, we expect the hash table to offer better throughput compared to a skip list on identical mix of operations. Particularly worrisome are the control flow divergences introduced by the complex structure of the skip list code. However, we expect the skip list to perform better than the linked list on identical mix of operations because the skip list guarantees an expected logarithmic search time.

D. Priority Queue

We design the lock-free priority queue by leveraging the lock-free skip list, as outlined in [10]. The keys are essentially the priorities of the elements. We implement a set using the priority queue and we support two operations on the set, namely, add and $deleteMin$. To implement the $deleteMin$ operation, we internally support a $delete$ operation as well.

The $deleteMin$ operation walks the level zero list starting from the head looking for the first unmarked node. Once it finds this node, it tries to mark it using atomicCAS. If the atomicCAS fails, it just continues walking the level zero list. Once the $deleteMin$ operation successfully marks an unmarked node, it notes its key x and calls $delete(x)$ on the skip list. A large number of concurrent $deleteMin$ operations can cause heavy contention near the head of the skip list leading to a number of atomicCAS failures. The add operation works identically as in the lock-free skip list and takes an expected amount of time that is logarithmic in the number of keys.

This lock-free construction of the priority queue is quiescently consistent and not linearizable. A $deleteMin$ operation executing concurrently with but completing after two sequential add operations, namely, $add(x)$ and $add(y)$ with $x < y$ may return y because it may have passed the insertion site of x before x is inserted. Therefore, these operations are not linearizable.

E. Memory Management

All the four data structures build on the lock-free linked list. As a result, all of these data structures share a generic node structure. We pre-allocate a large number of such nodes in word-aligned manner in the GPU memory from the CPU before the GPU kernel is launched. The pointers to these pre-allocated nodes are stored in an array P_array in the GPU memory. This setup is carried out with the help of `cudaMalloc` and `cudaMemcpy` calls. We also maintain an index in the GPU global memory and it is initialized to zero. When an add operation needs to create and insert a new node, it executes an atomicInc on the index and uses the return value v to index into P_array. The node pointed to by the pointer stored in $P\_array[v]$ is used as the new node for insertion. In this study, we pre-allocate enough number of nodes so that we never have to invoke dynamic memory allocation from the GPU (which is possible in the Fermi architecture). This makes sure that our performance evaluation can focus on the raw throughput achievable from the lock-free data structures without any perturbation from memory allocation overhead.

In this study, we do not reuse any of the deleted nodes because that would require a solution to avoid the ABA problem. We leave this to the future research and focus on evaluating the primary benefits of the lock-free data structures in this study.

IV. Evaluation Methodology

We implement the four lock-free data structures on a Tesla C2070 Fermi GPU as well as a 24-core Intel Xeon server. The core and memory clock frequencies of the GPU are 1.15 GHz and 1.49 GHz, respectively. We configure the GPU to have 48 KB shared memory and 16 KB L1 cache per thread block. It also has a globally shared 768 KB L2 cache and a 384-bit wide memory bus. The Xeon server is a quad-processor SMP with each processor being hex-core (Intel X7460 CPU) running at 2.66 GHz. Each of the four processors has a 16 MB L3 cache shared by the six cores of that processor. The CPU implementation uses POSIX threads and the x86 cmpxchg instruction to realize the atomicCAS primitive.

The performance of a lock-free data structure for a fixed number of threads may depend on the range of keys, the mix of operations done on the keys, and the total number of operations. We evaluate each data structure for a number of different mix of operations. For linked list, hash table, and skip list, we represent each different mix as a three-tuple $[x, y, z]$, where the operation stream has $x\%$ add, $y\%$ $delete$, and $z\%$ $search$. For priority queue, we represent each different operation mix as a pair $[x, y]$, where the operation

stream has $x\%$ add and $y\%$ $deleteMin$. We evaluate the data structures on each operation mix for four different integer key ranges, namely, $[0, 100)$, $[0, 1000)$, $[0, 10000)$, and $[0, 100000)$. Further, for each operation mix and each key range, we vary the total number of operations from 10000 to 100000 in steps of 10000.

The input to a CUDA kernel for a particular data structure and a particular key range consists of a string of operations. This string is generated as follows. The operations are generated from the supported set of operations for the data structure under evaluation such that the required mix of operations is achieved. The arguments to the operations (e.g., add, $delete$, and $search$) are generated uniformly at random from the key range under evaluation. Each such input string is evaluated on the GPU as well as the CPU thrice and we report the results based on the median execution time for each experiment.

For each data structure we have optimized the number of threads per block and the number of thread blocks for the CUDA kernel. For linked list, we use 64 threads per block, while for the other three data structures, we use 512 threads per block. To determine the number of thread blocks, we execute the kernel for four different configurations. In these configurations the number of thread blocks is selected such that a thread executing the kernel carries out 1, 4, 8, or 16 operations from the input string. We pick the best of these four configurations. In most of the cases, the best configuration is the one in which a thread carries out just one operation in the kernel. This configuration essentially maximizes the number of CUDA threads. For the CPU executions, it is not always the case that implementing a lock-free data structure on 24 threads offers the best performance. We pick the thread count (at most 24) that achieves the best performance. In summary, in each result that we present in the next section, we report/compare the best possible performance on the GPU and the CPU among the configurations that we have considered.

Our lock-free hash table implements ten thousand buckets and the skip list uses $p = 0.5$ and 32 levels.

V. Performance Results

We evaluate the lock-free linked list, hash table, and skip list on two different types of operation mixes. One is $search$-dominated and has 20% add, 20% $delete$, and 60% $search$ operations. The other one is add and $delete$-dominated and has 40% add, 40% $delete$, and 20% $search$. The priority queue is also evaluated on two types of operation mixes. One is unbiased and has 50% add and 50% $deleteMin$ operations, while the other one is add-dominated and has 80% add and 20% $deleteMin$ operations. Each data structure is evaluated on four key ranges and ten different operation counts, as already mentioned.

Figure 4 shows the speedup achieved by our CUDA implementation of the lock-free linked list over the CPU implementation. The upper panel shows the results for an operation mix of $[20, 20, 60]$ i.e., 20% add, 20% $delete$, and 60% $search$ operations. The lower panel is for an operation mix of $[40, 40, 20]$. In each panel, each of the four groups of bars represents one key range indicated on the x-axis. Within each group, the leftmost bar represents an input operation string with ten thousand operations and the rightmost bar represents an operation count of hundred thousand. The number of operations increases in steps of ten thousand within a group of bars from left to right.

While the speedup trends observed in both the operation mixes is same, we note that the GPU implementation loses its advantage

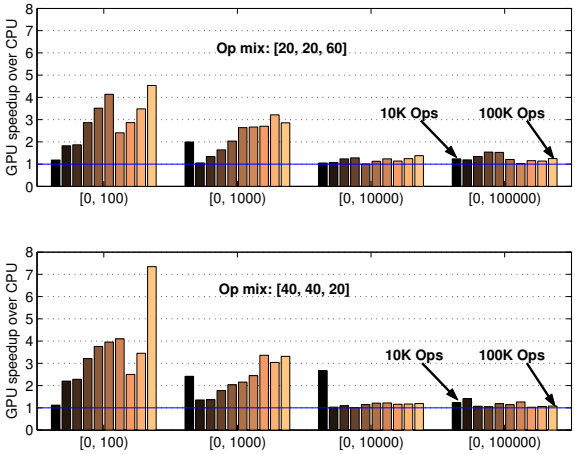

Figure 4. Speedup of lock-free linked list on GPU relative to CPU.

over the CPU as the key range increases. For small key ranges, many of the add operations actually do not insert any node in the list because the element to be added is already in the list. This reduces the overall contention and control flow divergences in the GPU. Also, we observe that for small to medium key ranges, increasing the total number of operations exposes more concurrency in the GPU and leads to better speedup. For large key ranges, the GPU and CPU offer almost the same performance for the lock-free linked list. Even though the GPU implementation can expose more concurrency, the overhead of the slow atomicCAS operations outweighs this benefit. The atomicCAS operations execute in the global memory of the GPU, while in the CPU, the atomic instructions execute in the coherent L1 caches and hence, are much faster. Overall, the GPU implementation benefits moderately for small to medium key ranges with the best speedup being 7.4 compared to the CPU implementation.

Figure 5 shows the speedup results for our lock-free hash table. As expected, the hash table benefits significantly from the GPU implementation and the benefits are consistent across all the key ranges. The best achieved speedup is 11.3 compared to the CPU implementation. The primary advantage of the hash table is that the contention is distributed across multiple buckets, which naturally exposes more concurrency.

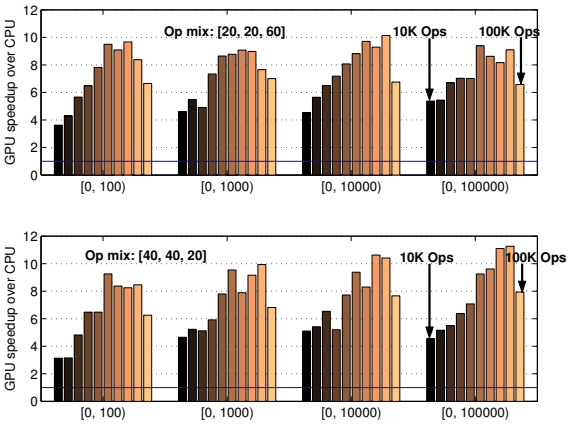

Figure 5. Speedup of lock-free hash table on GPU relative to CPU.

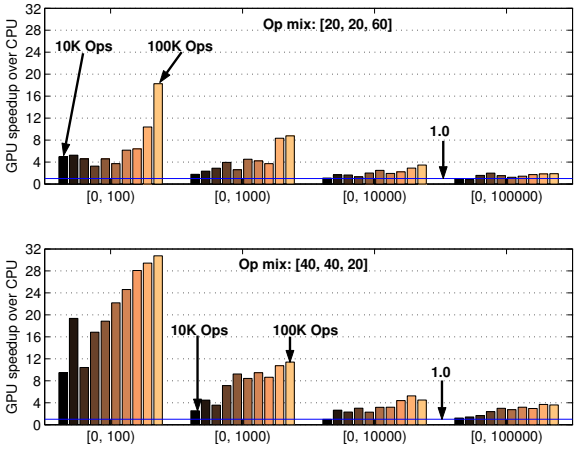

Figure 6. Speedup of lock-free skip list on GPU relative to CPU.

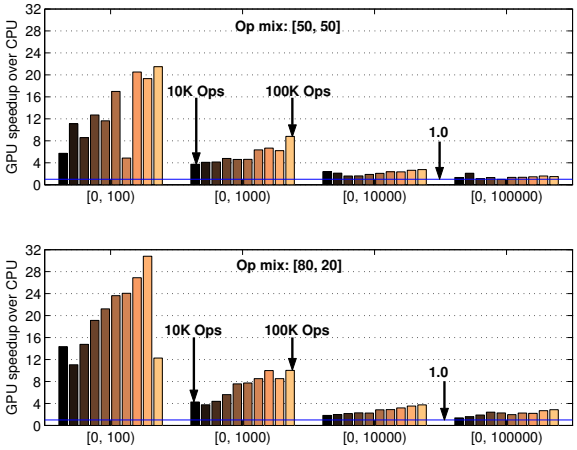

Figure 7. Speedup of lock-free priority queue on GPU relative to CPU.

Figure 6 presents the performance results for our lock-free skip list implementation. This data structure enjoys significant speedup on the GPU compared to the CPU implementation for small and medium key ranges. As the key range increases, more add and $delete$ operations have to modify the data structure through atomicCAS operations. As a result, the speedup drops sharply due to the overhead of atomicCAS operations and complicated control flow of the implementation. Nonetheless, the $[40, 40, 20]$ mix still enjoys a speedup of nearly 4.0 on large key ranges with hundred thousand operations. Interestingly, we observe that as the percentage of add operations increases, the speedup also increases (compare the upper panel with the lower panel in Figure 6). This is primarily because with more add operations, the expected number of shortcuts in the skip list increases leading to relatively less number of traversed nodes (log is a slowly increasing function) and hence, less control flow in the traversal. This is the reason for better performance in the GPU. As expected, this phenomenon affects the GPU performance much more than the CPU performance. Overall, the best speedup achieved by the skip list implementation is 30.7 compared to the CPU implementation.

Figure 7 shows the performance results for our lock-free priority queue. The upper panel shows the results for an input operation string with equal mix of add and $deleteMin$, while the lower panel has 80% add and 20% $deleteMin$ operations. The performance trends are similar to that of the skip list, as expected. The best speedup achieved by the priority queue is 30.8 on the GPU compared to the CPU implementation.

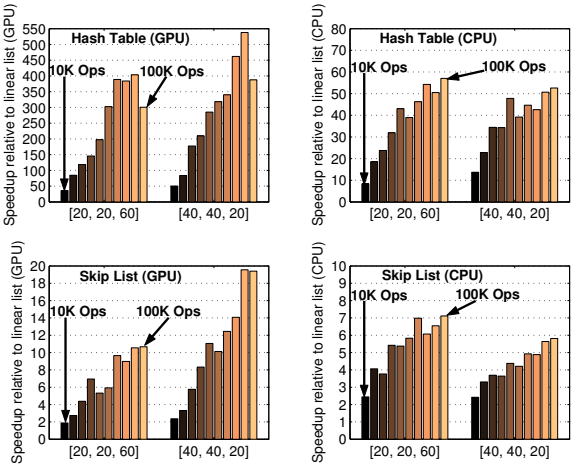

Figure 8. Comparison of hash table and skip list relative to linear list.

We close the discussion of the results by comparing the performance of the lock-free hash table and the skip list with that of the lock-free linked list on the GPU as well as the CPU. We carry out this comparison for the largest key range among the four ranges. The top-left panel of Figure 8 shows the performance of the lock-free hash table relative to the lock-free linked list when both are executed on the GPU. The top-right panel shows the same comparison when both are executed on the CPU. Interestingly, the hash table is better than the linked list by a factor of 36 to 538 on the GPU. However, this performance ratio varies from 8 to 54 on the CPU. Referring back to Figure 4, we note that the lock-free linked list delivers almost similar performance on the CPU and the GPU for the largest key range. Therefore, we can conclude from the top-left and top-right panels of Figure 8 that the GPU helps expose more performance potential of the lock-free hash table than what a CPU implementation can do. This is primarily due to the fact that the GPU implementation can leverage much higher degree of concurrency than a CPU implementation. We find that for the largest key range, the throughput of the lock-free hash table on GPU varies from 28.6 MOPS (million operations per second) to 98.9 MOPS for the $[20, 20, 60]$ operation mix. For the $[40, 40, 20]$ mix, the throughput ranges from 20.8 MOPS to 72.0 MOPS.

The lower left panel of Figure 8 shows the performance of the lock-free skip list relative to the lock-free linked list when both are executed on the GPU. The lower right panel shows the same comparison when both are executed on the CPU. Here also we observe that the GPU exposes bigger performance potential of the lock-free skip list than the CPU. The skip list is about two to twenty times better than the linked list when both are implemented for the GPU. However, as we speculated in Section III-C, the lock-free hash table offers far better performance than the skip list on both GPU and CPU. The performance gap between the hash table and the skip list is much bigger on the GPU due to the performance drawback of the control flow divergences of the skip list implementation.

VI. Summary

This study evaluates the performance of four lock-free data structures on Fermi GPU. All these data structures build upon

the lock-free linked list implementation. Our evaluation shows that for small to medium key ranges, all the four data structures, namely, linked list, hash table, skip list, and priority queue, enjoy moderate to high speedup (up to 30.8) on the GPU over multi-threaded lock-free CPU implementations. For large key ranges, the linked list, skip list, and priority queue do not benefit much from the GPU implementation due to slow atomicCAS operations on the global memory and the complex control flow of some of the lock-free implementations. The hash table emerges the best lock-free data structure among the ones we have evaluated for carrying out addition, deletion, and search operations on arbitrary key ranges. It offers consistently good performance for small as well as large key ranges with the GPU implementation showing significant benefit over the CPU implementation (more than eleven times better performance).

We feel that the best way to improve the performance of irregular pointer-based data structures on the GPUs is to incorporate support for fast atomic operations, especially compare-and-swap. Although our lock-free implementations already achieve good performance compared to equivalent CPU implementations, this study offers a strong motivation for exploring architectural techniques to further improve the performance of atomicCAS operations on the GPU. Techniques to reduce the overhead of control flow divergence in the GPU architectures would also be beneficial to the lock-free implementations.

ACKNOWLEDGMENT

The authors thank Nvidia for providing the GPU hardware as a professor partnership program equipment grant.

REFERENCES

[1] J. Aspnes, M. Herlihy, and N. Shavit. Counting Networks. In *Journal of the ACM*, **41**(5):1020–1048, September 1994.

[2] A. Basu et al. Scavenger: A New Last Level Cache Architecture with Global Block Priority. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 421–432, December 2007.

[3] R. Bayer and M. Schkolnick. Concurrency of Operations on B-Trees. In *Acta Informatica*, **9**: 1–21, 1977.

[4] D. Cederman, B. Chatterjee, and P. Tsigas. Understanding the Performance of Concurrent Data Structures on Graphics Processors. In *Proceedings of the 18th Euro-Par*, pages 883–894, August 2012.

[5] D. Cederman, P. Tsigas, and M. T. Chaudhry. Towards a Software Transactional Memory for Graphics Processors. In *Eurographics Symposium on Parallel Graphics and Visualization*, pages 121–129, 2010.

[6] C. Ellis. Concurrency in Linear Hashing. In *ACM Transactions on Database Systems*, **12**(2):195–217, June 1987.

[7] K. Fraser. "Practical Lock-Freedom". *PhD dissertation*, Kings College, University of Cambridge, September 2003.

[8] M. Greenwald. Two-handed Emulation: How to Build Non-blocking Implementations of Complex Data Structures using DCAS. In *Proceedings of the Twenty-First Annual ACM Symposium on Principles of Distributed Computing*, pages 260–269, July 2002.

[9] T. Harris. A Pragmatic Implementation of Non-blocking Linked Lists. In *Proceedings of the 15th International Conference on Distributed Computing*, pages 300–314, October 2001.

[10] M. Herlihy and N. Shavit. "The Art of Multiprocessor Programming". Morgan Kaufmann Publishers.

[11] M. Herlihy. Wait-free Synchronization. In *ACM Transactions on Programming Languages and Systems*, **13**(1):124–149, January 1991.

[12] M. Herlihy and J. M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. In *ACM Transactions on Programming Languages and Systems*, **12**(3):463–492, July 1990.

[13] M. Hsu and W. P. Yang. Concurrent Operations in Extensible Hashing. In *Proceedings of the Twelfth International Conference on Very Large Data Bases*, pages 241–247, August 1986.

[14] G. C. Hunt et al. An Efficient Algorithm for Concurrent Priority Queue Heaps. In *Information Processing Letters*, **60**(3):151–157, November 1996.

[15] A. Ioannou and M. Katevenis. Pipelined Heap (Priority Queue) Management for Advanced Scheduling in High-Speed Networks. In *IEEE/ACM Transactions on Networking*, **15**(2):450–461, April 2007.

[16] V. Kumar. Concurrent Operations on Extendible Hashing and Its Performance. In *Communications of the ACM*, **33**(6):681–694, June 1990.

[17] L. Lamport. A New Solution to Dijkstra's Concurrent Programming Problem. In *Communications of the ACM*, **17**(8):453–455, August 1974.

[18] I. Lotan and N. Shavit. Skiplist-based Concurrent Priority Queues. In *Proceedings of the 14th International Parallel and Distributed Processing Symposium*, pages 263–268, May 2000.

[19] M. M. Michael. High Performance Dynamic Lock-free Hash Tables and List-based Sets. In *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 73–82, August 2002.

[20] Nvidia. Nvidia CUDA C Programming Guide v4.1. November 2011.

[21] W. Pugh. Skip Lists: A Probabilistic Alternative to Balanced Trees. In *Communications of the ACM*, **33**(6):668–676, June 1990.

[22] W. Pugh. Concurrent Maintenance of Skip Lists. *Technical Report CS-TR-2222.1*, Department of Computer Science, University of Maryland, April 1989.

[23] V. N. Rao and V. Kumar. Concurrent Access of Priority Queues. In *IEEE Transactions on Computers*, **37**(12):1657–1665, December 1988.

[24] O. Shalev and N. Shavit. Split-ordered Lists: Lock-free Extensible Hash Tables. In *Journal of the ACM*, **53**(3):379–405, May 2006.

[25] N. Shavit and A. Zemach. Diffracting Trees. In *ACM Transactions on Computer Systems*, **14**(4):385–428, November 1996.

[26] J. A. Stuart and J. D. Owens. Efficient Synchronization Primitives for GPUs. In *arXiv:1110.4623v1* $[cs.OS]$, October 2011.

[27] H. Sundell and P. Tsigas. Fast and Lock-free Concurrent Priority Queues for Multi-thread Systems. In *Journal of Parallel and Distributed Computing*, **65**(5):609–627, May 2005.

[28] H. Sundell and P. Tsigas. Scalable and Lock-free Concurrent Dictionaries. In *Proceedings of the ACM Symposium on Applied Computing*, pages 1438–1445, March 2004.

[29] J. D. Valois. Lock-free Linked Lists using Compare-and-swap. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 214–222, August 1995.

[30] S. Xiao and W-c. Feng. Inter-block GPU Communication via Fast Barrier Synchronization. In *Proceedings of the 24th IEEE International Symposium on Parallel and Distributed Processing*, April 2010.